The TANGO Control System Manual
Version 8.0

The TANGO Team

June 6, 2012

Contents

Are you ready to dance the TANGO ?

Chapter 1

Introduction

1.1 Introduction to device server

Device servers were first developed at the European Synchrotron radiation Facility (ESRF) for controlling the 6 Gev synchrotron radiation source. This document is a Programmer's Manual on how to write TANGO device servers. It will not go into the details of the ESRF, nor its Control System nor any of the specific device servers in the Control System. The role of this document is to help programmers faced with the task of writing TANGO device servers.

Device servers have been developed at the ESRF in order to solve the main task of Control Systems viz provide read and write access to all devices in a distributed system. The problem of distributed device access is only part of the problem however. The other part of the problem is providing a programming framework for a large number of devices programmed by a large number of programmers each having different levels of experience and style.

Device servers have been written at the ESRF for a large variety of different devices. Devices vary from serial line devices to devices interfaced by field-bus to memory mapped VME cards or PC cards to entire data acquisition systems. The definition of a device depends very much on the user's requirements. In the simple case a device server can be used to hide the serial line protocol required to communicate with a device. For more complicated devices the device server can be used to hide the entire complexity of the device timing, configuration and acquisition cycle behind a set of high level commands.

In this manual the process of how to write TANGO client (applications) and device servers will be treated. The manual has been organized as follows :

- A getting started chapter.
- The TANGO device server model is treated in chapter 3
- Generalities on the Tango Application Programmer Interfaces are given in chapter 4
- The TANGO Java client Application Programmer Interface is described in chapter 5
- Chapter 6 describes the TANGO C++ client Application Programmer Interface
- Chapter 7 is an a programmer's guide for the Tango Application ToolKit (TangoATK). This is a Java toolkit to help Tango Java application developers.
- How to write a TANGO device server is explained in chapter 8
- Chapter 9 describes advanced Tango features

Throughout this manual examples of source code will be given in order to illustrate what is meant. Most examples have been taken from the StepperMotor class - a simulation of a stepper motor which illustrates how a typical device server for a stepper motor at the ESRF functions.

1.2 Device server history

The concept of using device servers to access devices was first proposed at the ESRF in 1989. It has been successfully used as the heart of the ESRF Control System for the institute accelerator complex. This Control System has been named TACO[1]. Then, it has been decided to also used TACO to control devices in the beam-lines. Today, more than 30 instances of TACO are running at the ESRF. The main technologies used within TACO are the leading technologies of the 80's. The Sun Remote Procedure Call (RPC) is used to communicate over the network between device server and applications, OS-9 is used on the front-end computers, C is the reference language to write device servers and clients and the device server framework follows the MIT Widget model. In 1999, a renewal of the control system was started. In June 2002, Soleil and ESRF offically decide to collaborate to develop this renewal of the old TACO control system. Soleil is a French synchrotron radiation facility currently under construction in the Paris suburbs. See [5] to get all information about Soleil. In December 2003, Elettra joins the club. Elettra is an Italian synchrotron radiation facility located in Trieste. See [20] to get all information about Elettra. Then, beginning of 2005, ALBA also decided to join. ALBA is a Spanish synchrotron radiation facility located in Barcelona. See [4] to get all information about ALBA. The new version of the Alba/Elettra/ESRF/Soleil control system is named TANGO[2] and is based on the 21 century technologies :

- CORBA[3] and ZMQ[23] to communicate between device server and clients
- C++, Python and Java as reference programming languages
- Linux and Windows as operating systems
- Modern object oriented design patterns

---

[1] TACO stands for **T**elescope and **A**ccelerator **C**ontrolled with **O**bjects
[2] TANGO stands for **TA**co **N**ext **G**eneration **O**bject
[3] CORBA stands for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

Chapter 2

Getting Started

2.1 A Java TANGO client

The quickest way of getting started is by studying this example:

```
/*
 *   Example of a client using the TANGO Api
 */
import fr.esrf.Tango.*;
import fr.esrf.TangoDs.*;
import fr.esrf.TangoApi.*;

public class TestDevice
{
  public static void main (String args[])
  {
    try
    {
// Connect to the device.

          DeviceProxy       dev = new DeviceProxy("my/serial/device");

// Send a write command to the device

          DeviceData argin = new DeviceData();
          argin.insert("Hello World !");
          dev.command_inout("DevWriteMessage", argin);

// Send a read command to the device

          DeviceData argout = dev.command_inout("DevReadMessage");
          String  received = argout.extractString();
          System.out.println(received);

// Read a device attribute (double data type)

          DeviceAttribute da = dev.read_attribute("TheAttr");
          System.out.println("\nRead " + da.extractDouble() + " on " +   dev.getName());
```

```
        }
        catch (DevFailed e)
        {
           Except.print_exception(e);
        }
      }
    }
```

Modify this example to fit your device server or client's needs, compile it.
Do not forget when you start it to set the parameter TANGO_HOST with <host_name>:<port_number> (i.e. java -DTANGO_HOST=tango:20000 TestDevice).
And forget about those painful early Tango days when you had to learn CORBA and manipulate Any's.
Life is going to easy and fun from now on.

2.2 A C++ TANGO client

The quickest way of getting started is by studying this example :

```
/*
 * example of a client using the TANGO C++ api.
 */
#include <tango.h>
using namespace Tango;
int main(unsigned int argc, char **argv)
{
    try
    {
//
// create a connection to a TANGO device
//

        DeviceProxy *device = new DeviceProxy("sys/database/2");

//
// Ping the device
//

        device->ping();

//
// Execute a command on the device and extract the reply as a string
//

        string db_info;
        DeviceData cmd_reply;
        cmd_reply = device->command_inout("DbInfo");
        cmd_reply >> db_info;
        cout << "Command reply " << db_info << endl;

//
```

```
// Read a device attribute (string data type)
//

        string spr;
        DeviceAttribute att_reply;
        att_reply = device->read_attribute("StoredProcedureRelease");
        att_reply >> spr;
        cout << "Database device stored procedure release: " << spr << endl;
    }
    catch (DevFailed &e)
    {
        Except::print_exception(e);
        exit(-1);
    }
}
```

Modify this example to fit your device server or client's needs, compile it and link with the library -ltango. Forget about those painful early TANGO days when you had to learn CORBA and manipulate Any's. Life's going to easy and fun from now on !

2.3 A TANGO device server

The code given in this chapter as example has been generated using POGO. Pogo is a code generator for Tango device server. See [15] for more information about POGO. The following examples briefly describe how to write device class with commands which receives and return different kind of Tango data types and also how to write device attributes The device class implements 5 commands and 3 attributes. The commands are :

- The command **DevSimple** deals with simple Tango data type
- The command **DevString** deals with Tango strings
- **DevArray** receive and return an array of simple Tango data type
- **DevStrArray** which does not receive any data but which returns an array of strings
- **DevStruct** which also does not receive data but which returns one of the two Tango composed types (DevVarDoubleStringArray)

For all these commands, the default behavior of the state machine (command always allowed) is acceptable. The attributes are :

- A spectrum type attribute of the Tango string type called **StrAttr**
- A readable attribute of the Tango::DevLong type called **LongRdAttr**. This attribute is linked with the following writable attribute
- A writable attribute also of the Tango::DevLong type called **LongWrAttr**.

2.3.1 The commands and attributes code in C++

For each command called DevXxxx, pogo generates in the device class a method named dev_xxx which will be executed when the command is requested by a client. In this chapter, the name of the device class is *DocDs*

2.3.1.1 The DevSimple command

This method receives a Tango::DevFloat type and also returns a data of the Tango::DevFloat type which is simply the double of the input value. The code for the method executed by this command is the following:

```
1  Tango::DevFloat DocDs::dev_simple(Tango::DevFloat argin)
2  {
3          Tango::DevFloat argout ;
4          DEBUG_STREAM << "DocDs::dev_simple(): entering... !" << endl;
5
6          //      Add your own code to control device here
7
8          argout = argin * 2;
9          return argout;
10 }
```

This method is fairly simple. The received data is passed to the method as its argument. It is doubled at line 8 and the method simply returns the result.

2.3.1.2 The DevArray command

This method receives a data of the Tango::DevVarLongArray type and also returns a data of the Tango::DevVarLongArray type. Each element of the array is doubled. The code for the method executed by the command is the following :

```
Tango::DevVarLongArray *DocDs::dev_array(const Tango::DevVarLongArray *argin)
{
        //      POGO has generated a method core with argout allocation.
        //      If you would like to use a static reference without copying,
        //      See "TANGO Device Server Programmer's Manual"
        //              (chapter x.x)
        //------------------------------------------------------------
        Tango::DevVarLongArray  *argout  = new Tango::DevVarLongArray();

        DEBUG_STREAM << "DocDs::dev_array(): entering... !" << endl;

        //      Add your own code to control device here

        long argin_length = argin->length();
        argout->length(argin_length);
        for (int i = 0;i < argin_length;i++)
                (*argout)[i] = (*argin)[i] * 2;

        return argout;
}
```

The argout data array is created at line 8. Its length is set at line 15 from the input argument length. The array is populated at line 16,17 and returned. This method allocates memory for the argout array. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). It is also possible to return data from a statically allocated array without copying. Look at chapter 8.2 for all the details.

2.3.1.3 The DevString command

This method receives a data of the Tango::DevString type and also returns a data of the Tango::DevString type. The command simply displays the content of the input string and returns a hard-coded string. The code for the method executed by the command is the following :

```
1  Tango::DevString DocDs::dev_string(Tango::DevString argin)
2  {
3          //      POGO has generated a method core with argout allocation.
4          //      If you would like to use a static reference without copying,
5          //      See "TANGO Device Server Programmer's Manual"
6          //              (chapter x.x)
7          //------------------------------------------------------------
8          Tango::DevString        argout;
9          DEBUG_STREAM << "DocDs::dev_string(): entering... !" << endl;
10
11         //      Add your own code to control device here
12
13         cout << "the received string is " << argin << endl;
14
15         string str("Am I a good Tango dancer ?");
16         argout = new char[str.size() + 1];
17         strcpy(argout,str.c_str());
18
19         return argout;
20 }
```

The argout string is created at line 8. Internally, this method is using a standard C++ string. Memory for the returned data is allocated at line 16 and is initialized at line 17. This method allocates memory for the argout string. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). It is also possible to return data from a statically allocated string without copying. Look at chapter 8.2 for all the details.

2.3.1.4 The DevStrArray command

This method does not receive input data but returns an array of strings (Tango::DevVarStringArray type). The code for the method executed by this command is the following:

```
Tango::DevVarStringArray *DocDs::dev_str_array()
{
        //      POGO has generated a method core with argout allocation.
        //      If you would like to use a static reference without copying,
```

```
5           //      See "TANGO Device Server Programmer's Manual"
6           //              (chapter x.x)
7           //------------------------------------------------------------
8           Tango::DevVarStringArray        *argout  = new Tango::DevVarStringArray();
9
10          DEBUG_STREAM << "DocDs::dev_str_array(): entering... !" << endl;
11
12          //      Add your own code to control device here
13
14          argout->length(3);
15          (*argout)[0] = CORBA::string_dup("Rumba");
16          (*argout)[1] = CORBA::string_dup("Waltz");
17          string str("Jerck");
18          (*argout)[2] = CORBA::string_dup(str.c_str());
19          return argout;
20  }
```

The argout data array is created at line 8. Its length is set at line 14. The array is populated at line 15,16 and 18. The last array element is initialized from a standard C++ string created at line 17. Note the usage of the *string_dup* function of the CORBA namespace. This is necessary for strings array due to the CORBA memory allocation schema.

2.3.1.5 The DevStruct command

This method does not receive input data but returns a structure of the Tango::DevVarDoubleStringArray type. This type is a composed type with an array of double and an array of strings. The code for the method executed by this command is the following:

```
Tango::DevVarDoubleStringArray *DocDs::dev_struct()
{
        //      POGO has generated a method core with argout allocation.
        //      If you would like to use a static reference without copying,
        //      See "TANGO Device Server Programmer's Manual"
        //              (chapter x.x)
        //------------------------------------------------------------
        Tango::DevVarDoubleStringArray  *argout  = new Tango::DevVarDoubleStringArray();

        DEBUG_STREAM << "DocDs::dev_struct(): entering... !" << endl;

        //      Add your own code to control device here

        argout->dvalue.length(3);
        argout->dvalue[0] = 0.0;
        argout->dvalue[1] = 11.11;
        argout->dvalue[2] = 22.22;

        argout->svalue.length(2);
        argout->svalue[0] = CORBA::string_dup("Be Bop");
        string str("Smurf");
        argout->svalue[1] = CORBA::string_dup(str.c_str());

```

```
24          return argout;
25  }
```

The argout data structure is created at line 8. The length of the double array in the output structure is set at line 14. The array is populated between lines 15 and 17. The length of the string array in the output structure is set at line 19. This string array is populated between lines 20 an 22 from a hard-coded string and from a standard C++ string. This method allocates memory for the argout data. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). Note the usage of the *string_dup* function of the CORBA namespace. This is necessary for strings array due to the CORBA memory allocation schema.

2.3.1.6 The three attributes

Some data have been added to the definition of the device class in order to store attributes value. These data are (part of the class definition) :

```
protected :
        //      Add your own data members here
        //-----------------------------------------
        Tango::DevString        attr_str_array[5];
        Tango::DevLong          attr_rd;
        Tango::DevLong          attr_wr;
```

One data has been created for each attribute. As the StrAttr attribute is of type spectrum with a maximum X dimension of 5, an array of length 5 has been reserved.

Several methods are necessary to implement these attributes. One method to read the hardware which is common to all "readable" attributes plus one "read" method for each readable attribute and one "write" method for each writable attribute. The code for these methods is the following :

```
void DocDs::read_attr_hardware(vector<long> &attr_list)
{
    DEBUG_STREAM << "DocDs::read_attr_hardware(vector<long> &attr_list) entering... "<< endl;
// Add your own code here

    string att_name;
    for (long i = 0;i < attr_list.size();i++)
    {
        att_name = dev_attr->get_attr_by_ind(attr_list[i]).get_name();

        if (att_name == "LongRdAttr")
        {
            attr_rd = 5;
        }
    }
}
```

```
18 void DocDs::read_LongRdAttr(Tango::Attribute &attr)
19 {
20     DEBUG_STREAM << "DocDs::read_LongRdAttr(Tango::Attribute &attr) entering... "<< endl;
21
22     attr.set_value(&attr_rd);
23 }
24
25 void DocDs::read_LongWrAttr(Tango::Attribute &attr)
26 {
27     DEBUG_STREAM << "DocDs::read_LongWrAttr(Tango::Attribute &attr) entering... "<< endl;
28
29     attr.set_value(&attr_wr);
30 }
31
32 void DocDs::write_LongWrAttr(Tango::WAttribute &attr)
33 {
34     DEBUG_STREAM << "DocDs::write_LongWrAttr(Tango::WAttribute &attr) entering... "<< endl;
35
36     attr.get_write_value(attr_wr);
37     DEBUG_STREAM << "Value to be written = " << attr_wr << endl;
38 }
39
40 void DocDs::read_StrAttr(Tango::Attribute &attr)
41 {
42     DEBUG_STREAM << "DocDs::read_StrAttr(Tango::Attribute &attr) entering... "<< endl;
43
44     attr_str_array[0] = CORBA::string_dup("Rock");
45     attr_str_array[1] = CORBA::string_dup("Samba");
46
47     attr_set_value(attr_str_array, 2);
48 }
```

The *read_attr_hardware()* method is executed once when a client execute the read_attributes CORBA request whatever the number of attribute to be read is. The rule of this method is to read the hardware and to store the read values somewhere in the device object. In our example, only the LongRdAttr attribute internal value is set by this method at line 13. The method *read_LongRdAttr()* is executed by the read_attributes CORBA call when the LongRdAttr attribute is read but after the read_attr_hardware() method has been executed. Its rule is to set the attribute value in the TANGO core classes object representing the attribute. This is done at line 22. The method *read_LongWrAttr()* will be executed when the LongWrAttr attribute is read (after the *read_attr_hardware()* method). The attribute value is set at line 29. In the same manner, the method called *read_StrAttr()* will be executed when the attribute StrAttr is read. Its value is initialized in this method at line 44 and 45 with the *string_dup* CORBA function. The *write_LongWrAttr()* method is executed when the LongWrAttr attribute value is set by a client. The new attribute value coming from the client is stored in the object data at line 36.

Pogo also generates a file called "DocDsStateMachine.cpp" (for a Tango device server class called DocDs). This file is used to store methods coding the device state machine. By default a allways allowed state machine is provided. For more information about coding the state machine, refer to the chapter "Writing a device server".

2.3.2 The commands and attributes code in java

For each command called DevXxxx, pogo generates in the device class a method named dev_xxx which will be executed when the command is requested by a client. In this chapter, the name of the device class is *DocDs*

2.3.2.1 The DevSimple command

This method receives a Tango DevFloat type and also returns a data of the Tango DevFloat type which is simply the double of the input value. Using java, the Tango::DevFloat type is mapped to classical java float type. The code for the method executed by this command is the following:

```
1  public float dev_simple(float argin) throws DevFailed
2  {
3      float   argout =  (float)0;
4
5      Util.out2.println("Entering dev_simple()");
6
7  // ---Add your Own code to control device here ---
8
9      argout = argin * 2;
10     return argout;
11 }
```

This method is fairly simple. The received data is passed to the method as its argument. It is doubled at line 9 and the method simply returns the result.

2.3.2.2 The DevArray command

This method receives a data of the Tango::DevVarLongArray type and also returns a data of the Tango::DevVarLongArray type. Each element of the array is doubled. Using java, the Tango DevVarLongArray type is mapped to an array of java long. The code for the method executed by the command is the following :

```
1  public int[] dev_array(int[] argin) throws DevFailed
2  {
3      int[]   argout = new int[argin.length];
4
5      Util.out2.println("Entering dev_array()");
6
7  // ---Add your Own code to control device here ---
8
9      for (int i = 0;i < argin.length;i++)
10         argout[i] = argin[i] * 2;
11     return argout;
12 }
```

The argout data array is created at line 3. The array is populated at line 9,10 and returned.

2.3.2.3 The DevString command

This method receives a data of the Tango DevString type and also returns a data of the Tango DevString type. The command simply displays the content of the input string and returns a hard-coded string. Using java, the Tango DevString type simply maps to java String.The code for the method executed by the command is the following :

```
1  public String dev_string(String argin) throws DevFailed
2  {
3      Util.out2.println("Entering dev_string()");
4
5  // ---Add your Own code to control device here ---
6
7      System.out.println("the received string is "+argin);
8
9      String argout = new String("Am I a good Tango dancer ?");
10     return argout;
11 }
```

The argout string is created at line 9.

2.3.2.4 The DevStrArray command

This method does not receive input data but returns an array of strings (Tango DevVarStringArray type). Using java, the Tango DevVarStringArray type maps to an array of java String. The code for the method executed by this command is the following:

```
1  public String[] dev_str_array() throws DevFailed
2  {
3
4      Util.out2.println("Entering dev_str_array()");
5
6  // ---Add your Own code to control device here ---
7
8      String[] argout = new String[3];
9      argout[0] = new String("Rumba");
10     argout[1] = new String("Waltz");
11     argout[2] = new String("Jerck");
12     return argout;
13 }
```

The argout data array is created at line 8. The array is populated at line 9,10 and 11.

2.3.2.5 The DevStruct command

This method does not receive input data but returns a structure of the Tango DevVarDoubleStringArray type. This type is a composed type with an array of double and an array of strings. This is mapped to a specific java class called DevVarDoubleStringArray. The code for the method executed by this command is the following:

```
1 public DevVarDoubleStringArray dev_struct() throws DevFailed
2 {
3     DevVarDoubleStringArray argout = new DevVarDoubleStringArray();
4
5     Util.out2.println("Entering dev_struct()");
6
7 // ---Add your Own code to control device here ---
8
9     argout.dvalue = new double[3];
10    argout.dvalue[0] = 0.0;
11    argout.dvalue[1] = 11.11;
12    argout.dvalue[2] = 22.22;
13
14    argout.svalue = new String[2];
15    argout.svalue[0] = new String("Be Bop");
16    argout.svalue[1] = new String("Smurf");
17
18    return argout;
19 }
```

The argout data structure is created at line 3. The double array in the output structure is created at line 9. The array is populated between lines 10 and 12. The string array in the output structure is created at line 14. This string array is populated between lines 15 and 16 from hard-coded strings.

2.3.2.6 The three attributes

Some data have been added to the definition of the device class in order to store attributes value. These data are (part of the class definition) :

```
protected String[]      attr_str_array = new String[5];
protected int           attr_rd;
protected int           attr_wr;
```

One data has been created for each attribute. As the StrAttr attribute is of type spectrum with a maximum X dimension of 5, an array of length 5 has been reserved.

Unfortunately, Java Tango device server are not at the same level of development than C++ device servers. This is why they are not written exactly the same way than C++ device servers. Three methods are necessary to implement these attributes. The code for these methods is the following :

```
public void write_attr_hardware(Vector attr_list)
{
    Util.out2.println("In write_attr_hardware for "+attr_list.size()+" attribute(s)");

    for (int i=0 ; i<attr_list.size() ; i++)
```

```
    {
        int ind = ((Integer)(attr_list.elementAt(i))).intValue();
        WAttribute att = dev_attr.get_w_attr_by_ind(ind);
        String attr_name = att.get_name();

        //      Switch on attribute name
        //--------------------------------
        if (attr_name.equals("LongWrAttr") == true)
        {
            //      Add your own code here
            attr_wr = att.get_lg_write_value();
            System.out.println("Value to be written = "+attr_wr);
        }
    }
}


public void read_attr_hardware(Vector attr_list)
{
    Util.out2.println("In read_attr_hardware for "+attr_list.size()+" attribute(s)");

    //      Add you own code here
    //--------------------------------

    for (int i=0; i<attr_list.size() ; i++)
    {
        int ind = ((Integer)(attr_list.elementAt(i))).intValue();
        Attribute att = dev_attr.get_attr_by_ind(ind);
        String attr_name = attr_list.elementAt(i);

        if (attr_name.equals("LongRdAttr") == true)
        {
            attr_rd = 5;
        }
        else if (attr_name.equals("StrAttr") == true)
        {
            attr_str_array[0] = new String("Rock");
            attr_str_array[1] = new String("Samba");
        }
    }
}


public void read_attr(Attribute attr) throws DevFailed
{
    String attr_name = attr.get_name();
    Util.out2.println("In read_attr for attribute "+attr_name);

    //      Switch on attribute name
    //--------------------------------
    if (attr_name.equals("LongWrAttr") == true)
    {
        //      Add your own code here
        attr.set_value(attr_wr);
```

```
60      }
61      if (attr_name.equals("LongRdAttr") == true)
62      {
63          //      Add your own code here
64          attr.set_value(attr_rd);
65      }
66      if (attr_name.equals("StrAttr") == true)
67      {
68          //      Add your own code here
69          attr.set_value(attr_str_array);
70      }
71  }
```

The *write_attr_hardware()* method is executed when an attribute value is set by a client. In our example only one attribute is writable (the LongWrAttr attribute). The new attribute value coming from the client is stored in the object data at line 16. The *read_attr_hardware()* method is executed once when a client execute the read_attributes CORBA request. The rule of this method is to read the hardware and to store the read values somewhere in the device object. In our example, the LongRdAttr attribute internal value is set by this method at line 38 at the StrAttr attribute internal value is set at lines 42 and 43. The method *read_attr()* is executed for each attribute to be read by the read_attributes CORBA call. Its rule is to set the attribute value in the TANGO core classes object representing the attribute. This is done at line 64 for the LongRdAttr attribute, at line 59 for the LongWrAttr attribute and at line 69 for the StrAttr attribute

Chapter 3

The TANGO device server model

This chapter will present the TANGO device server object model hereafter referred as TDSOM. First, it will introduce CORBA. Then, it will describe each of the basic features of the TDSOM and their function. The TDSOM can be divided into the following basic elements - the *device*, the *server*, the *database* and the *application programmers interface*. This chapter will treat each of the above elements separately.

3.1 Introduction to CORBA

CORBA is a definition of how to write object request brokers (ORB). The definition is managed by the Object Management Group (OMG [1]). Various commercial and non-commercial implementations exist for CORBA for all the mainstream operating systems. CORBA uses a programming language independent definition language (called IDL) to defined network object interfaces. Language mappings are defined from IDL to the main programming languages e.g. C++, Java, C, COBOL, Smalltalk and ADA. Within an interface, CORBA defines two kinds of actions available to the outside world. These actions are called **attributes** and **operations**.

Operations are all the actions offered by an interface. For instance, within an interface for a Thermostat class, operations could be the action to read the temperature or to set the nominal temperature. An attribute defines a pair of operations a client can call to send or receive a value. For instance, the position of a motor can be defined as an attribute because it is a data that you only set or get. A read only attribute defines a single operation the client can call to receives a value. In case of error, an operation is able to throw an exception to the client, attributes cannot raises exception except system exception (du to network fault for instance).

Intuitively, IDL interface correspond to C++ classes and IDL operations correspond to C++ member functions and attributes as a way to read/write public member variable. Nevertheless, IDL defines only the interface to an object and say nothing about the object implementation. IDL is only a descriptive language. Once the interface is fully described in the IDL language, a compiler (from IDL to C++, from IDL to Java...) generates code to implement this interface. Obviously, you still have to write how operations are implemented.

The act of invoking an operation on an interface causes the ORB to send a message to the corresponding object implementation. If the target object is in another address space, the ORB run time sends a remote procedure call to the implementation. If the target object is in the same address space as the caller, the invocation is accomplished as an ordinary function call to avoid the overhead of using a networking protocol.

For an excellent reference on CORBA with C++ refer to [2]. The complete TANGO IDL file can be found in the TANGO web page[3] or at the end of this document in the appendix 2 chapter.

3.2 The model

The basic idea of the TDSOM is to treat each device as an **object**. Each device is a separate entity which has its own data and behavior. Each device has a unique name which identifies it in network name space. Devices are organized according to **classes**, each device belonging to a class. All classes are derived from one root class thus allowing some common behavior for all devices. Four kind of requests can be sent to a device (locally i.e. in the same process, or remotely i.e. across the network) :

- Execute actions via **commands**
- Read/Set data specific to each device belonging to a class via TANGO **attributes**
- Read some basic device data available for all devices via CORBA attributes.
- Execute a predefined set of actions available for every devices via CORBA operations

Each device is stored in a process called a **device server**. Devices are configured at runtime via **properties** which are stored in a **database**.

3.3 The device

The device is the heart of the TDSOM. A device is an abstract concept defined by the TDSOM. In reality, it can be a piece of hardware (an interlock bit) a collection of hardware (a screen attached to a stepper motor) a logical device (a taper) or a combination of all these (an accelerator). Each device has a unique name in the control system and eventually one alias. Within Tango, a four field name space has been adopted consisting of

[//FACILITY/]DOMAIN/CLASS/MEMBER

Facility refers to the control system instance, domain refers to the sub-system, class the class and member the instance of the device. Device name alias(es) must also be unique within a control system. There is no predefined syntax for device name alias.

Each device belongs to a class. The device class contains a complete description and implementation of the behavior of all members of that class. New device classes can be constructed out of existing device classes. This way a new hierarchy of classes can be built up in a short time. Device classes can use existing devices as sub-classes or as sub-objects. The practice of reusing existing classes is classical for Object Oriented Programming and is one of its main advantages.

All device classes are derived from the same class (the device root class) and implement **the same CORBA interface**. All devices implementing the same CORBA interface ensures all control object support the same set of CORBA operations and attributes. The device root class contains part of the common device code. By inheriting from this class, all devices shared a common behavior. This also makes maintenance and improvements to the TDSOM easy to carry out.

All devices also support a **black box** where client requests for attributes or operations are recorded. This feature allows easier debugging session for device already installed in a running control system.

3.3.1 The commands

Each device class implements a list of commands. Commands are very important because they are the client's major dials and knobs for controlling a device. Commands have a fixed calling syntax - consisting of one input argument and one output argument. Arguments type must be chosen in a fixed set of data types: All simple types (boolean, short, long (32 bits), long (64 bits), float, double, unsigned short, unsigned long (32 bits), unsigned long (64 bits) and string) and arrays of simple types plus array of strings and longs and array of strings and doubles). Commands can execute any sequence of actions. Commands can be executed synchronously (the requester is blocked until the command ended) or asynchronously (the requester send the request and is called back when the command ended).

Commands are executed using two CORBA operations named **command_inout** for synchronous commands and **command_inout_async** for asynchronous commands. These two operations called a special method implemented in the device root class - the *command_handler* method. The *command_handler* calls an *is_allowed* method implemented in the device class before calling the command itself. The *is_allowed* method is specific to each command[1]. It checks to see whether the command to be executed is compatible with the present device state. The command function is executed only if the *is_allowed* method allows it. Otherwise, an exception is sent to the client.

3.3.2 The TANGO attributes

In addition to commands, TANGO devices also support normalized data types called attributes[2]. Commands are device specific and the data they transport are not normalized i.e. they can be any one of the TANGO data types with no restriction on what each byte means. This means that it is difficult to interpret the output of a command in terms of what kind of value(s) it represents. Generic display programs need to know what the data returned represents, in what units it is, plus additional information like minimum, maximum, quality etc. Tango attributes solve this problem.

TANGO attributes are zero, one or two dimensional data which have a fix set of properties e.g. quality, minimum and maximum, alarm low and high. They are transferred in a specialized TANGO type and can be read, write or read-write. A device can support a list of attributes. Clients can read one or more attributes from one or more devices. To read TANGO attributes, the client uses the **read_attributes** operation. To write TANGO attributes, a client uses the **write_attributes** operation. To write then read TANGO attributes within the same network request, the client uses the **write_read_attributes** operation. To query a device for all the attributes it supports, a client uses the **get_attribute_config** operation. A client is also able to modify some of parameters defining an attribute with the **set_attribute_config** operation. These four operations are defined in the device CORBA interface.

TANGO support thirteen data types for attributes (and arrays of for one or two dimensional data) which are: boolean, short, long (32 bits), long (64 bits), float, double, unsigned char, unsigned short, unsigned long (32 bits), unsigned long (64 bits), string, a specific data type for Tango device state and finally another specific data type to transfer data as an array of unsigned char with a string describing the coding of these data.

3.3.3 Command or attributes ?

There are no strict rules concerning what should be returned as command result and what should be implemented as an attribute. Nevertheless, attributes are more adapted to return physical value which have a kind of time consistency. Attribute also have more properties which help the client to precisely know what it represents. For instance, the state and the status of a power supply are not physical values and are returned as command result. The current generated by the power supply is a physical value and is implemented as an attribute. The attribute properties allow a client to know its unit, its label and some other informations which are related to a physical value. Command are well adapted to send order to a device like switching from one mode of operation to another mode of operation. For a power supply, the switch from a STANDBY mode to a ON mode is typically done via a command.

3.3.4 The CORBA attributes

Some key data implemented for each device can be read without the need to call a command or read an attribute. These data are :

- The device state
- The device status
- The device name

[1] In contrary to the state_handler method of the TACO device server model which is not specific to each command.

[2] TANGO attributes were known as signals in the TACO device server model

- The administration device name called adm_name
- The device description

The device state is a number representing its state. A set of predefined states are defined in the TDSOM. The device status is a string describing in plain text the device state and any additional useful information of the device as a formatted ascii string. The device name is its name as defined in 3.3. For each set of devices grouped within the same server, an administration device is automatically added. This adm_name is the name of the administration device. The device description is also an ascii string describing the device rule.

These five CORBA attributes are implemented in the device root class and therefore do not need any coding from the device class programmer. As explained in 3.1, the CORBA attributes are not allowed to raise exceptions whereas command (which are implemented using CORBA operations) can.

3.3.5 The remaining CORBA operations

The TDSOM also supports a list of actions defined as CORBA operations in the device interface and implemented in the device root class. Therefore, these actions are implemented automatically for every TANGO device. These operations are :

ping — to ping a device to check if the device is alive. Obviously, it checks only the connection from a client to the device and not all the device functionalities

command_list_query — request a list of all the commands supported by a device with their input and output types and description

command_query — request information about a specific command which are its input and output type and description

info — request general information on the device like its name, the host where the device server hosting the device is running...

black_box — read the device black-box as an array of strings

3.3.6 The special case of the device state and status

Device state and status are the most important key device informations. Nearly all client software dealing with Tango device needs device(s) state and/or status. In order to simplify client software developper work, it is possible to get these two piece of information in three different manners :

1. Using the appropriate CORBA attribute (state or status)
2. Using command on the device. The command are called State or Status
3. Using attribute. Even if the state and status are not real attribute, it is possible to get their value using the read_attributes operation. Nevertheless, it is not possible to set the attribute configuration for state and status. An error is reported by the server if a client try to do so.

3.3.7 The device polling

Within the Tango framework, it is also possible to force executing command(s) or reading attribute(s) at a fixed frequency. It is called *device polling*. This is automatically handled by Tango core software with a polling threads pool. The command result or attribute value are stored in circular buffers. When a client want to read attribute value (or command result) for a polled attribute (or a polled command), he has the choice to get the attribute value (or command result) with a real access to the device of from the last value stored in the device ring buffer. This is a great advantage for “slow” devices. Getting data from the buffer is much faster than accessing the device itself. The technical disadvantage is the time shift between the data returned from the polling buffer and the time of the request. Polling a command is only possible for command without input arguments.

Two other CORBA operations called *command_inout_history_X* and *read_attribute _history_X* allow a client to retrieve the history of polled command or attribute stored in the polling buffers. Obviously, this history is limited to the depth of the polling buffer.

The whole polling system is available only since Tango release 2.x and above in CPP and since TangORB release 3.7.x and above in Java.

3.4 The server

Another integral part of the TDSOM is the server concept. The server (also referred as device server) is a process whose main task is to offer one or more services to one or more clients. To do this, the server has to spend most of its time in a wait loop waiting for clients to connect to it. The devices are hosted in the server process. A server is able to host several classes of devices. In the TDSOM, a device of the **DServer** class is automatically hosted by each device server. This class of device supports commands which enable remote device server process administration.

TANGO supports device server process on two families of operating system : Linux and Windows.

3.5 The Tango Logging Service

During software life, it is always convenient to print miscellaneous informations which help to:

- Debug the software
- Report on error
- Give regular information to user

This is classically done using cout (or C printf) in C++ or println method in Java language. In a highly distributed control system, it is difficult to get all these informations coming from a high number of different processes running on a large number of computers. Since its release 3, Tango has incorporated a Logging Service called the Tango Logging Service (TLS) which allows print messages to be:

- Displayed on a console (the classical way)
- Sent to a file
- Sent to specific Tango device called log consumer. Tango package has an implementation of log consumer where every consumer device is associated to a graphical interface. This graphical interface display messages but could also be used to sort messages, to filter messages... Using this feature, it is possible to centralise display of these messages coming from different devices embedded within different processes. These log consumers can be:
  - Statically configured meaning that it memorizes the list of Tango devices for which it will get and display messages.
  - Dynamically configured. The user, with the help of the graphical interface, chooses devices from which he want to see messages.

3.6 The database

To achieve complete device independence, it is necessary however to supplement device classes with a possibility for configuring device dependencies at runtime. The utility which does this in the TDSOM is the **property database**. Properties[3] are identified by an ascii string and the device name. TANGO attributes

[3]Properties were known as resources in the TACO device server model

are also configured using properties. This database is also used to store device network addresses (CORBA IOR's), list of classes hosted by a device server process and list of devices for each class in a device server process. The database ensure the uniqueness of device name and of alias. It also links device name and it list of aliases.

TANGO uses MySQL[6] as its database. MySQL is a relational database which implements the SQL language. However, this is largely enough to implement all the functionalities needed by the TDSOM. The database is accessed via a classical TANGO device hosted in a device server. Therefore, client access the database via TANGO commands requested on the database device. For a good reference on MySQL refer to [7]

3.7 The controlled access

Tango also provides a controlled access system. It's a simple controlled access system. It does not provide encrypted communication or sophisticated authentification. It simply defines which user (based on computer loggin authentification) is allowed to do which command (or write attribute) on which device and from which host. The information used to configure this controlled access feature are stored in the Tango database and accessed by a specific Tango device server which is not the classsical Tango database device server described in the previous section. Two access levels are defined:

- Everything is allowed for this user from this host
- The write-like calls on the device are forbidden and according to configuration, a command subset is also forbidden for this user from this host

This feature is precisely described in the chapter "Advanced features"

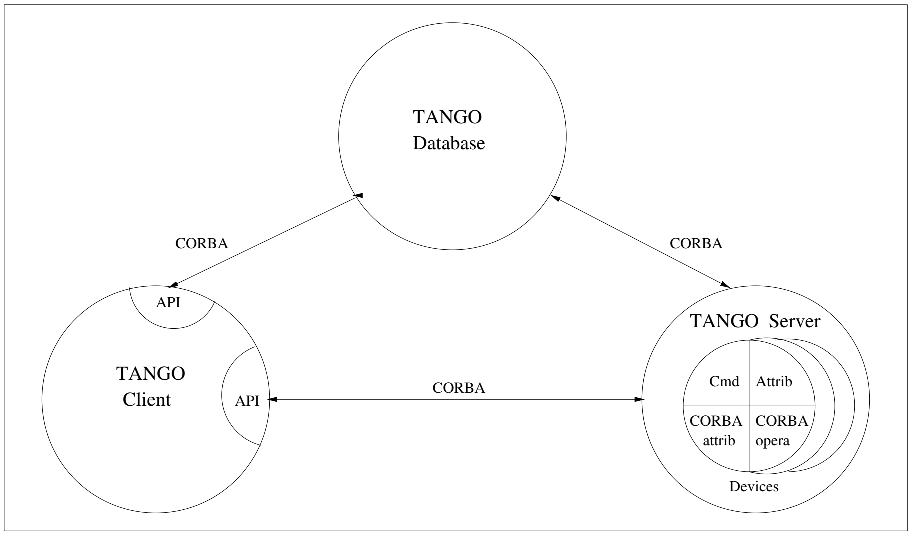
3.8 The Application Programmers Interfaces

3.8.1 Rules of the API

While it is true TANGO clients can be programmed using only the CORBA API, CORBA knows nothing about TANGO. This means client have to know all the details of retrieving IORs from the TANGO database, additional information to send on the wire, TANGO version control etc. These details can and should be wrapped in TANGO Application Programmer Interface (API). The API is implemented as a library in C++ and as a package in Java. The API is what makes TANGO clients easy to write. The API's consists the following basic classes :

- DeviceProxy which is a *proxy* to the real device
- DeviceData to encapsulate data send/receive from/to device via commands
- DeviceAttribute to encapsulate data send/receive from/to device via attributes
- Group which is a *proxy* to a group of devices

In addition to these main classes, many other classes allows a full interface to TANGO features. The following figure is a drawing of a typical client/server application using TANGO.



The database is used during server and client startup phase to establish connection between client and server.

3.8.2 Communication between client and server using the API

With the API, it is possible to request command to be executed on a device or to read/write device attribute(s) using one of the two communication models implemented. These two models are:

1. The synchronous model where client waits (and is blocked) for the server to send the answer or until the timeout is reached

2. The asynchronous model. In this model, the clients send the request and immediately returns. It is not blocked. It is free to do whatever it has to do like updating a graphical user interface. The client has the choice to retrieve the server answer by checking if the reply is arrived by calling an API specific call or by requesting that a call-back method is executed when the client receives the server answer.

The asynchronous model is available with Tango release 3 and above.

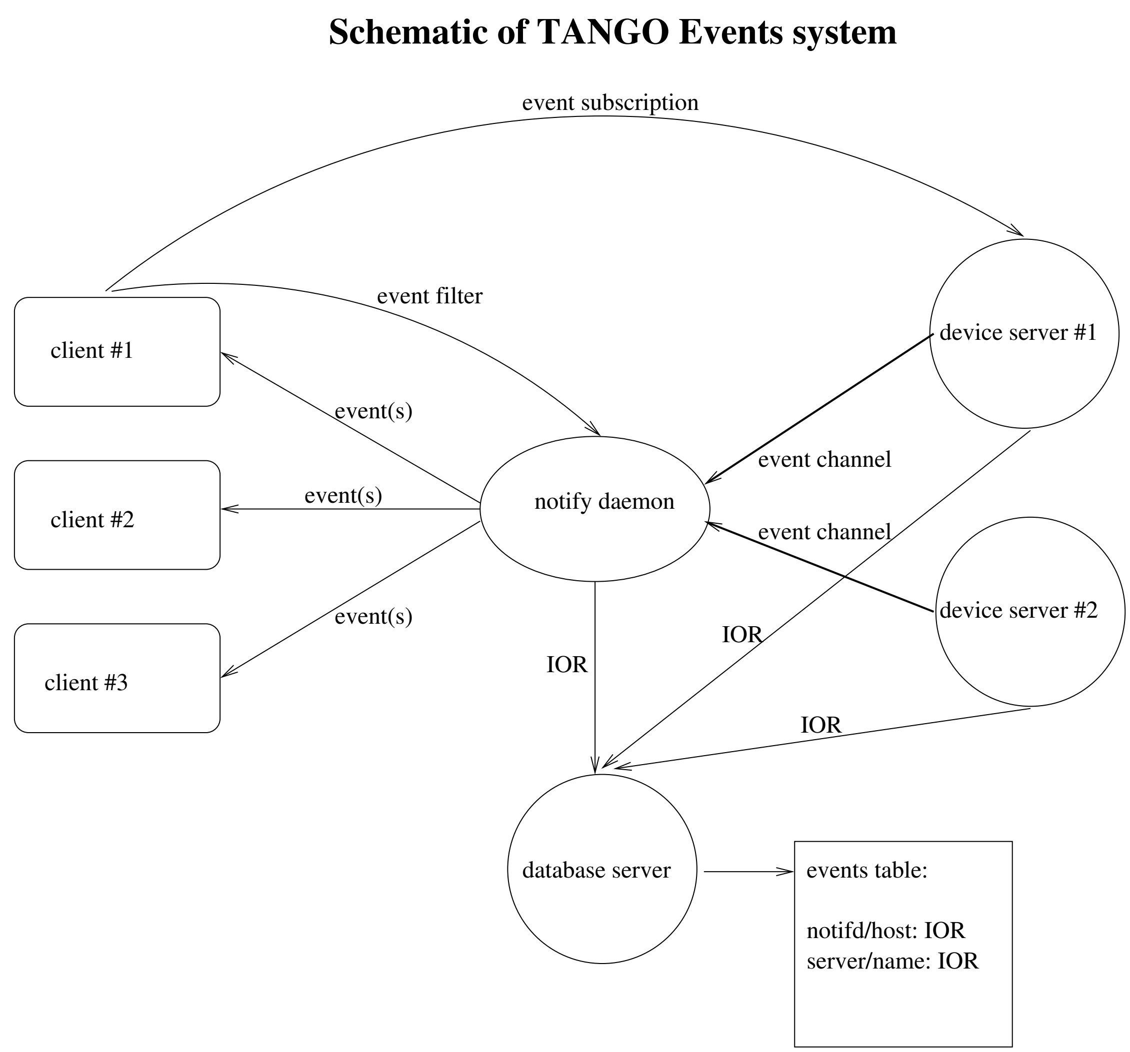
3.8.3 Tango events

On top of the two communication model previously described, TANGO offers an "event system". The standard TANGO communication paradigm is a synchronou/asynchronous two-way call. In this paradigm the call is initiated by the client who contacts the server. The server handles the client's request and sends the answer to the client or throws an exception which the client catches. This paradigm involves two calls to receive a single answer and requires the client to be active in initiating the request. If the client has a permanent interest in a value he is obliged to poll the server for an update in a value every time. This is not efficient in terms of network bandwidth nor in terms of client programming.

For clients who are permanently interested in values the event-driven communication paradigm is a more efficient and natural way of programming. In this paradigm the client registers his interest once in an event (value). After that the server informs the client every time the event has occurred. This paradigm avoids the client polling, frees it for doing other things, is fast and makes efficient use of the network.

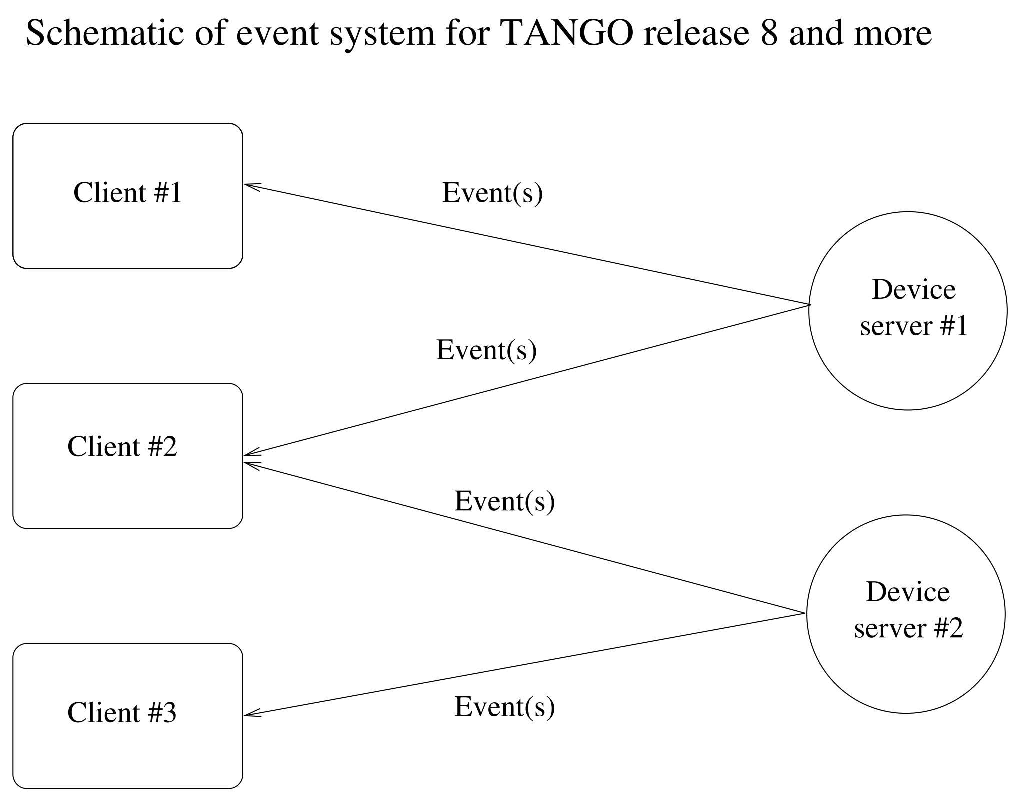
Before TANGO release 8, TANGO used the CORBA OMG COS Notification Service to generates events. TANGO uses the omniNotify implementation of the Notification service. omniNotify was developed in conjunction with the omniORB CORBA implementation also used by TANGO. The heart of the Notification Service is the notification daemon. The omniNotify daemons are the processes which receive events from device servers and distribute them to all clients which are subscribed. In order to distribute the load of the events there is one notification daemon per host. Servers send their events to the daemon on the local host. Clients and servers get the IOR for the host from the TANGO database.

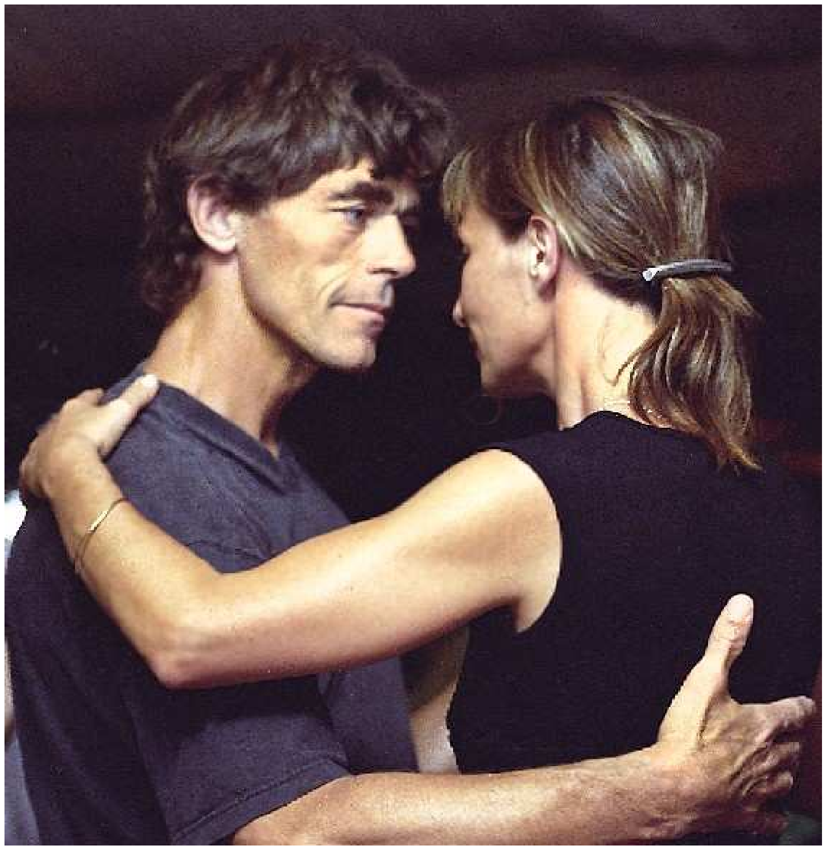
The following figure is a schematic of the Tango event system for Tango releases before Tango 8.



Starting with Tango 8, a new design of the event system has been implemented. This new design is based on the ZMQ library. ZMQ is a library allowing users to create communicating system. It implements several well known communication pattern including the Publish/Subscribe pattern which is the basic of the new Tango event system. Using this library, a separate notification service is not needed anymore and event communiction is available with only client and server processes which simplifies the overall design.

The following figure is a schematic of the Tango event system for Tango releases starting with Tango release 8.


Schematic of event system for TANGO release 8 and more
Client #1
Client #2
Client #3
Event(s)
Event(s)
Event(s)
Event(s)
Device
server #1
Device
server #2



Chapter 4

Writing a TANGO client using TANGO APIs

4.1 Introduction

TANGO devices and database are implemented using the TANGO device server model. To access them the user has the CORBA interface e.g. command_inout(), write_attributes() etc. defined by the idl file. These methods are very low-level and assume a good working knowledge of CORBA. In order to simplify this access, high-level api in C++ and Java have been implemented which hides all CORBA aspects of TANGO. In addition the api hides details like how to connect to a device via the database, how to reconnect after a device has been restarted, how to correctly pack and unpack attributes and so on by implementing these in a manner transparent to the user. The api provides a unified error handling for all TANGO and CORBA errors. Unlike the CORBA C++ bindings the TANGO api supports native C++ data types e.g. strings and vectors.

This chapter describes how to use these API's. It is not a reference guide. See chapter 6 for the C++ API details or chapter 5 for a Java API reference guide.

4.2 Getting Started

Refer to the chapter "Getting Started" for an example on getting start with the C++ or Java api.

4.3 Basic Philosophy

The basic philosophy is to have high level classes to deal with Tango devices. To communicate with Tango device, uses the **DeviceProxy** class. To send/receive data to/from Tango device, uses the **DeviceData** or **DeviceAttribute** classes. To communicate with a group of devices, use the **Group** class. If you are interested only in some attributes provided by a Tango device, uses the **AttributeProxy** class. Even if the Tango database is implemented as any other devices (and therefore accessible with one instance of a DeviceProxy class), specific high level classes have been developped to query it. Uses the **Database**, **DbDevice**, **DbClass**, **DbServer** or **DbData** classes when interfacing the Tango database. Callback for asynchronous requests or events are implemented via a **CallBack** class. An utility class called **ApiUtil** is also available.

4.4 Data types

The definition of the basic data type you can transfert using Tango is:

Tango type name	C++ equivalent type
DevBoolean	boolean
DevShort	short
DevLong	int (always 32 bits data)
DevLong64	long long on 32 bits chip or long on 64 bits chip always 64 bits data
DevFloat	float
DevDouble	double
DevString	char *
DevEncoded	structure with 2 fields: a string and an array of unsigned char
DevUChar	unsigned char
DevUShort	unsigned short
DevULong	unsigned int (always 32 bits data)
DevULong64	unsigned long long on 32 bits chip or unsigned long on 64 bits chip always 64 bits data
DevState	Tango specific data type

Using commands, you are able to transfert all these data types, array of these basic types and two other Tango specific data types called DevVarLongStringArray and DevVarDoubleStringArray. See chapter 8.2 to get details about them. You are also able to create attributes using any of these basic data types to transfer data between clients and servers.

4.5 Request model

For the most important API remote calls (command_inout, read_attribute(s) and write_attribute(s)), Tango supports two kind of requests which are the synchronous model and the asynchronous model. Synchronous model means that the client wait (and is blocked) for the server to send an answer. Asynchronous model means that the client does not wait for the server to send an answer. The client sends the request and immediately returns allowing the CPU to do anything else (like updating a graphical user interface). Within Tango, there are two ways to retrieve the server answer when using asynchronous model. They are:

1. The polling mode
2. The callback mode

In polling mode, the client executes a specific call to check if the answer is arrived. If this is not the case, an exception is thrown. If the reply is there, it is returned to the caller and if the reply was an exception, it is re-thrown. There are two calls to check if the reply is arrived:

- Call which does not wait before the server answer is returned to the caller.
- Call which wait with timeout before returning the server answer to the caller (or throw the exception) if the answer is not arrived.

In callback model, the caller must supply a callback method which will be executed when the command returns. They are two sub-modes:

1. The pull callback mode
2. The push callback mode

In the pull callback mode, the callback is triggered if the server answer is arrived when the client decide it by calling a *synchronization* method (The client pull-out the answer). In push mode, the callback is executed as soon as the reply arrives in a separate thread (The server pushes the answer to the client).

4.5.1 Synchronous model

Synchronous access to Tango device are provided using the *DeviceProxy* or *AttributeProxy* class. For the *DeviceProxy* class, the main synchronous call methods are :

- *command_inout()* to execute a Tango device command
- *read_attribute()* or *read_attributes()* to read a Tango device attribute(s)
- *write_attribute()* or *write_attributes()* to write a Tango device attribute
- *write_read_attribute()* to write then read a Tango device attribute

For commands, data are send/received to/from device using the *DeviceData* class. For attributes, data are send/received to/from device attribute using the *DeviceAttribute* class.

In some cases, only attributes provided by a Tango device are interesting for the application. You can use the *AttributeProxy* class. Its main synchronous methods are :

- *read()* to read the attribute value
- *write()* to write the attribute value
- *write_read()* to write then read the attribute value

Data are transmitted using the *DeviceAttribute* class.

4.5.2 Asynchronous model

Asynchronous access to Tango device are provided using *DeviceProxy* or *AttributeProxy, CallBack* and *ApiUtil* classes methods. The main asynchronous call methods and used classes are :

- To execute a command on a device
  - *DeviceProxy::command_inout_asynch()* and *DeviceProxy::command_inout_reply()* in polling model.
  - *DeviceProxy::command_inout_asynch()*, *DeviceProxy::get_asynch_replies()* and *CallBack* class in callback pull model
  - *DeviceProxy::command_inout_asynch()*, *ApiUtil::set_asynch_cb_sub_model()* and *CallBack* class in callback push model
- To read a device attribute
  - *DeviceProxy::read_attribute_asynch()* and *DeviceProxy::read_attribute_reply()* in polling model
  - *DeviceProxy::read_attribute_asynch()*, *DeviceProxy::get_asynch_replies()* and *CallBack* class in callback pull model.
  - *DeviceProxy::read_attribute_asynch()*, *ApiUtil::set_asynch_cb_sub_model()* and *CallBack* class in callback push model
- To write a device attribute
  - *DeviceProxy::write_attribute_asynch()* in polling model
  - *DeviceProxy::write_attribute_asynch()* and *CallBack* class in callback pull model
  - *DeviceProxy::write_attribute_asynch()*, *ApiUtil::set_asynch_cb_sub_model()* and *CallBack* class in callback push model

For commands, data are send/received to/from device using the *DeviceData* class. For attributes, data are send/received to/from device attribute using the *DeviceAttribute* class. It is also possible to generate asynchronous request(s) using the *AttributeProxy* class following the same schema than above. Methods to use are :

- *read_asynch()* and *read_reply()* to asynchronously read the attribute value
- *write_asynch()* and *write_reply()* to asynchronously write the attribute value

4.6 Events

4.6.1 Introduction

Events are a critical part of any distributed control system. Their aim is to provide a communication mechanism which is fast and efficient.

The standard CORBA communication paradigm is a synchronous or asynchronous two-way call. In this paradigm the call is initiated by the client who contacts the server. The server handles the client's request and sends the answer to the client or throws an exception which the client catches. This paradigm involves two calls to receive a single answer and requires the client to be active in initiating the request. If the client has a permanent interest in a value he is obliged to poll the server for an update in a value every time. This is not efficient in terms of network bandwidth nor in terms of client programming.

For clients who are permanently interested in values the event-driven communication paradigm is a more efficient and natural way of programming. In this paradigm the client registers her interest once in an event (value). After that the server informs the client every time the event has occurred. This paradigm avoids the client polling, frees it for doing other things, is fast and makes efficient use of the network.

The rest of this chapter explains how the TANGO events are implemented and the application programmer's interface.

4.6.2 Event definition

TANGO events represent an alternative channel for reading TANGO device attributes. Device attributes values are sent to all subscribed clients when an event occurs. Events can be an attribute value change, a change in the data quality or a periodically send event. The clients continue receiving events as long as they stay subscribed. Most of the time, the device server polling thread detects the event and then pushes the device attribute value to all clients. Nevertheless, in some cases, the delay introduced by the polling thread in the event propagation is detrimental. For such cases, some API calls directly push the event. Until TANGO release 8, the omniNotify implementation of the CORBA Notification service was used to dispatch events. Starting with TANGO 8, this CORBA Notification service has been replaced by the ZMQ library which implements a Publish/Subscribe communication model well adapted to TANGO events communication.

4.6.3 Event types

The following five event types have been implemented in TANGO :

1. **change** - an event is triggered and the attribute value is sent when the attribute value changes significantly. The exact meaning of significant is device attribute dependent. For analog and digital values this is a delta fixed per attribute, for string values this is any non-zero change i.e. if the new attribute value is not equal to the previous attribute value. The delta can either be specified as a relative or absolute change. The delta is the same for all clients unless a filter is specified (see below). To easily write applications using the change event, it is also triggered in the following case :

   (a) When a spectrum or image attribute size changes.

   (b) At event subscription time

   (c) When the polling thread receives an exception during attribute reading
   (d) When the polling thread detects that the attribute quality factor has changed.
   (e) The first good reading of the attribute after the polling thread has received exception when trying to read the attribute
   (f) The first time the polling thread detects that the attribute quality factor has changed from INVALID to something else
   (g) When a change event is pushed manually from the device server code. (*DeviceImpl::push_change_event()*).
   (h) By the methods Attribute::set_quality() and Attribute::set_value_date_quality() if a client has subscribed to the change event on the attribute. This has been implemented for cases where the delay introduced by the polling thread in the event propagation is not authorized.

2. **periodic** - an event is sent at a fixed periodic interval. The frequency of this event is determined by the *event_period* property of the attribute and the polling frequency. The polling frequency determines the highest frequency at which the attribute is read. The event_period determines the highest frequency at which the periodic event is sent. Note if the event_period is not an integral number of the polling period there will be a beating of the two frequencies[1]. Clients can reduce the frequency at which they receive periodic events by specifying a filter on the periodic event counter.

3. **archive** - an event is sent if one of the archiving conditions is satisfied. Archiving conditions are defined via properties in the database. These can be a mixture of delta_change and periodic. Archive events can be send from the polling thread or can be manually pushed from the device server code (*DeviceImpl::push_archive_event()*).

4. **attribute configuration** - an event is sent if the attribute configuration is changed.

5. **data ready** - This event is sent when coded by the device server programmer who uses a specific method of one of the Tango device server class to fire the event (*DeviceImpl::push_data_ready_event()*). The rule of this event is to inform a client that it is now possible to read an attribute. This could be useful in case of attribute with many data.

6. **user** - The criteria and configuration of these user events are managed by the device server programmer who uses a specific method of one of the Tango device server class to fire the event (*DeviceImpl::push_event()*).

The first three above events are automatically generated by the TANGO library or fired by the user code. Even number 4 is only automatically sent by the library and the last two are fired only by the user code.

4.6.4 Event filtering (Removed in Tango release 8 and above)

Please, note that this feature is available only for Tango releases older than Tango 8. The CORBA Notification Service allows event filtering. This means that a client can ask the Notification Service to send the event only if some filter is evaluated to true. Within the Tango control system, some pre-defined fields can be used as filter. These fields depend on the event type.

Event type	Filterable field name	Filterable field value	type
change	delta_change_rel	Relative change (in %) since last event	double
	delta_change_abs	Absolute change since last event	double
	quality	Is set to 1 when the attribute quality factor has changed, otherwise it is 0	double
	forced_event	Is set to 1 when the event was fired on exception or a quality factor set to invalid	double

[1] note: the polling is not synchronized is currently not synchronized on the hour

periodic	counter	Incremented each time the event is sent	long
archive	delta_change_rel	Relative change (in %) since last event	double
	delta_change_abs	Absolute change since last event	double
	quality	Is set to 1 when the attribute quality factor has changed, otherwise it is 0	double
	counter	Incremented each time the event is sent for periodic reason. Set to -1 if event sent for change reason	long
	forced_event	Is set to 1 when the event was fired on exception or a quality factor set to invalid	double
	delta_event	Number of milli-seconds since previous event	double

Filter are defined as a string following a grammar defined by CORBA. It is defined in [18]. The following example shows you the most common use of these filters in the Tango world :

- To receive periodic event one out of every three, the filter must be

"$counter % 3 == 0"

- To receive change event only if the relative change is greater than 20 % (positive and negative), the filter must be

"$delta_change_rel >= 20 or $delta_change_rel <= -20"

- To receive a change event only on quality change, the filter must be

"$quality == 1"

For user events, the filter field name(s) and their value are defined by the device server programmer.

4.6.5 Application Programmer's Interface

How to setup and use the TANGO events ? The interfaces described here are intended as user friendly interfaces to the underlying CORBA calls. The interface is modeled after the asynchronous *command_inout()* interface so as to maintain coherency. The event system supports **push callback model** as well as the **pull callback model.**

The two event reception modes are:

- **Push callback model** : On event reception a callbacks method gets immediately executed.
- **Pull callback model** : The event will be buffered the client until the client is ready to receive the event data. The client triggers the execution of the callback method.

The event reception buffer in the **pull callback model**, is implemented as a round robin buffer. The client can choose the size when subscribing for the event. This way the client can set-up different ways to receive events.

- Event reception buffer size = 1 : The client is interested only in the value of the last event received. All other events that have been received since the last reading are discarded.
- Event reception buffer size > 1 : The client has chosen to keep an event history of a given size. When more events arrive since the last reading, older events will be discarded.
- Event reception buffer size = ALL_EVENTS : The client buffers all received events. The buffer size is unlimited and only restricted by the available memory for the client.

4.6.5.1 Configuring events

The attribute configuration set is used to configure under what conditions events are generated. A set of standard attribute properties (part of the standard attribute configuration) are read from the database at device startup time and used to configure the event engine. If there are no properties defined then default values specified in the code are used.

4.6.5.1.1 change The attribute properties and their default values for the "change" event are :

1. **rel_change** - a property of maximum 2 values. It specifies the positive and negative relative change of the attribute value w.r.t. the value of the previous change event which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no property is specified, no events are generated.

2. **abs_change** - a property of maximum 2 values.It specifies the positive and negative absolute change of the attribute value w.r.t the value of the previous change event which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then the relative change is used.

4.6.5.1.2 periodic The attribute properties and their default values for the "periodic" event are :

1. **event_period** - the minimum time between events (in milliseconds). If no property is specified then a default value of 1 second is used.

4.6.5.1.3 archive The attribute properties and their default values for the "archive" event are :

1. **archive_rel_change** - a property of maximum 2 values which specifies the positive and negative relative change w.r.t. the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then no events are generate.

2. **archive_abs_change** - a property of maximum 2 values which specifies the positive and negative absolute change w.r.t the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then the relative change is used.

3. **archive_period** - the minimum time between archive events (in milliseconds). If no property is specified, no periodic archiving events are send.

4.6.5.2 C++ Clients

This is the interface for clients who want to receive events. The main action of the client is to subscribe and unsubscribe to events. Once the client has subscribed to one or more events the events are received in a separate thread by the client.

Two reception modes are possible:

- On event reception a callbacks method gets immediately executed.
- The event will be buffered until the client until the client is ready to receive the event data.

The mode to be used has to be chosen when subscribing for the event.

4.6.5.2.1 Subscribing to events The client call to subscribe to an event is named *DeviceProxy::subscribe_event()* . During the event subscription the client has to choose the event reception mode to use.

Push model:

```
int DeviceProxy::subscribe_event(
             const string &attribute,
             Tango::EventType event,
             Tango::CallBack *callback,
             bool stateless = false);
```

The client implements a callback method which is triggered when the event is received. Note that this callback method will be executed by a thread started by the underlying ORB. This thread is not the application main thread. For Tango releases before 8, a similar call with one extra parameter for event filtering is also available.

Pull model:

```
int DeviceProxy::subscribe_event(
             const string &attribute,
             Tango::EventType event,
             int event_queue_size,
             bool stateless = false);
```

The client chooses the size of the round robin event reception buffer. Arriving events will be buffered until the client uses *DeviceProxy::get_events()* to extract the event data. For Tango releases before 8, a similar call with one extra parameter for event filtering is also available.

On top of the user filter defined by the *filters* parameter, basic filtering is done based on the reason specified and the event type. For example when reading the state and the reason specified is "change" the event will be fired only when the state changes. Events consist of an attribute name and the event reason. A standard set of reasons are implemented by the system, additional device specific reasons can be implemented by device servers programmers.

The stateless flag = false indicates that the event subscription will only succeed when the given attribute is known and available in the Tango system. Setting stateless = true will make the subscription succeed, even if an attribute of this name was never known. The real event subscription will happen when the given attribute will be available in the Tango system.

Note that in this model, the callback method will be executed by the thread doing the *DeviceProxy::get_events()* call.

4.6.5.2.2 The CallBack class In C++, the client has to implement a class inheriting from the Tango CallBack class and pass this to the *DeviceProxy::subscribe_event()* method. The CallBack class is the same class as the one proposed for the TANGO asynchronous call. This is as follows for events :

```
class MyCallback : public Tango::CallBack
{
   .
   .
   .
   virtual push_event(Tango::EventData *);
   virtual push_event(Tango::AttrConfEventData *);
   virtual push_event(Tango::DataReadyEventData *);
}
```

where EventData is defined as follows :

```
class EventData
{
```

```
    DeviceProxy *device;
    string &attr_name;
    string &event;
    DeviceAttribute *attr_value;
    bool err;
    DevErrorList &errors;
}
```

AttrConfEventData is defined as follows :

```
class AttrConfEventData
{
    DeviceProxy *device;
    string &attr_name;
    string &event;
    AttributeInfoEx*attr_conf;
    bool err;
    DevErrorList &errors;
}
```

and DataReadyEventData is defined as follows :

```
class DataReadyEventData
{
    DeviceProxy *device;
    string &attr_name;
    string &event;
    int attr_data_type;
    int ctr;
    bool err;
    DevErrorList &errors;
}
```

In push model, there are some cases (same callback used for events coming from different devices hosted in device server process running on different hosts) where the callback method could be executed concurently by different threads started by the ORB. The user has to code his callback method in a **thread safe** manner.

4.6.5.2.3 Unsubscribing from an event Unsubscribe a client from receiving the event specified by *event_id* is done by calling the *DeviceProxy::unsubscribe_event()* method :

```
void DeviceProxy::unsubscribe_event(int event_id);
```

4.6.5.2.4 Extract buffered event data When the pull model was chosen during the event subscription, the received event data can be extracted with *DeviceProxy::get_events()*. Two possibilities are available for data extraction. Either a callback method can be executed for every event in the buffer when using

```
int DeviceProxy::get_events(
             int event_id,
             CallBack *cb);
```

Or all the event data can be directly extracted as EventDataList, AttrConfEventDataList or DataReadyEventDataList when using

```
int DeviceProxy::get_events(
             int event_id,
```

```
                 EventDataList &event_list);
    int DeviceProxy::get_events(
                 int event_id,
                 AttrConfEventDataList &event_list);
    int DeviceProxy::get_events(
                 int event_id,
                 DataReadyEventDataList &event_list);
```

The event data lists are vectors of EventData, AttrConfEventData or DataReadyEventData pointers with special destructor and clean-up methods to ease the memory handling.

```
    class EventDataList:public vector<EventData *>
    class AttrConfEventDataList:public vector<AttrConfEventData *>
    class DataReadyEventDataList:public vector<DataReadyEventDataList *>
```

4.6.5.2.5 Example Here is a typical code example of a client to register and receive events. First, you have to define a callback method as follows:

```
    class DoubleEventCallBack : public Tango::CallBack
    {
       void push_event(Tango::EventData*);
    };

    void DoubleEventCallBack::push_event(Tango::EventData *myevent)
    {
        Tango::DevVarDoubleArray *double_value;
        try
        {
            cout << "DoubleEventCallBack::push_event(): called attribute "
                 << myevent->attr_name
                 << " event "
                 << myevent->event
                 << " (err="
                 << myevent->err
                 << ")" << endl;

             if (!myevent->err)
             {
                 myevent->attr_value >> double_value;
                 cout << "double value "
                      << (*double_value)[0]
                      << endl;
                 delete double_value;
             }
        }
        catch (...)
        {
             cout << "DoubleEventCallBack::push_event(): could not extract data !\n";
        }
    }
```

Then the main code must subscribe to the event and choose the push or the pull model for event reception.

Push model:

```
DoubleEventCallBack *double_callback = new DoubleEventCallBack;

Tango::DeviceProxy *mydevice = new Tango::DeviceProxy("my/device/1");

int event_id;
const string attr_name("current");
event_id = mydevice->subscribe_event(attr_name,
                         Tango::CHANGE_EVENT,
                         double_callback);
cout << "event_client() id = " << event_id << endl;
// The callback methods are executed by the Tango event reception thread.
// The main thread is not concerned of event reception.
// Whatch out with synchronisation and data access in a multi threaded environment!
sleep(1000); // wait for events

event_test->unsubscribe_event(event_id);
```

Pull model:

```
DoubleEventCallBack *double_callback = new DoubleEventCallBack;
int event_queue_size = 100; // keep the last 100 events

Tango::DeviceProxy *mydevice = new Tango::DeviceProxy("my/device/1");

int event_id;
const string attr_name("current");
event_id = mydevice->subscribe_event(attr_name,
                         Tango::CHANGE_EVENT,
                         event_queue_size);
cout << "event_client() id = " << event_id << endl;
// Check every 3 seconds whether new events have arrived and trigger the callback method
// for the new events.
for (int i=0; i < 100; i++)
{
    sleep (3);

    // Read the stored event data from the queue and call the callback method for every event.
    mydevice->get_events(event_id, double_callback);
}

event_test->unsubscribe_event(event_id);
```

4.6.5.3 Java Clients

This is the interface for java clients who want to receive events. There are two ways to receive events using the TANGO java API :

1. Using Callback.
2. Using Java listener

Using callback, is very similar to a C++ clients. Using listener is more in the Java philosophy.

4.6.5.3.1 Using CallBack In Java when using callback, the client has to implement a class inheriting from the Tango CallBack class and pass this to the *DeviceProxy.subscribe_event()* method. The CallBack class is the same class as the one proposed for the TANGO asynchronous call. This is as follows for events :

```
class MyCallback extends CallBack
{
   .
   .
   .
   public void push_event(EventData evt)
   {

   }
}
```

where EventData is similar to the C++ EventData class. To subscribe to an event, use the *DeviceProxy.subscribe_event()* method. To unsubscribe from an event, use the *DeviceProxy.unsubscribe_event()* method.

4.6.5.3.2 Using listeners The Tango API defined four Java interfaces called

- *ITangoChangeListener* for the change event
- *ITangoPeriodicListener* for the periodic event
- *ITangoQualityChangeListener* for the quality change event
- *ITangoArchiveListener* for the archive event

All these interfaces defined one method respectively called *change()*, *periodic()*, *qualityChange()* and *archive()* which will be called when the event is received. The user must write a class implementing the interface for which he (she) want to receive event.

To install or remove a listener, use the *TangoEventsAdapter* class which has methods to install/remove listeners for the four different types of listener. This TangoEventAdapter class is created from the Tango device name.

4.6.5.3.2.1 Example Here is a typical example of what a client will need to do to register for and receive events. First, you have to define a class implementing an interface as follows:

```
class DoubleEventListener implements ITangoPeriodicListener
{
   public void periodic(TangoPeriodicEvent event)
   {
      DeviceAttribute attr = event.getValue();
      double[] double_value;
      try
      {
         double_value =   attr.extractDoubleArray();
         System.out.println(" double value " + double_value[0]);
      }
      catch (Exception e)
      {
         System.out.println(
"DoubleEventListener.periodic() : could not extract data!");
      }
   }
}
```

The main code looks like (suppose the device generating event is called *my/event/tester* and the attribute name is *double_event*) :

```
DoubleEventListener listener = new DoubleEventListener();

TangoEventsAdapter adapter = new TangoEventsAdapter("my/event/tester") ;

String[] filters = new String[0];
adapter.addTangoPeriodicListener(listener,"double_event",filters);
```

4.7 Group

A Tango Group provides the user with a single point of control for a collection of devices. By analogy, one could see a Tango Group as a proxy for a collection of devices. For instance, the Tango Group API supplies a *command_inout()* method to execute the same command on all the elements of a group.

A Tango Group is also a hierarchical object. In other words, it is possible to build a group of both groups and individual devices. This feature allows creating logical views of the control system - each view representing a hierarchical family of devices or a sub-system.

In this chapter, we will use the term *hierarchy* to refer to a group and its sub-groups. The term *Group* designates to the local set of devices attached to a specific Group.

4.7.1 Getting started with Tango group

The quickest way of getting started is to study an example…

Imagine we are vacuum engineers who need to monitor and control hundreds of gauges distributed over the 16 cells of a large-scale instrument. Each cell contains several penning and pirani gauges. It also contains one "strange" gauge. Our main requirement is to be able to control the whole set of gauges, a family of gauges located into a particular cell (e.g. all the penning gauges of the 6th cell) or a single gauge (e.g. the strange gauge of the 7th cell). Using a Tango Group, such features are quite straightforward to obtain.

Reading the description of the problem, the device hierarchy becomes obvious. Our "gauges" group will have the following structure:

```
-> gauges
  |  -> cell-01
  |      |-> inst-c01/vac-gauge/strange
  |      |-> penning
  |      |    |-> inst-c01/vac-gauge/penning-01
  |      |    |-> inst-c01/vac-gauge/penning-02
  |      |    |- ...
  |      |    |-> inst-c01/vac-gauge/penning-xx
  |      |-> pirani
  |           |-> inst-c01/vac-gauge/pirani-01
  |           |-> ...
  |           |-> inst-c01/vac-gauge/pirani-xx
  |  -> cell-02
  |      |-> inst-c02/vac-gauge/strange
  |      |-> penning
  |      |    |-> inst-c02/vac-gauge/penning-01
  |      |    |-> ...
  |      |
  |      |-> pirani
  |      |    |-> ...
  |  -> cell-03
  |      |-> ...
  |           | -> ...
```

In the C++, such a hierarchy can be build as follows (basic version):

```
//- step0: create the root group
Tango::Group *gauges = new Tango::Group("gauges");

//- step1: create a group for the n-th cell
Tango::Group *cell = new Tango::Group("cell-01");

//- step2: make the cell a sub-group of the root group
gauges->add(cell);

//- step3: create a "penning" group
Tango::Group *gauge_family = new Tango::Group("penning");

//- step4: add all penning gauges located into the cell (note the wildcard)
gauge_family->add("inst-c01/vac-gauge/penning*");

//- step5: add the penning gauges to the cell
cell->add(gauge_family);

//- step6: create a "pirani" group
gauge_family = new Tango::Group("pirani");
```

```
//- step7: add all pirani gauges located into the cell (note the wildcard)
gauge_family->add("inst-c01/vac-gauge/pirani*");

//- step8: add the pirani gauges to the cell
cell->add(gauge_family);

//- step9: add the "strange" gauge to the cell
cell->add("inst-c01/vac-gauge/strange");

//- repeat step 1 to 9 for the remaining cells
cell = new Tango::Group("cell-02");
...
```

Here is the Java version:

```
import fr.esrf.TangoApi.Group;

//- step0: create the root group Group
gauges = new Group("gauges");

//- step1: create a group for the n-th cell
Group cell = new Group("cell-01");

//- step2: make the cell a sub-group of the root group
gauges.add(cell);

//- step3: create a "penning" group
Group gauge_family = new Group("penning");

//- step4: add all penning gauges located into the cell (note the wildcard)
gauge_family.add("inst-c01/vac-gauge/penning*");

//- step5: add the penning gauges to the cell
cell.add(gauge_family);

//- step6: create a "pirani" group
gauge_family = new Group("pirani");

//- step7: add all pirani gauges located into the cell (note the wildcard)
gauge_family.add("inst-c01/vac-gauge/pirani*");

//- step8: add the pirani gauges to the cell cell.add(gauge_family);

//- step9: add the "strange" gauge to the cell
cell.add("inst-c01/vac-gauge/strange");

//- repeat step 1 to 9 for the remaining cells
cell = new Group("cell-02");
```

Important note: There is no particular order to create the hierarchy. However, the insertion order of the devices is conserved throughout the lifecycle of the Group and cannot be changed. That way, the Group implementation can guarantee the order in which results are returned (see below).

Keeping a reference to the root group is enough to manage the whole hierarchy (i.e. there no need to keep trace of the sub-groups or individual devices). The Group interface provides methods to retrieve a sub-group or an individual device.

Be aware that a C++ group allways gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a Group (respectively a DeviceProxy) returned by a call to *Tango::Group::get_group()* (respectively to *Tango::Group::get_device()*). Use the *Tango::Group::remove()* method instead (see the Tango Group class API documentation for details).

We can now perform any action on any element of our "gauges" group. For instance, let's ping the whole hierarchy to be sure that all devices are alive.

```
//- ping the whole hierarchy
if (gauges->ping() == true)
{
    std::cout << "all devices alive" << std::endl;
}
else
{
    std::cout << "at least one dead/busy/locked/... device" << std::endl;
}
```

4.7.2 Forward or not forward?

Since a Tango Group is a hierarchical object, any action performed on a group can be forwarded to its sub-groups. Most of the methods in the Group interface have a so-called *forward* option controlling this propagation. When set to *false*, the action is only performed on the local set of devices. Otherwise, the action is also forwarded to the sub-groups, in other words, propagated along the hierarchy. In C++ , the forward option defaults to true (thanks to the C++ default argument value). There is no such mechanism in Java and the forward option must be systematically specified.

4.7.3 Executing a command

As a proxy for a collection of devices, the Tango Group provides an interface similar to the Device-Proxy's. For the execution of a command, the Group interface contains several implementations of the *command_inout* method. Both synchronous and asynchronous forms are supported.

4.7.3.1 Obtaining command results

Command results are returned using a Tango::GroupCmdReplyList. This is nothing but a vector containing a Tango::GroupCmdReply for each device in the group. The Tango::GroupCmdReply contains the actual data (i.e. the Tango::DeviceData). By inheritance, it may also contain any error occurred during the execution of the command (in which case the data is invalid).

We previously indicated that the Tango Group implementation guarantees that the command results are returned in the order in which its elements were attached to the group. For instance, if g1 is a group containing three devices attached in the following order:

```
g1->add("my/device/01");
g1->add("my/device/03");
g1->add("my/device/02");
```

the results of

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status");
```

will be organized as follows:

crl[0] contains the status of my/device/01
crl[1] contains the status of my/device/03
crl[2] contains the status of my/device/02

Things get more complicated if sub-groups are added "between" devices.

```
g2->add("my/device/04");
g2->add("my/device/05");

g4->add("my/device/08");
g4->add("my/device/09");

g3->add("my/device/06");
g3->add(g4);
g3->add("my/device/07");

g1->add("my/device/01");
g1->add(g2);
g1->add("my/device/03");
g1->add(g3);
g1->add("my/device/02");
```

The result order in the Tango::GroupCmdReplyList depends on the value of the forward option. If set to *true*, the results will be organized as follows:

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status", true);
```

crl[0] contains the status of my/device/01 which belongs to g1
crl[1] contains the status of my/device/04 which belongs to g1.g2
crl[2] contains the status of my/device/05 which belongs to g1.g2
crl[3] contains the status of my/device/03 which belongs to g1
crl[4] contains the status of my/device/06 which belongs to g1.g3
crl[5] contains the status of my/device/08 which belongs to g1.g3.g4
crl[6] contains the status of my/device/09 which belongs to g1.g3.g
crl[7] contains the status of my/device/07 which belongs to g1.g3
crl[8] contains the status of my/device/02 which belongs to g1

If the forward option is set to *false*, the results are:

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status", false);
```

crl[0] contains the status of my/device/01 which belongs to g
crl[1] contains the status of my/device/03 which belongs to g1
crl[2] contains the status of my/device/02 which belongs to g1

The Tango::GroupCmdReply contains some public members allowing the identification of both the device (Tango::GroupCmdReply::dev_name) and the command (Tango::GroupCmdReply::obj_name). It means that, depending of your application, you can associate a response with its source using its position in the response list or using the Tango::GroupCmdReply::dev_name member.

4.7.3.2 Case 1: a command, no argument

As an example, we execute the Status command on the whole hierarchy synchronously.

```
Tango::GroupCmdReplyList crl = gauges->command_inout("Status");
```

As a first step in the results processing, it could be interesting to check value returned by the *has_failed()* method of the GroupCmdReplyList. If it is set to true, it means that at least one error occurred during the execution of the command (i.e. at least one device gave error).

```
if (crl.has_failed())
{
    cout << "at least one error occurred" << endl;
}
else
{
    cout << "no error " << endl;
}
```

In Java, we should write:

```
import fr.esrf.TangoApi.Group;
GroupCmdReplyList crl = gauges.command_inout("Status",true);
if (crl.has_failed())
{
   System.out.println("at least one error occurred");
}
else
{
   System.out.println("no error");
}
```

Now, we have to process each "individual response" in the list.

4.7.3.3 A few words on error handling and data extraction

Depending of the application and/or the developer's programming habits, each individual error can be handle by the C++ (or Java) exception mechanism or using the dedicated *has_failed()* method. The GroupReply class - which is the mother class of both GroupCmdReply and GroupAttrReply - contains a static method to enable (or disable) exceptions called *enable_exception()*. By default, exceptions are disabled (in both Java and C++). The following example is proposed with both exceptions enable and disable.

In C++, data can be extracted directly from an individual reply. The GroupCmdReply interface contains a template operator >> allowing the extraction of any supported Tango type (in fact the actual data extraction is delegated to DeviceData::operator >>). One dedicated extract method is also provided in order to extract DevVarLongStringArray and DevVarDoubleStringArray types to std::vectors.

Error and data handling C++ example:

```
//-----------------------------------------------------------
//- synch. group command example with exception enabled
//-----------------------------------------------------------
//- enable exceptions and save current mode
bool last_mode = GroupReply::enable_exception(true);
//- process each response in the list ...
for (int r = 0; r < crl.size(); r++)
{
//- enter a try/catch block
   try
   {
//- try to extract the data from the r-th reply
//- suppose data contains a double
       double ans;
       crl[r] >> ans;
       cout << crl[r].dev_name()
            << "::"
            << crl[r].obj_name()
            << " returned "
            << ans
            << endl;
    }
    catch (const DevFailed& df)
    {
//- DevFailed caught while trying to extract the data from reply
      for (int err = 0; err < df.errors.length(); err++)
      {
           cout << "error: " << df.errors[err].desc.in() << endl;
      }
//- alternatively, one can use crl[r].get_err_stack() see below
    }
    catch (...)
    {
       cout << "unknown exception caught";
    }
}
//- restore last exception mode (if needed)
GroupReply::enable_exception(last_mode);
//- Clear the response list (if reused later in the code)
crl.reset();

//-----------------------------------------------------------
//- synch. group command example with exception disabled
//-----------------------------------------------------------
//- disable exceptions and save current mode bool
last_mode = GroupReply::enable_exception(false);
//- process each response in the list ...
for (int r = 0; r < crl.size(); r++)
{
//- did the r-th device give error?
    if (crl[r].has_failed() == true)
    {
//- printout error description
       cout << "an error occurred while executing "
```

```
            << crl[r].obj_name()
            << " on "
            << crl[r].dev_name() << endl;
//- dump error stack
       const DevErrorList& el = crl[r].get_err_stack();
       for (int err = 0; err < el.size(); err++)
       {
           cout << el[err].desc.in();
       }
    }
    else
    {
//- no error (suppose data contains a double)
       double ans;
       bool result = crl[r] >> ans;
       if (result == false)
       {
           cout << "could not extract double from "
                << crl[r].dev_name()
                << " reply"
                << endl;
       }
       else
       {
           cout << crl[r].dev_name()
                << "::"
                << crl[r].obj_name()
                << " returned "
                << ans
                << endl;
       }
    }
}
//- restore last exception mode (if needed)
GroupReply::enable_exception(last_mode);
//- Clear the response list (if reused later in the code)
crl.reset();
```

Error and data handling Java example:

```
//-----------------------------------------------------------------
//- synch. group command example with exception enabled
//-----------------------------------------------------------------
//- enable exceptions and save current mode
boolean last_mode = GroupReply.enable_exception(true);
//- process each response in the list ...
Iterator it = crl.iterator();
//- try to extract the data from the each reply
//- suppose data contains a double
double ans;
while (it.hasNext())
```

```
{
//- cast from Object to GroupCmdreply
GroupCmdreply     cr = (GroupCmdreply)it.next();
//- enter a try/catch block
    try
    {
//- extract value from data (may throw DevFailed)
       ans = get_data().extractDouble();
//- verbose
       System.out.println(cr.dev_name()
                        + "::"
                        + cr.obj_name()
                        + " returned "
                        + ans);
    }
    catch (DevFailed d)
    {
//- DevFailed caught while trying to extract the data from reply
       for (int err = 0; err < d.errors.length; err++)
       {
          System.out.println("error: " + d.errors[err].desc);
       }
//- alternatively, one can use cr.get_err_stack() see below
    }
    catch (Exception e)
    {
       System.out.println("unknown exception caught");
    }
}
//- restore last exception mode (if needed)
GroupReply.enable_exception(last_mode);


//---------------------------------------------------------
//- synch. group command example with exception disabled
//---------------------------------------------------------
//- disable exceptions and save current mode
boolean last_mode = GroupReply.enable_exception(false);
//- process each response in the list ...
Iterator it = crl.iterator();
//- try to extract the data from the each reply
while (it.hasNext())
{
//- cast from Object to GroupCmdreply
    GroupCmdreply cr = (GroupCmdreply)it.next();
//- did the device give error?
    if (cr.has_failed() == true)
    {
//- printout error description
       System.out.println("an error occurred while executing "
                 + cr.obj_name()
                 + " on "
                 + cr.dev_name());
//- dump error stack
```

```
        DevError[] de = cr.get_err_stack();
        for (int err = 0; err < de.length; err++)
        {
           System.out.println("error: " + de[err].desc);
        }
     }
     else
     {
//- no error (suppose data contains a double)
        double ans = cr.get_data().extractDouble();
//- verbose
        System.out.println(cr.dev_name()
                           + "::"
                           + cr.obj_name()
                           + " returned "
                           + ans);
     }
}
//- restore last exception mode (if needed)
GroupReply.enable_exception(last_mode);
```

Now execute the same command asynchronously. C++ example:

```
//---------------------------------------------------------
//- asynch. group command example (C++ example)
//---------------------------------------------------------
long request_id = gauges->command_inout_asynch("Status");
//- do some work
do_some_work();


//- get results
crl = gauges->command_inout_reply(request_id);
//- process responses as previously describe in the synch. implementation
for (int r = 0; r < crl.size(); r++)
{
//- data processing and error handling goes here
//- copy/paste code from previous example
. . .
}
//- clear the response list (if reused later in the code)
crl.reset();
```

The same asynchronous example in Java:

```
//---------------------------------------------------------
//- asynch. group command example (Java example)
```

```
//------------------------------------------------------
int request_id = gauges.command_inout_asynch("Status",false,true);
//- do some work
do_some_work();


//- get results
crl = gauges.command_inout_reply(request_id);
//- process responses as previously describe in the synch. implementation
Iterator it = crl.iterator();
//- try to extract the data from the each reply
while (it.hasNext())
{
//- data processing and error handling goes here
//- copy/paste code from previous example
. . .
}
```

4.7.3.4 Case 2: a command, one argument

Here, we give an example in which the same input argument is applied to all devices in the group (or its sub-groups).

In C++:

```
//- the argument value
double d = 0.1;
//- insert it into the TANGO generic container for command: DeviceData
Tango::DeviceData dd;
dd << d;
//- execute the command: Dev_Void SetDummyFactor (Dev_Double)
Tango::GroupCmdReplyList crl = gauges->command_inout("SetDummyFactor", dd);
```

In Java:

```
//- the argument value
double d = 0.1;
//- insert it into the TANGO generic container for command: DeviceData
DeviceData dd = new DeviceData();
dd.insert(d);
//- execute the command: Dev_Void SetDummyFactor (Dev_Double)
GroupCmdReplyList crl = gauges.command_inout("SetDummyFactor", dd, false, true);
```

Since the SetDummyFactor command does not return any value, the individual replies (i.e. the GroupCmdReply) do not contain any data. However, we have to check their *has_failed()* method returned value to be sure that the command completed successfully on each device (acknowledgement). Note that in such a case, exceptions are useless since we never try to extract data from the replies.

In C++ we should have something like:

```
//- no need to process the results if no error occurred (Dev_Void command)
if (crl.has_failed())
{
//- at least one error occurred
    for (int r = 0; r < crl.size(); r++)
    {
//- handle errors here (see previous C++ examples)
    }
}
//- clear the response list (if reused later in the code)
crl.reset();
```

While in Java

```
//- no need to process the results if no error occurred (Dev_Void command)
if (crl.has_failed())
{
//- at least one error occurred
   for (int r = 0; r < crl.length; r++)
   {
//- handle errors here (see previous Java examples)
   }
}
```

See case 1 for an example of asynchronous command.

4.7.3.5 Case 3: a command, several arguments

Here, we give an example in which a **specific** input argument is applied to each device in the hierarchy. In order to use this form of command_inout, the user must have an "a priori" and "perfect" knowledge of the devices order in the hierarchy. In such a case, command arguments are passed in an "array" (with one entry for each device in the hierarchy).

The C++ implementation provides a template method which accepts a std::vector of "C++ type for command argument". This allows passing any kind of data using a single method.

Since templates are not (already) supported in Java, the implementation is somewhat different and an array of DeviceData is used to pass the specific arguments.

In both cases (C++ and Java), the size of this vector must equal the number of device in the hierarchy (respectively the number of device in the group) if the forward option is set to true (respectively set to false). Otherwise, an exception is thrown.

The first item in the vector is applied to the first device in the hierarchy, the second to the second device in the hierarchy, and so on. . . That's why the user must have a "perfect" knowledge of the devices order in the hierarchy.

Assuming that gauges are ordered by name, the SetDummyFactor command can be executed on group "cell-01" (and its sub-groups) as follows:

Remember, "cell-01" has the following internal structure:

```
-> gauges
   | -> cell-01
   |     |-> inst-c01/vac-gauge/strange
   |     |-> penning
   |     |    |-> inst-c01/vac-gauge/penning-01
   |     |    |-> inst-c01/vac-gauge/penning-02
   |     |    |-> ...
   |     |    |-> inst-c01/vac-gauge/penning-xx
   |     |-> pirani
   |          |-> inst-c01/vac-gauge/pirani-01
   |          |-> ...
   |          |-> inst-c01/vac-gauge/pirani-xx
```

Passing a specific argument to each device in C++:

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the hierarchy (starting at cell-01)
long n_dev = g->get_size(true);
//- Build argin list
std::vector<double> argins(n_dev);
//- argument for inst-c01/vac-gauge/strange
argins[0] = 0.0;
//- argument for inst-c01/vac-gauge/penning-01
argins[1] = 0.1;
//- argument for inst-c01/vac-gauge/penning-02
argins[2] = 0.2;
//- argument for remaining devices in cell-01.penning
. . .
//- argument for devices in cell-01.pirani
. . .
//- the reply list
Tango::GroupCmdReplyList crl;
//- enter a try/catch block (see below)
try
{
//- execute the command
    crl = g->command_inout("SetDummyFactor", argins, true);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (const DevFailed& df)
{
//- see below
}
crl.reset();
```

If we want to execute the command locally on "cell-01" (i.e. not on its sub-groups), we should write the following C++ code:

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the group (starting at cell-01)
long n_dev = g->get_size(false);
//- Build argin list
std::vector<double> argins(n_dev);
//- argins for inst-c01/vac-gauge/penning-01
argins[0] = 0.1;
//- argins for inst-c01/vac-gauge/penning-02
argins[1] = 0.2;
//- argins for remaining devices in cell-01.penning
. . .
//- the reply list
Tango::GroupCmdReplyList crl;
//- enter a try/catch block (see below)
try
{
//- execute the command
    crl = g->command_inout("SetDummyFactor", argins, false);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (const DevFailed& df)
{
//- see below
}
crl.reset();
```

Passing a specific argument to each device in Java:

```
//- get a reference to the target group
Group g = gauges.get_group("cell-01");
//- get pre-build arguments list for the whole hierarchy (starting@cell-01)
DeviceData[] argins = g.get_command_specific_argument_list(true);
//- argument for inst-c01/vac-gauge/strange
argins[0].insert(0.0);
//- argument for inst-c01/vac-gauge/penning-01
argins[1].insert(0.1);
//- argument for inst-c01/vac-gauge/penning-02
argins[2].insert(0.2);
//- argument for remaining devices in cell-01.penning
. . .
//- argument for devices in cell-01.pirani
. . .
//- the reply list GroupCmdReplyList
crl = null;
//- enter a try/catch block (see below)
```

```
try
{
//- execute the command
    crl = g.command_inout("SetDummyFactor", argins, false, true);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (DevFailed d)
{
//- see below
}
```

Note: if we want to execute the command locally on "cell-01" (i.e. not on its sub-groups), we should write the following code:

```
//- get a reference to the target group
Group g = gauges.get_group("cell-01");
//- get pre-build arguments list for the group (starting@cell-01)
DeviceData[] argins = g.get_command_specific_argument_list(false);
//- argins for inst-c01/vac-gauge/penning-01
argins[0].insert(0.1);
//- argins for inst-c01/vac-gauge/penning-02
argins[1].insert(0.2);
//- argins for remaining devices in cell-01.penning
. . .
//- the reply list
GroupCmdReplyList crl;
//- enter a try/catch block (see below)
try
{
//- execute the command
    crl = g.command_inout("SetDummyFactor", argins, false, false);
    if (crl.has_failed())
    {
//- error handling goes here (see case 1)
    }
}
catch (DevFailed d)
{
//- see below
}
```

This form of *command_inout* (the one that accepts an array of value as its input argument), may throw an exception **before** executing the command if the number of elements in the input array does not match the number of individual devices in the group or in the hierarchy (depending on the forward option).

Java developers should use the *Group.get_command_specific_argument_list* helper method (see previous example). It guarantees than the "returned array" has the right number of elements. This array may be kept and reused as far as the group size is not changed (i.e. no add or remove of elements).

An asynchronous version of this method is also available. See case 1 for an example of asynchronous command.

4.7.4 Reading attribute(s)

In order to read attribute(s), the Group interface contains several implementations of the *read_attribute()* and *read_attributes()* methods. Both synchronous and asynchronous forms are supported. Reading several attributes is very similar to reading a single attribute. Simply replace the std::string used for attribute name by a vector of std::string with one element for each attribute name. In case of read_attributes() call, the order of attribute value returned in the GroupAttrReplyList is all attributes for first element in the group followed by all attributes for the second group element and so on.

4.7.4.1 Obtaining attribute values

Attribute values are returned using a GroupAttrReplyList. This is nothing but an array containing a GroupAttrReply for each device in the group. The GroupAttrReply contains the actual data (i.e. the DeviceAttribute). By inheritance, it may also contain any error occurred during the execution of the command (in which case the data is invalid).

Here again, the Tango Group implementation guarantees that the attribute values are returned in the order in which its elements were attached to the group. See Obtaining command results for details.

The GroupAttrReply contains some public methods allowing the identification of both the device (GroupAttrReply::dev_name) and the attribute (GroupAttrReply::obj_name). It means that, depending of your application, you can associate a response with its source using its position in the response list or using the Tango::GroupAttrReply::dev_name member.

4.7.4.2 A few words on error handling and data extraction

Here again, depending of the application and/or the developer's programming habits, each individual error can be handle by the C++ exception mechanism or using the dedicated *has_failed()* method. The GroupReply class - which is the mother class of both GroupCmdReply and GroupAttrReply - contains a static method to enable (or disable) exceptions called *enable_exception()*. By default, exceptions are disabled (in both Java and C++). The following example is proposed with both exceptions enable and disable.

In C++, data can be extracted directly from an individual reply. The GroupAttrReply interface contains a template operator>> allowing the extraction of any supported Tango type (in fact the actual data extraction is delegated to DeviceAttribute::operator>>).

Reading an attribute is very similar to executing a command.

Reading an attribute in C++:

```
//---------------------------------------------------------------------
//- synch. read "vacuum" attribute on each device in the hierarchy
//- with exceptions enabled - C++ example
//---------------------------------------------------------------------
//- enable exceptions and save current mode
bool last_mode = GroupReply::enable_exception(true);
//- read attribute
Tango::GroupAttrReplyList arl = gauges->read_attribute("vacuum");
//- for each response in the list ...
for (int r = 0; r < arl.size(); r++)
{
//- enter a try/catch block
   try
```

```
    {
//- try to extract the data from the r-th reply
//- suppose data contains a double
       double ans;
       arl[r] >> ans;
       cout << arl[r].dev_name()
            << "::"
            << arl[r].obj_name()
            << " value is "
            << ans << endl;
    }
    catch (const DevFailed& df)
    {
//- DevFailed caught while trying to extract the data from reply
       for (int err = 0; err < df.errors.length(); err++)
       {
          cout << "error: " << df.errors[err].desc.in() << endl;
       }
//- alternatively, one can use arl[r].get_err_stack() see below
    }
    catch (...)
    {
       cout << "unknown exception caught";
    }
}
//- restore last exception mode (if needed)
GroupReply::enable_exception(last_mode);
//- clear the reply list (if reused later in the code)
arl.reset();
```

Reading an attribute in Java:

```
//-----------------------------------------------------------------
//- synch. read "vacuum" attribute on each device in the hierarchy
//- with exceptions enabled - Java example
//-----------------------------------------------------------------
//- enable exceptions and save current mode
boolean last_mode = GroupReply.enable_exception(true);
//- read attribute
GroupAttrReplyList arl = gauges.read_attribute("vacuum",true);
//- try to extract the data from the each reply
//- suppose data contains a double
double ans;
while (it.hasNext())
{
//- cast from Object to GroupAttrReply
    GroupAttrReply ar = (GroupAttrreply)it.next();
//- enter a try/catch block
    try
    {
```

```
//- extract value from data (may throw DevFailed)
ans = get_data().extractDouble();
//- verbose
       System.out.println(cr.dev_name()
                          + "::"
                          + cr.obj_name()
                          + " returned "
                          + ans);
    }
    catch (DevFailed d)
    {
//- DevFailed caught while trying to extract the data from reply
        for (int err = 0; err < d.errors.length; err++)
        {
           System.out.println("error: " + d.errors[err].desc);
        }
//- alternatively, one can use cr.get_err_stack() see below
    }
    catch (Exception e)
    {
        System.out.println("unknown exception caught");
    }
}
//- restore last exception mode (if needed)
GroupReply.enable_exception(last_mode);
```

In C++, an asynchronous version of the previous example could be:

```
//- read the attribute asynchronously
long request_id = gauges->read_attribute_asynch("vacuum");
//- do some work
do_some_work();


//- get results
Tango::GroupAttrReplyList arl = gauges->read_attribute_reply(request_id);
//- process replies as previously described in the synch. implementation
for (int r = 0; r < arl.size(); r++)
{
//- data processing and/or error handling goes here
...
}
//- clear the reply list (if reused later in the code)
arl.reset();
```

The same asynchronous example in Java:

```
//- read the attribute asynchronously
int request_id = gauges.read_attribute_asynch("vacuum",true);
//- do some work
do_some_work();


//- get results
GroupAttrReplyList arl = gauges.read_attribute_reply(request_id);
Iterator it = arl.iterator();
//- try to extract the data from the each reply
while (it.hasNext())
{
//- data processing and error handling goes here
//- copy/paste code from previous example
. . .
}
```

4.7.5 Writing an attribute

The Group interface contains several implementations of the *write_attribute()* method. Both synchronous and asynchronous forms are supported. However, writing more than one attribute at a time is not supported.

4.7.5.1 Obtaining acknowledgement

Acknowledgements are returned using a GroupReplyList. This is nothing but an array containing a GroupReply for each device in the group. The GroupReply may contain any error occurred during the execution of the command. The return value of the *has_failed()* method indicates whether an error occurred or not. If this flag is set to true, the *GroupReply::get_err_stack()* method gives error details.

Here again, the Tango Group implementation guarantees that the attribute values are returned in the order in which its elements were attached to the group. See Obtaining command results for details.

The GroupReply contains some public members allowing the identification of both the device (GroupReply::dev_name) and the attribute (GroupReply::obj_name). It means that, depending of your application, you can associate a response with its source using its position in the response list or using the GroupReply::dev_name member.

4.7.5.2 Case 1: one value for all devices

Here, we give an example in which the same attribute value is written on all devices in the group (or its sub-groups). Exceptions are supposed to be disabled.

Writing an attribute in C++:

```
//-----------------------------------------------------------------
//- synch. write "dummy" attribute on each device in the hierarchy
//-----------------------------------------------------------------
//- assume each device support a "dummy" writable attribute
//- insert the value to be written into a generic container
Tango::DeviceAttribute value(std::string("dummy"), 3.14159);
//- write the attribute
Tango::GroupReplyList rl = gauges->write_attribute(value);
//- any error?
```

```
if (rl.has_failed() == false)
{
    cout << "no error" << endl;
}
else
{
    cout << "at least one error occurred" << endl;
//- for each response in the list ...
    for (int r = 0; r < rl.size(); r++)
    {
//- did the r-th device give error?
       if (rl[r].has_failed() == true)
       {
//- printout error description
           cout << "an error occurred while reading "
                << rl[r].obj_name()
                << " on "
                << rl[r].dev_name()
                << endl;
//- dump error stack
           const DevErrorList& el = rl[r].get_err_stack();
           for (int err = 0; err < el.size(); err++)
           {
              cout << el[err].desc.in();
           }
        }
     }
}
//- clear the reply list (if reused later in the code)
rl.reset();
```

Writing an attribute in Java:

```
//-----------------------------------------------------------------
//- synch. write "dummy" attribute on each device in the hierarchy
//-----------------------------------------------------------------
//- assume each device support a "dummy" writable attribute
//- insert the value to be written into a generic container
DeviceAttribute value = new DeviceAttribute("dummy"), 3.14159);
//- write the attribute
GroupReplyList rl = gauges.write_attribute(value,true);
//- any error?
if (rl.has_failed() == false)
{
    System.out.println("no error");
}
else
{
    System.out.println("at least one error occurred");
//- for each response in the list ...
```

```
        Iterator it = rl.iterator();
        while (it.hasNext())
        {
    //- cast from Object to GroupReply
            GroupReply gr = (GroupReply)it.next();
    //- did the r-th device give error?
            if (gr.has_failed())
            {
    //- printout error description
                System.out.println("an error occurred while reading "
                      + gr.obj_name()
                      + " on "
                      + gr.dev_name());
    //- dump error stack
                DevError[] el = gr.get_err_stack();
                for (int err = 0; err < el.length; err++)
                {
                    System.out.println(el[err].desc);
                }
            }
        }
     }
```

Here is a C++ asynchronous version:

```
    //- insert the value to be written into a generic container
    Tango::DeviceAttribute value(std::string("dummy"), 3.14159);
    //- write the attribute asynchronously
    long request_id = gauges.write_attribute_asynch(value);
    //- do some work
    do_some_work();


    //- get results
    Tango::GroupReplyList rl = gauges->write_attribute_reply(request_id);
    //- process replies as previously describe in the synch. implementation ...
```

The same asynchronous example in Java:

```
    //- insert the value to be written into a generic container
    DeviceAttribute value = new DeviceAttribute("dummy", 3.14159);
    //- write the attribute asynchronously
    int request_id = gauges.write_attribute_asynch(value,true);
    //- do some work
    do_some_work();


    //- get results
    GroupReplyList rl = gauges.write_attribute_reply(request_id, 0);
    //- process replies as previously describe in the synch. implementation ...
```

4.7.5.3 Case 2: a specific value per device

Here, we give an example in which a **specific** attribute value is applied to each device in the hierarchy. In order to use this form of *write_attribute()*, the user must have an "a priori" and "perfect" knowledge of the devices order in the hierarchy.

The C++ implementation provides a template method which accepts a std::vector of "C++ type for command argument". This allows passing any kind of data using a single method.

Since templates are not (already) supported in Java, the implementation is somewhat different and an array of DeviceAttribute is used to pass the specific arguments.

In both cases (C++ and Java), the size of this vector must equal the number of device in the hierarchy (respectively the number of device in the group) if the forward option is set to true (respectively set to false). Otherwise, an exception is thrown.

The first item in the vector is applied to the first device in the group, the second to the second device in the group, and so on... That's why the user must have a "perfect" knowledge of the devices order in the group.

Assuming that gauges are ordered by name, the dummy attribute can be written as follows on group "cell-01" (and its sub-groups) as follows:

Remember, "cell-01" has the following internal structure:

```
-> gauges
    | -> cell-01
    |     |-> inst-c01/vac-gauge/strange
    |     |-> penning
    |     |     |-> inst-c01/vac-gauge/penning-01
    |     |     |-> inst-c01/vac-gauge/penning-02
    |     |     |-> ...
    |     |     |-> inst-c01/vac-gauge/penning-xx
    |     |-> pirani
    |           |-> inst-c01/vac-gauge/pirani-01
    |           |-> ...
    |           |-> inst-c01/vac-gauge/pirani-xx
```

C++ version:

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the hierarchy (starting at cell-01)
long n_dev = g->get_size(true);
//- Build value list
std::vector<double> values(n_dev);
//- value for inst-c01/vac-gauge/strange
values[0] = 3.14159;
//- value for inst-c01/vac-gauge/penning-01
values[1] = 2 * 3.14159;
//- value for inst-c01/vac-gauge/penning-02
values[2] = 3 * 3.14159;
//- value for remaining devices in cell-01.penning
. . .
//- value for devices in cell-01.pirani
. . .
//- the reply list
```

```
Tango::GroupReplyList rl;
//- enter a try/catch block (see below)
try
{
//- write the "dummy" attribute
    rl = g->write_attribute("dummy", values, true);
    if (rl.has_failed())
    {
//- error handling (see previous cases)
    }
}
catch (const DevFailed& df)
{
//- see below
}
rl.reset();
```

Here is a Java version:

```
//- get a reference to the target group
Group g = gauges.get_group("cell-01");
//- get pre-build arguments list for the whole hierarchy (starting@cell-01)
DeviceAttribute[] values = g.get_attribute_specific_value_list(true);
//- value for inst-c01/vac-gauge/strange
values[0] = 3.14159;
//- value for inst-c01/vac-gauge/penning-01
values[1] = 2 * 3.14159;
//- value for inst-c01/vac-gauge/penning-02
values[2] = 3 * 3.14159;
//- value for remaining devices in cell-01.penning
. .
//- value for devices in cell-01.pirani
. . .
//- the reply list
GroupReplyList rl;
try
{
//- write the "dummy" attribute
rl = g.write_attribute("dummy", values, true);
   if (rl.has_failed())
   {
//- error handling (see previous cases)
   }
}
catch (DevFailed d)
{
//- see below
}
```

Note: if we want to execute the command locally on "cell-01" (i.e. not on its sub-groups), we should write the following code (example is only proposed for C++ - Java port is straightforward):

```
//- get a reference to the target group
Tango::Group *g = gauges->get_group("cell-01");
//- get number of device in the group
long n_dev = g->get_size(false);
//- Build value list
std::vector<double> values(n_dev);
//- value for inst-c01/vac-gauge/penning-01
values[0] = 2 * 3.14159;
//- value for inst-c01/vac-gauge/penning-02
values[1] = 3 * 3.14159;
//- value for remaining devices in cell-01.penning
. . .
//- the reply list
Tango::GroupReplyList rl;
//- enter a try/catch block (see below)
try
{
//- write the "dummy" attribute
   rl = g->write_attribute("dummy", values, false);
   if (rl.has_failed())
   {
//- error handling (see previous cases)
   }
}
catch (const DevFailed& df)
{
//- see below
}
rl.reset();
```

This form of *write_attribute()* (the one that accepts an array of value as its input argument), may throw an exception before executing the command if the number of elements in the input array does not match the number of individual devices in the group or in the hierarchy (depending on the forward option).

Java developers should use the *Group.get_attribute_specific_value_list* helper method (see previous example). It guarantees than the "returned array" has the right number of elements. This array may be kept and reused as far as the group size is not changed (i.e. no add or remove of elements).

An asynchronous version of this method is also available.

4.8 Device locking

Starting with Tango release 7 (and device inheriting from Device_4Impl), device locking is supported. For instance, this feature could be used by an application doing a scan on a synchrotron beam line. In such a case, you want to move an actuator then read a sensor, move the actuator again, read the sensor...You don't want the actuator to be moved by another client while the application is doing the scan. If the application doing the scan locks the actuator device, it will be sure that this device is "reserved" for the application doing the scan and other client will not be able to move it until the scan application un-locks this actuator.

A locked device is protected against:

- *command_inout* call except for device state and status requested via command and for the set of commands defined as allowed following the definition of allowed command in the Tango control access schema.
- *write_attribute* call
- *write_read_attribute* call
- *set_attribute_config* call
- polling and logging commands related to the locked device

Other clients trying to do one of these calls on a locked device will get a DevFailed exception. In case of application with locked device crashed, the lock will be automatically release after a defined interval. The API provides a set of methods for application code to lock/unlock device. These methods are:

- *DeviceProxy::lock()* and *DeviceProxy::unlock()* to lock/unlock device
- *DeviceProxy::locking_status()*, *DeviceProxy::is_locked()*, *DeviceProxy::is_locked_by_me()* and *DeviceProxy::get_locker()* to get locking information

These methods are precisely described in the API reference chapters.

4.9 Reconnection and exception

The Tango API automatically manages re-connection between client and server in case of communication error during a network access between a client and a server. By default, when a communication error occurs, an exception is returned to the caller and the connection is internally marked as bad. On the next try to contact the device, the API will try to re-build the network connection. With the *set_transparency_reconnection()* method of the DeviceProxy class, it is even possible not to have any exception thrown in case of communication error. The API will try to re-build the network connection as soon as it is detected as bad. This is the default mode. See 6.17 for more details on this subject.

4.10 Thread safety

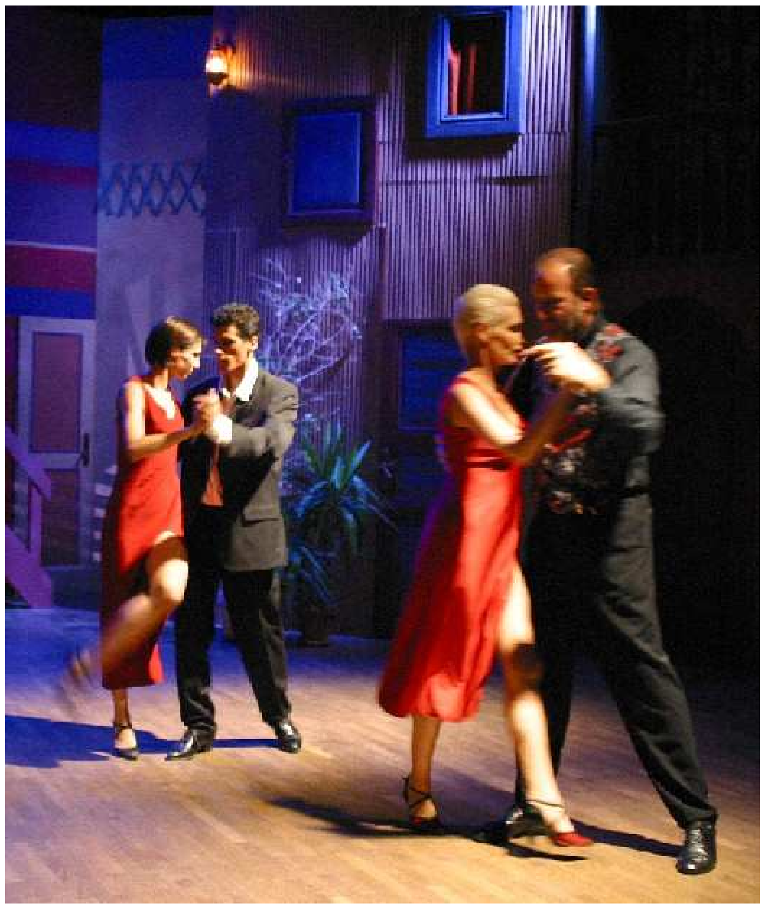
Starting with Tango 7.2, some classes of the C++ API has been made thread safe. These classes are:

- DeviceProxy
- Database
- Group
- ApiUtil
- AttributeProxy

This means that it is possible to share between threads a pointer to a DeviceProxy instance. It is safe to execute a call on this DeviceProxy instance while another thread is also doing a call to the same DeviceProxy instance. Obviously, this also means that it is possible to create thread local DeviceProxy instances and to execute method calls on these instances. Nevertheless, data local to a DeviceProxy instance like its timeout are not managed on a per thread basis. For a DeviceProxy instance shared between two threads, if thread 1 changes the instance timeout, thread 2 will also see this change.

4.11 Compiling and linking a Tango client

Compiling and linking a Tango client is similar to compiling and linking a Tango device server. Please, refer to chapter "Compiling, Linking and executing a Tango device server process" (8.6) to get all the details.

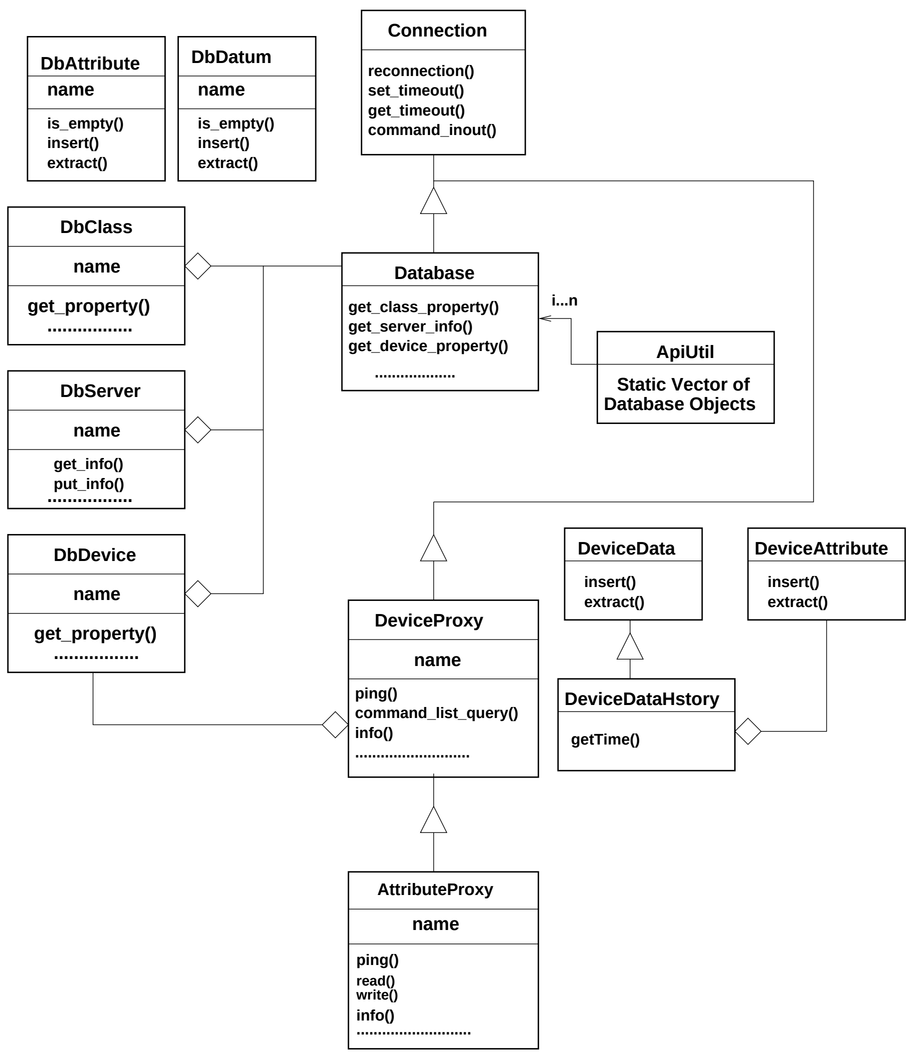


Chapter 5

TANGO Java API

THIS CHAPTER DOCUMENTS THE JAVA API FOR THE TANGO DATABASE AND DEVICE SERVERS.

TANGO Java API Architecture



5.1 Introduction

5.1.1 Description

This chapter documents the high level interface for Java.

Remarks:

This java api is based on *Jacorb* ORB implementation. The *Jacorb* and *Tango* classes are both available in TangORB.jar file.

5.1.2 Basic Philosophy

The basic philosophy is to have high level classes for the database, properties, device, group and database object info. Classes also exist for sending and receiving database or device values.

All classes and data types are defined in *fr.esrf.TangoApi* package. Group related classes are in a package called *fr.esrf.TangoApi.Group*. Event related classes are in a package called *fr.esrf.TangoApi.events*

5.1.3 Classes

5.1.3.1 Data object classes

DeviceData: Obect used to send and receive data on device.

DbDatum: Object used to put or get properties on database.

DbDevInfo: Object used to read device information on database.

DbDevImportInfo: Object used to read imported device information on database.

DbDevExportInfo: Object used to read exported device information on database.

5.1.3.2 Asynchronous callback related classes

CallBack: Object called at asynchronous call reply

CmdDoneEvent: Object to pass asynchronous command reply data to a CallBack object.

ReadAttrEvent: Object to pass asynchronous read_attribute reply data to a CallBack object.

AttrWrittenEvent: Object to pass asynchronous write_attribute reply data to a CallBack object.

5.1.3.3 Devices and Database access classes

DeviceProxy: Device access (aggregates DbDevice class).

Group: Multiple device access class

Database: Direct access to TANGO database.

DbClass: Class properties access to TANGO database.

DbServer: Server properties access to TANGO database.

DbDevice Device properties access to TANGO database.

5.1.4 Reporting errors

For the device and database classes, most methods throw a *DevFailed* exception in case of error. See *Writing a TANGO Device Server* chapter *Reporting Errors* (8.2.4) , except those which specified.

In opposite, for the data object classes, only the specified method throw *DevFailed* exception in case of error.

The reason field could be set to:

- *TangoApi_TANGO_HOST_NOT_SET* : The *TANGO_HOST* environment variable has not been set or has been set with a syntax error.
- *TangoApi_DATABASE_CONNECTION_FAILED* : The database server cannot be connected (bad *TANGO_HOST* or database server stopped).
- *TangoApi_CANNOT_IMPORT_DEVICE* : The device is exported but cannot be connected.
- *TangoApi_DEVICE_NOT_EXPORTED* : The device has not be exported.

5.1.5 Compiling a Java client

5.1.5.1 Supported java release

Tango client written using Java language needs release **1.5.0** (or above) of the Java environment.

5.1.5.2 Setting CLASSPATH and other environment variables

To correctly compile a Java Tango client, the CLASSPATH environment variable must be set to :

- The jar file with all the Tango, TangoDs, TangoApi and Jacorb package classes. This file is named TangORB.jar
- The jar file with all the JDK classes (not always necessary, could be implicit)
- Your own package directory

For UNIX like operating system, setting environment variable is done with the *export* or *setenv* command depending on the shell used. For Windows, setting environment variable is possible from the control panel.

The client/server timeout as been fixed by default to 3000 milliseconds but it can be set to another value a startup using *TANGO_TIMEOUT* environement variable.

eg : *java -DTANGO_HOST=hal:20000 -DTANGO_TIMEOUT=5000 mypackage.MyClient*

Will start *MyClient* class using the database server running on the host named *hal* on port 20000 with a command timeout of 5 seconds.

5.2 Reference manual

The Tango Java API documentation is now managed using the Java tool javadoc and is available online at

Tango Java API reference documentation

Chapter 6

The TANGO C++ Application Programmer Interface

6.1 Tango::DeviceProxy()

The high level object which provides the client with an easy-to-use interface to TANGO devices. DeviceProxy is a handle to the real Device (hence the name Proxy) and is not the real Device (of course). DeviceProxy provides interfaces to all TANGO Device interfaces. The DeviceProxy manages timeouts, stateless connections (new DeviceProxy() nearly always works), and reconnection if the device server is restarted.

6.1.1 Constructors

6.1.1.1 DeviceProxy::DeviceProxy(string &name, CORBA::ORB *orb=NULL)

Create a DeviceProxy to a device of the specified name. The TANGO_HOST environment variable is used to determine which TANGO database to connect to. The client can specify an ORB as argument if she wants to. The constructor will connect to the TANGO database, query for the client's network address and build a connection to the device. If the device is defined in the TANGO database but the device server is not running DeviceProxy will try to build a connection every time the client tries to access the device. If the device is not defined an exception is thrown. Example :

```
DeviceProxy *my_device = new DeviceProxy("my/own/device");
```

See appendix on device naming for all details about Tango device naming syntax. If an alias name is defined for the device, this alias name can be used to create the DeviceProxy instance.

Exception: WrongNameSyntax, ConnectionFailed

6.1.1.2 DeviceProxy::DeviceProxy(const char *name, CORBA::ORB *orb = NULL)

Idem previous call

6.1.2 Miscellaneous methods

6.1.2.1 DeviceInfo DeviceProxy::info()

A method which returns information on the device in a DeviceInfo structure. Example :

```
cout << " device info : " << endl
DeviceInfo dev_info = my_device->info() << endl;
cout << " dev_class " << dev_info.dev_class;
cout << " server_id " << dev_info.server_id;
cout << " server_host " << dev_info.server_host;
cout << " server_version " << dev_info.server_version;
cout << " doc_url " << dev_info.doc_url;
cout << " device_type " << dev_info.dev_type;
```

All DeviceInfo fields are strings except for the server_version server_version which is a long integer.
Exception: Connection Failed, CommunicationFailed, DevFailed from device

6.1.2.2 DevState DeviceProxy::state()

A method which return the state of the device as a Tango::DevState type. Example :

```
dev_state = my_device->state() << endl;
```

Exception: ConnectionFailed, CommunicationFailed

6.1.2.3 string DeviceProxy::status()

A method which return the status of the device as a string. Example :

```
cout << "device status " << my_device->status() << endl;
```

Exception: ConnectionFailed, CommunicationFailed

6.1.2.4 int DeviceProxy::ping()

A method which sends a ping to the device and returns the time elapsed as microseconds. Example :

```
cout << " device ping took " << my_device->ping() << " microseconds" << endl;
```

Exception: ConnectionFailed, CommunicationFailed

6.1.2.5 void DeviceProxy::set_timeout_millis(int timeout)

Set client side timeout for device in milliseconds. Any method which takes longer than this time to execute will throw an exception.
Exception: none

6.1.2.6 int DeviceProxy::get_timeout_millis()

Get the client side timeout in milliseconds.

Exception: none

6.1.2.7 int DeviceProxy::get_idl_version()

Get the version of the Tango Device IDL interface implemented by the device

Exception: none

6.1.2.8 void DeviceProxy::set_source(DevSource source)

Set the data source (device, polling buffer, polling buffer than device) for command_inout and read_attribute methods. The DevSource is an enumerated type which can be one of {DEV, CACHE, CACHE_DEV}. The default value is CACHE_DEV. See chapter on Advanced Feature for all details regarding polling.

Exception: none

6.1.2.9 DevSource DeviceProxy::get_source()

Get the device data source used by command_inout or read_attribute methods. The DevSource is an enumerated type which can be one of {DEV, CACHE, CACHE_DEV}. See chapter on Advanced Feature for all details regarding polling.

Exception: none

6.1.2.10 vector<string> *DeviceProxy::black_box(int n)

Get the last n commands executed on the device server and return a pointer to a vector of strings containing the date, time, command, and from which client computer the command was executed. This method allocates memory for the vector of strings returned to the caller. It is the caller responsibility to delete this memory.

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.2.11 string DeviceProxy::name()

Return the device name (from the device itself)

Exception: ConnectionFailed, CommunicationFailed

6.1.2.12 string DeviceProxy::adm_name()

Returns the name of the corresponding administrator device. This is useful if you need to send an administration command to the device server e.g. restart it.

Exception: ConnectionFailed, CommunicationFailed

6.1.2.13 string DeviceProxy::dev_name()

Return the device name as it is stored locally

6.1.2.14 string DeviceProxy::description()

Returns the device description as a string.

Exception: ConnectionFailed, CommunicationFailed

6.1.2.15 DbDevImportInfo DeviceProxy::import_info()

Query the device for import info from the database. This method returns a DbDevImprtInfo type. The DbDevImportInfo type is a struct defined as follows :

```
class DbDevImportInfo {
public :
    string name;
    long exported;
    string ior;
    string version; };
```

Exception: NonDbDevice

6.1.2.16 void DeviceProxy::set_transparency_reconnection(bool flag)

If flag is true, no exception will be thrown in case of network communication error between client and server. The API will try to re-build the network connection between client and server as soon as an error is detected. See 6.17 more more details on reconnection and exception

6.1.2.17 bool DeviceProxy::get_transparency_reconnection()

Returns the transparency reconnection flag.

6.1.2.18 string DeviceProxy::alias()

Returns the device alias name if one is defined otherwise, throws a DevFailed exception with the reason field set to Db_AliasNotDefined.

6.1.2.19 AccessControlType DeviceProxy::get_access_right()

Returns the device access right. AccessControlType is one enumeration with 2 values which are ACCESS_READ and ACCESS_WRITE. In case the Tango Access Control systemis not used, ACCESS_WRITE is returned.

6.1.3 Synchronous command oriented methods

6.1.3.1 CommandInfo DeviceProxy::command_query(string command)

Query the device for information about a single command. This command returns a single CommandInfo type. The CommandInfo type is a struct described in command_list_query().

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.3.2 CommandInfoList *DeviceProxy::command_list_query()

Query the device for info on all commands. This method returns a vector of CommandInfo types. This method allocates memory for the vector of CommandInfo returned to the caller. It is the caller responsibility to delete this memory. The CommandInfo type is a struct defined as follows :

```
typedef _CommandInfo
{
    string           cmd_name;      /* command name as ascii string */
    long             cmd_tag;       /* command as binary value (for TACO) */
    long             in_type;       /* in type as binary value */
    long             out_type;      /* out type as binary value */
    string           in_type_desc;  /* description of in type (optional) */
    string           out_type_desc; /* description of out type (optional) */
    Tango::DispLevel disp_level;    /* Command display level */
} CommandInfo;
typedef CommandInfoList vector<CommandInfo>;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.3.3 DeviceData DeviceProxy::command_inout(string)

Execute a command on a device which takes no input arguments (void). The result is returned in a DeviceData object (cf. below how to insert and extract data from DeviceData).

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.1.3.4 DeviceData DeviceProxy::command_inout(const char *)

Idem previous call

6.1.3.5 DeviceData Deviceproxy::command_inout(string, DeviceData &)

Execute a command on a device. Input arguments are passed in a DeviceData object, output is returned as a DeviceData object (see below on how to insert and extract data from DeviceData).

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.1.3.6 DeviceData DeviceProxy::command_inout(const char *, DeviceData &)

Idem previous call

6.1.3.7 vector<DeviceDataHistory> *command_history(string &, int)

Retrieve command history from the command polling buffer. The first argument is the command name. The second argument is the wanted history depth. This method returns a vector of DeviceDataHistory types. This method allocates memory for the vector of DeviceDataHistory returned to the caller. It is the caller responsibility to delete this memory. Class DeviceDataHistory is detailed on chapter 6.3. See chapter on Advanced Feature for all details regarding polling.

```
DeviceProxy dev = new DeviceProxy("...");
vector<DeviceDataHistory> *hist;
hist = dev->command_history("Status",5);
for (int i = 0;i < 5;i++)
{
    bool fail = (*hist)[i].failed();
    if (fail == false)
    {
        string str;
        (*hist)[i] >> str;
```

```
            cout << "Status = " << str << endl;
        }
        else
        {
            cout << "Command failed !" << endl;
            cout << "Error level 0 desc = " << ((*hist)[i].errors())[0].desc << endl;
        }
        cout << "Date = " << (*hist)[i].date().tv_sec << endl;
    }
    delete hist;
```

Exception: NonSupportedFeature, ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.3.8 DeviceDataHistoryList *command_history(const char *, int)

Idem previous call

6.1.4 Synchronous attribute related methods

6.1.4.1 Compatibility between Tango release 4 and release 5 regarding attribute properties

Between Tango V4 and Tango V5, attribute configuration has been modified to incorporate alarm and event related parameters. This explains why it exists two structure types for attribute configuration parameters. All Tango V4 parameters are defined in a structure called **AttributeInfo** and a new structure called **AttributeInfoEx** has been defined for all Tango V5 parameters. Nevertheless, AttributeInfoEx inherits from AttributeInfo and it is always possible to call the Tango V5 *DeviceProxy::attribute_query()* method and to store its result in one AttributeInfo structure thus allowing compatibility for client written for Tango V4 but linked with Tango V5. It is also possible for a client written and linked with Tango V5 to call Tango V5 *DeviceProxy::attribute_query()* method to all kind of Tango devices. For device using Tango V4, the alarm and event related parameters will be retrieved from the database instead of from the device.

6.1.4.2 AttributeInfoEx DeviceProxy::attribute_query(string attribute)

Query the device for information about a single attribute. This command returns a single AttributeInfoEx type which inherits from the AttributeInfo type. The AttributeInfoEx and AttributeInfo types are structures described in get_attribute_config() and get_attribute_config_ex().

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.4.3 AttributeInfoList * DeviceProxy::attribute_list_query()

Query the device for info on all attributes. This method returns a vector of AttributeInfo types. The AttributeInfo type is a structure described in get_attribute_config(). This method allocates memory for the vector of AttributeInfo structures returned to the caller. It is the caller responsibility to delete this memory.

6.1.4.4 AttributeInfoListEx * DeviceProxy::attribute_list_query_ex()

Query the device for info on all attributes. This method returns a vector of AttributeInfoEx types. The AttributeInfoEx type is a structure described in get_attribute_config_ex(). This method allocates memory for the vector of AttributeInfoEx structures returned to the caller. It is the caller responsibility to delete this memory.

6.1.4.5 vector<string> *DeviceProxy::get_attribute_list()

Return the names of all attributes implemented for this device as a vector of strings. This method allocates memory for the vector of strings returned to the caller. It is the caller responsibility to delete this memory.

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.4.6 AttributeInfoList *DeviceProxy::get_attribute_config(vector<string>&)

Return the attribute configuration for the list of specified attributes. To get all the attributes pass a vector containing the string AllAttr (defined in tango_const.h). This method allocates memory for the vector of AttributeInfo returned to the caller. It is the caller responsibility to delete this memory. AttributeInfo is a struct defined as follows :

```
typedef struct _AttributeInfo
{
   string             name;
   AttrWriteType      writable;
   AttrDataFormat     data_format;
   int                data_type;
   int                max_dim_x;
   int                max_dim_y;
   string             description;
   string             label;
   string             unit;
   string             standard_unit;
   string             display_unit;
   string             format;
   string             min_value;
   string             max_value;
   string             min_alarm;
   string             max_alarm;
   string             writable_attr_name;
   vector<string>     extensions;
   Tango::DispLevel   disp_level;
} AttributeInfo;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.4.7 AttributeInfoListEx *DeviceProxy::get_attribute_config_ex(vector<string>&)

Return the extended attribute configuration for the list of specified attributes. To get all the attributes pass a vector containing the string AllAttr (defined in tango_const.h). This method allocates memory for the vector of AttributeInfoEx returned to the caller. It is the caller responsibility to delete this memory. AttributeInfoEx is a structure defined as follows :

```
struct AttributeInfoEx: public AttributeInfo
{
   AttributeAlarmInfo alarms;
   AttributeEventInfo events;
   vector<string>     sys_extensions;
```

```
};

struct AttributeAlarmInfo
{
   string           min_alarm;
   string           max_alarm;
   string           min_warning;
   string           max_warning;
   string           delta_t;
   string           delta_val;
   vector<string>   extensions;
};

struct AttributeEventInfo
{
   ChangeEventInfo     ch_event;
   PeriodicEventInfo   per_event;
   ArchiveEventInfo    arch_event;
};

struct ChangeEventInfo
{
   string           rel_change;
   string           abs_change;
   vector<string>   extensions;
};

struct PeriodicEventInfo
{
   string           period;
   vector<string>   extensions;
};

struct ArchiveEventInfo
{
   string           archive_rel_change;
   string           archive_abs_change;
   string           archive_period;
   vector<string>   extensions;
};
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.4.8 AttributeInfoEx DeviceProxy::get_attribute_config(string&)

Return the attribute configuration for a single attributes. The AttributeInfoEx is a structure defined above.
Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.4.9 void DeviceProxy::set_attribute_config(AttributeInfoList &)

Change the attribute configuration for the specified attributes.
Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.1.4.10 void DeviceProxy::set_attribute_config(AttributeInfoListEx &)

Change the attribute configuration for the specified attributes.

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.1.4.11 vector<DeviceAttribute> *DeviceProxy::read_attributes(vector<string>&)

Read the list of specified attributes. To extract the value you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. By default, if the server reports error for one of the attribute in the list, this error will be passed to the user using exception when he (she) will try to extract the data form the corresponding See sub-chapter on DeviceAttribute to learn how to change this default behaviour. DeviceAttribute object. This method allocates memory for the vector of DeviceAttribute objects returned to the caller. This is the caller responsibility to delete this memory. Example :

```
vector<DeviceAttribute> *devattr;
vector<string> attr_names;

attr_names.push_back("attribute_1");
attr_names.push_back("attribute_2");
devattr = device->read_attributes(attr_names);
short short_attr_1;
long long_attr_2;
(*devattr)[0] >> short_attr_1;
(*devattr)[1] >> long_attr_2;
cout << "my_attribute value " << short_attr;
delete devattr;
```

Exception: ConnectionFailed, CommunicationFailed

6.1.4.12 DeviceAttribute DeviceProxy::read_attribute(string&)

Read a single attribute. To extract the value you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double (this will return 0) you have to extract it as a short. See example above.

Exception: ConnectionFailed, CommunicationFailed

6.1.4.13 DeviceAttribute DeviceProxy::read_attribute(const char *)

Idem previous call

6.1.4.14 void DeviceProxy::write_attributes(vector<DeviceAttribute>&)

Write the specified attributes. To insert the values to write you have to use the operator of the DeviceAttribute class which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short. Note that this is the only API call which could throw a NamedDevFailedList exception. See 6.16.10 to get all the details on this exception. Example :

```
vector<DeviceAttribute> attr_in;
string att1_name("First_attr");
string att2_name("Second_attr");
short short_attr;
double double_attr; attr_in.push_back(DeviceAttribute(att1_name,short_attr));
attr_in.push_back(DeviceAttribute(att2_name,double_attr));
device->write_attributes(attr_in);
```

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed or NamedDevFailedList from device

6.1.4.15 void DeviceProxy::write_attribute(DeviceAttribute&)

Write a single attribute. To insert the value to write you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short. See example above.

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.1.4.16 DeviceAttribute DeviceProxy::write_read_attribute(DeviceAttribute&)

Write then read a single attribute in a single network call. By default (serialisation by device), the execution of this call in the server can't be interrupted by other clients. To insert/extract the value to write/read you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short.

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.1.4.17 vector<DeviceAttributeHistory> *DeviceProxy::attribute_history(string &, int)

Retrieve attribute history from the attribute polling buffer. The first argument is the attribute name. The second argument is the wanted history depth. This method returns a vector of DeviceAttributeHistory types. This method allocates memory for the vector of DeviceAttributeHistory returned to the caller. It is the caller responsibility to delete this memory. Class DeviceAttributeHistory is detailed on chapter 6.5See also chapter on Advanced Feature for all details regarding polling.

```
DeviceProxy dev = new DeviceProxy("...");
vector<DeviceAttributeHistory> *hist;
hist = dev->attribute_history("Current",5);
for (int i = 0;i < 5;i++)
{
    bool fail = (*hist)[i].has_failed();
    if (fail == false)
    {
        cout << "Attribute name = " << (*hist)[i].get_name() << endl;
        cout << "Attribute quality factor = " << (*hist)[i].get_quality() << endl;
        long value;
```

```
            (*hist)[i] >> value;
            cout << "Current = " << value << endl;
        }
        else
        {
            cout << "Attribute failed !" << endl;
            cout << "Error level 0 desc = " << ((*hist)[i].get_err_stack())[0].desc << endl;
        }
        cout << "Date = " << (*hist)[i].get_date().tv_sec << endl;
    }
    delete hist;
```

Exception: NonSupportedFeature, ConnectionFailed, CommunicationFailed, DevFailed from device

6.1.4.18 vector<DeviceAttributeHistory> *DeviceProxy::attribute_history(const char *, int)

Idem previous call

6.1.5 Asynchronous command oriented methods

6.1.5.1 long DeviceProxy::command_inout_asynch(string &name, bool forget)

Execute asynchronously (polling model) a command on a device which takes no input argument. The last argument is a *fire and forget* flag. If this flag is set to true, this means that the client does not care at all about the server answer and will even not try to get it. A false default value is provided. Please, note that device re-connection will not take place (in case it is needed) if the fire and forget mode is used. Therefore, an application using only fire and forget requests is not able to automatically re-connnect to device. This call returns an *asynchronous call identifier* which is needed to get the command result.

Exception: ConnectionFailed

6.1.5.2 long DeviceProxy::command_inout_asynch(const char *name, bool forget)

Idem previous call

6.1.5.3 long DeviceProxy::command_inout_asynch(string &name, DeviceData &argin, bool forget)

Execute asynchronously (polling model) a command on a device. Input arguments are passed in a DeviceData object (see following chapters on how to insert data into DeviceData object). The last argument is a *fire and forget* flag. If this flag is set to true, this means that the client does not care at all about the server answer and will even not try to get it. A false default value is provided. Please, note that device re-connection will not take place (in case it is needed) if the fire and forget mode is used. Therefore, an application using only fire and forget requests is not able to automatically re-connnect to device. This call returns an *asynchronous call identifier* which is needed to get the command result.

Exception: ConnectionFailed

6.1.5.4 long DeviceProxy::command_inout_asynch(const char *name, Devicedata &argin, bool forget)

Idem previous call

6.1.5.5 DeviceData DeviceProxy::command_inout_reply(long id)

Check if the answer of an asynchronous command_inout is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceData object. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived. Example :

```
Tango::DeviceProxy dev("...");
long asyn_id;
asyn_id = dev.command_inout_asynch("MyCmd");
...
...
...
Tango::DeviceData arg;
try
{
        arg = dev.command_inout_reply(asyn_id);
}
catch(Tango::AsynReplyNotArrived)
{
        cerr << "Command not arrived !" << endl;
}
catch (Tango::DevFailed &e)
{
        Tango::Except::print_exception(e);
}
```

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.5.6 DeviceData DeviceProxy::command_inout_reply(long id, long timeout)

Check if the answer of an asynchronous command_inout is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceData object. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.5.7 void DeviceProxy::command_inout_asynch(string &name, CallBack &cb)

Execute asynchronously (callback model) a command on a device which takes no input argument. The last argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *cmd_ended()* method overloaded.

Exception: ConnectionFailed

6.1.5.8 void DeviceProxy::command_inout_asynch(const char *name, CallBack &cb)

Idem previous call

6.1.5.9 void DeviceProxy::command_inout_asynch(string &name, DeviceData &argin, CallBack &cb)

Execute asynchronously (callback model) a command on a device. Input arguments are passed in a DeviceData object (see following chapters on how to insert data into DeviceData object). The last argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *cmd_ended()* method overloaded.

Exception: ConnectionFailed

6.1.5.10 void DeviceProxy::command_inout_asynch(const char *name, DeviceData &argin, CallBack &cb)

Idem previous call

6.1.6 Asynchronous attribute related methods

6.1.6.1 long DeviceProxy::read_attribute_asynch(string &name)

Read asynchronously (polling model) a single attribute. This call returns an *asynchronous call identifier* which is needed to get the attribute value.

Exception: ConnectionFailed

6.1.6.2 long DeviceProxy::read_attribute_asynch(const char *name)

Idem previous call

6.1.6.3 long DeviceProxy::read_attributes_asynch(vector<string> &names)

Read asynchronously (polling model) the list of specified attributes. This call returns an *asynchronous call identifier* which is needed to get attributes value.

Exception: ConnectionFailed

6.1.6.4 DeviceAttribute *DeviceProxy::read_attribute_reply(long id)

Check if the answer of an asynchronous read_attribute is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceAttribute object. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived. To extract attribute value, you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the DeviceAttribute object returned to the caller. This is the caller responsibility to delete this memory.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.5 DeviceAttribute *DeviceProxy::read_attribute_reply(long id, long timeout)

Check if the answer of an asynchronous read_attribute is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceAttribute object. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived. To extract attribute value, you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the DeviceAttribute object returned to the caller. This is the caller responsibility to delete this memory.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.6 vector<DeviceAttribute> *DeviceProxy::read_attributes_reply(long id)

Check if the answer of an asynchronous read_attributes is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a vector<DeviceAttribute>. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived. To extract attribute value, you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the vector<DeviceAttribute> object returned to the caller. This is the caller responsibility to delete this memory.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.7 vector<DeviceAttribute> *DeviceProxy::read_attributes_reply(long id, long timeout)

Check if the answer of an asynchronous read_attributes is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a vector<DeviceAttribute>. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived. To extract attribute value, you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the vector<DeviceAttribute> object returned to the caller. This is the caller responsibility to delete this memory.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.8 long DeviceProxy::write_attribute_asynch(DeviceAttribute &argin)

Write asynchronously (polling model) a single attribute. To insert the value to write you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short. This call returns an *asynchronous call identifier* which is needed to get the server reply.

Exception: ConnectionFailed

6.1.6.9 long DeviceProxy::write_attributes_asynch(vector<DeviceAttribute> &argin)

Write asynchronously (polling model) the specified attributes. To insert the value to write you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short. This call returns an *asynchronous call identifier* which is needed to get the server reply.

Exception: ConnectionFailed

6.1.6.10 void DeviceProxy::write_attribute_reply(long id)

Check if the answer of an asynchronous write_attribute is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.11 void DeviceProxy::write_attribute_reply(long id, long timeout)

Check if the answer of an asynchronous write_attribute is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.12 void DeviceProxy::write_attributes_reply(long id)

Check if the answer of an asynchronous write_attributes is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.13 void DeviceProxy::write_attributes_reply(long id, long timeout)

Check if the answer of an asynchronous write_attributes is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.1.6.14 void DeviceProxy::read_attribute_asynch(string &name, CallBack &cb)

Read asynchronously (callback model) a single attribute. The last argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *attr_read()* method overloaded.

Exception: ConnectionFailed

6.1.6.15 void DeviceProxy::read_attribute_asynch(const char *name, CallBack &cb)

Idem previous call

6.1.6.16 void DeviceProxy::read_attributes_asynch(vector<string> &names, CallBack &cb)

Read asynchronously (callback model) an attribute list. The last argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *attr_read()* method overloaded.

Exception: ConnectionFailed

6.1.6.17 void DeviceProxy::write_attribute_asynch(DeviceAttribute &argin, CallBack &cb)

Write asynchronously (callback model) a single attribute. The last argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *attr_written()* method overloaded.

Exception: ConnectionFailed

6.1.6.18 void DeviceProxy::write_attributes_asynch(vector<DeviceAttribute> &argin, CallBack &cb)

Write asynchronously (callback model) an attribute list. The last argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *attr_written()* method overloaded.

Exception: ConnectionFailed

6.1.7 Miscellaneous asynchronous related methods

6.1.7.1 long DeviceProxy::pending_asynch_call(asyn_req_type req)

Return number of device asynchronous pending requests. The input parameter is an enumeration with three values which are:

POLLING : Returns only device polling model asynchronous request number

CALLBACK : Returns only device callback model asynchronous request number

ALL_ASYNCH : Returns device asynchronous request number

Exception: None

6.1.7.2 void DeviceProxy::get_asynch_replies()

Fire callback methods for device asynchronous requests with already arrived replied. Returns immediately if there is no replies already arrived or if there is no asynchronous request for the device. Example :

```
class MyCallBack: Tango::CallBack
{
public:
   MyCallback(double d):data(d) {};
   virtual void cmd_ended(Tango::CmdDoneEvent *);
private:
   double data;
};

void MyCallBack::cmd_ended(Tango CmdDoneEvent *cmd)
{
   if (cmd->err == true)
      Tango::Except::print_error_stack(cmd->errors);
   else
   {
      short cmd_result;
      cmd->argout >> cmd_result;
      cout << "Command result = " << cmd_result << endl;
      cout << "Callback personal data = " << data << endl;
   }
}

int main(int argc, char *argv[])
{
   ....
   ....
   Tango::DeviceProxy dev("...");
   double my_data = ...;
   MyCallBack cb(my_data);
   dev.command_inout_asynch("MyCmd",cb);
   ...
   ...
   ...
   dev.get_asynch_replies();
   ...
```

```
        ...
}
```

Exception: None, all errors are reported using the err and errors fields of the parameter passed to the callback method. See chapter 6.8 for details.

6.1.7.3 void DeviceProxy::get_asynch_replies(long timeout)

Fire callback methods for device asynchronous requests (command and attributes) with already arrived replied. Wait and block the caller for timeout milliseconds if they are some device asynchronous requests which are not yet arrived. Returns immediately if there is no asynchronous request for the device. If timeout is set to 0, the call waits until all the asynchronous requests sent to the device has received a reply.

Exception: AsynReplyNotArrived. All other errors are reported using the err and errors fields of the object passed to the callback methods. See chapter 6.8 for details.

6.1.7.4 void DeviceProxy::cancel_asynch_request(long id)

Cancel a pending asynchronous request. id is the asynchronous call identifier. This is a call local to the client. It simply allows the caller not to get the answer of the asynchronous request. It does not interrupt the call execution on the remote device.

Exception: AsynCall

6.1.7.5 void DeviceProxy::cancel_all_polling_asynch_request()

Cancel all pending polling asynchronous requests. This is a call local to the client. It simply allows the caller not to get the answers of the asynchronous requests. It does not interrupt the call execution on the remote devices.

6.1.8 Polling related methods

6.1.8.1 bool DeviceProxy::is_command_polled(string &cmd_name)

Returns true if the command "cmd_name" is polled. Otherwise, returns false.

6.1.8.2 bool DeviceProxy::is_command_polled(const char *cmd_name)

Idem previous call

6.1.8.3 bool DeviceProxy::is_attribute_polled(string &attr_name)

Returns true if the attribute "attr_name" is polled. Otherwise, returns false.

6.1.8.4 bool Deviceproxy::is_attribute_polled(const char *attr_name)

Idem previous call

6.1.8.5 int DeviceProxy::get_command_poll_period(string &cmd_name)

Returns the command "cmd_name" polling period in mS. If the command is not polled, it returns 0.

6.1.8.6 int DeviceProxy::get_command_poll_period(const char *cmd_name)

Idem previous call

6.1.8.7 int DeviceProxy::get_attribute_poll_period(string &attr_name)

Returns the attribute "attr_name" polling period in mS. If the attribute is not polled, it returns 0.

6.1.8.8 int Deviceproxy::get_attribute_poll_period(const char *attr_name)

Idem previous call

6.1.8.9 vector<string> *DeviceProxy::polling_status()

Returns the device polling status. There is one string for each polled command/attribute. Each string is multi-line string with :

- The attribute/command name
- The attribute/command polling period (in mS)
- The attribute/command polling ring buffer depth
- The time needed for the last command/attribute execution (in mS)
- The time since data in the ring buffer has not been updated
- The delta time between the last records in the ring buffer
- The exception parameters in case of the last command/attribute execution failed

This method allocates memory for the vector of string(s) returned to the caller. It is the caller responsibility to delete this memory.

6.1.8.10 void DeviceProxy::poll_command(string &cmd_name,int period)

Add the command "cmd_name" to the list of polled command. The polling period is specified by "period" (in mS). If the command is already polled, this method will update the polling period according to "period".

6.1.8.11 void DeviceProxy::poll_command(const char *cmd_name, int period)

Idem previous call

6.1.8.12 void DeviceProxy::poll_attribute(string &attr_name, int period)

Add the attribute "attr_name" to the list of polled attributes. The polling period is specified by "period" (in mS). If the attribute is already polled, this method will update the polling period according to "period".

6.1.8.13 void DeviceProxy::poll_attribute(const char *attr_name, int period)

Idem previous call

6.1.8.14 void DeviceProxy::stop_poll_command(string &cmd_name)

Remove command "cmd_name" from the list of polled command.

6.1.8.15 void DeviceProxy::stop_poll_command(const char *cmd_name)

Idem previous call

6.1.8.16 void DeviceProxy::stop_poll_attribute(string &attr_name)

Remove attribute "attr_name" from the list of polled attributes.

6.1.8.17 void DeviceProxy::stop_poll_attribute(const char *attr_name)

Idem previous call

6.1.9 Event related methods

6.1.9.1 int DeviceProxy::subscribe_event(const string &attribute, EventType event, CallBack *cb)

The client call to subscribe for event reception in the **push model**. The client implements a callback method which is triggered when the event is received. Filtering is done based on the event type. For example when reading the state and the reason specified is "change" the event will be fired only when the state changes. Events consist of an attribute name and the event reason. A standard set of reasons are implemented by the system, additional device specific reasons can be implemented by device servers programmers.

The *attribute* parameter is the device attribute name which will be sent as an event e.g. “current”, *event* parameter is the event reason and must be on the enumerated values:

- Tango::CHANGE_EVENT
- Tango::PERIODIC_EVENT
- Tango::ARCHIVE_EVENT
- Tango::ATTR_CONF_EVENT
- Tango::DATA_READY_EVENT
- Tango::USER_EVENT

cb is a pointer to a class inheriting from the Tango CallBack class and implementing a *push_event()* method.

The *subscribe_event()* call returns an event id which has to be specified when unsubscribing from this event. Please, note that the *cb* parameter is a pointer. The lifetime of the pointed to object must at least be equal to the time when events are requested because only the pointer is stored into the event machinery. The same thing is true for the DeviceProxy instance on which the *subscribe_event()* method is called.

Exception: EventSystemFailed

Note: For releases prior to Tango 8, a similar call with a forth argument (const vector<string> &filters) was available. This extra argument gave the user a way to define extra event filtering. For compatibility reason, this call still exist but the extra filtering features is not implemented.

6.1.9.2 int DeviceProxy::subscribe_event(const string &attribute, EventType event, CallBack *cb, bool stateless)

This subscribe event method has the same functionality as described in the last section. It adds an additional flag called *stateless.* When the *stateless* flag is set to *false*, an exception will be thrown when the event subscription encounters a problem.

With the *stateless* flag set to *true*, the event subscription will always succeed, even if the corresponding device server is not running. The keep alive thread will try every 10 seconds to subscribe for the specified event. At every subscription retry, a callback is executed which contains the corresponding exception.

Exception: EventSystemFailed

Note: For releases prior to Tango 8, a similar call with a fifth argument (const vector<string> &filters) was available. This extra argument gave the user a way to define extra event filtering and it was the forth argument in the argument list. For compatibility reason, this call still exist but the extra filtering features is not implemented.

6.1.9.3 int DeviceProxy::subscribe_event(const string &attribute, EventType event, int event_queue_size, bool stateless)

The client call to subscribe for event reception in the **pull model**. Instead of a callback method the client has to specify the size of the event reception buffer.

The event reception buffer is implemented as a round robin buffer. This way the client can set-up different ways to receive events.

- Event reception buffer size = 1 : The client is interested only in the value of the last event received. All other events that have been received since the last reading are discarded.
- Event reception buffer size > 1 : The client has chosen to keep an event history of a given size. When more events arrive since the last reading, older events will be discarded.
- Event reception buffer size = ALL_EVENTS : The client buffers all received events. The buffer size is unlimited and only restricted by the available memory for the client.

All other parameters are similar to the descriptions given in the last two sections.

Exception: EventSystemFailed

Note: For releases prior to Tango 8, a similar call with a fifth argument (const vector<string> &filters) was available. This extra argument gave the user a way to define extra event filtering and it was the forth argument in the argument list. For compatibility reason, this call still exist but the extra filtering features is not implemented.

6.1.9.4 void DeviceProxy::unsubscribe_event(int event_id)

Unsubscribe a client from receiving the event specified by *event_id*. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.5 void DeviceProxy::get_events(int event_id, CallBack *cb)

The method extracts all waiting events from the event reception buffer and executes the callback method *cb* for every event. During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.6 void DeviceProxy::get_events(int event_id, EventDataList &event_list)

The method extracts all waiting events from the event reception buffer. The returned *event_list* is a vector of EventData pointers. The EventData object contains the event information as for the callback methods.

During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.7 void DeviceProxy::get_events(int event_id, AttrConfEventDataList &event_list)

The method extracts all waiting attribute configuration events from the event reception buffer. The returned *event_list* is a vector of AttrConfEventData pointers. The AttrConfEventData object contains the event information as for the callback methods.

During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.8 void DeviceProxy::get_events(int event_id, DataReadyEventDataList &event_list)

The method extracts all waiting attribute configuration events from the event reception buffer. The returned *event_list* is a vector of DataReadyEventData pointers. The DataReadyEventData object contains the event information as for the callback methods.

During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.9 int DeviceProxy::event_queue_size(int event_id)

Returns the number of stored events in the event reception buffer. After every call to *DeviceProxy:get_events()*, the event queue size is 0.

During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.10 TimeVal DeviceProxy::get_last_event_date(int event_id)

Returns the arrival time of the last event stored in the event reception buffer. After every call to *DeviceProxy:get_events()*, the event reception buffer is empty. In this case an exception will be returned.

During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.9.11 bool DeviceProxy::is_event_queue_empty(int event_id)

Returns true when the event reception buffer is empty.

During event subscription the client must have chosen the **pull model** for this event. *event_id* is the event identifier returned by the *DeviceProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.1.10 Property related methods

6.1.10.1 void DeviceProxy::get_property (string&, DbData&)

Get a single property for a device. The property to get is specified as a string. Refer to DbDevice::get_property() and DbData sections below for details on the DbData type.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.2 void DeviceProxy::get_property (vector<string>&, DbData&)

Get a list of properties for a device. The properties to get are specified as a vector of strings. Refer to DbDevice::get_property() and DbData sections below for details on the DbData type.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.3 void DeviceProxy::get_property(DbData&)

Get property(ies) for a device. Properties to get are specified using the DbData type. Refer to DbDevice::get_property() and DbData sections below for details.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.4 void DeviceProxy::put_property(DbData&)

Put property(ies) for a device. Properties to put are specified using the DbData type. Refer to DbDevice::put_property() and DbData sections below for details.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.5 void DeviceProxy::delete_property (string&)

Delete a single property for a device. The property to delete is specified as a string.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.6 void DeviceProxy::delete_property (vector<string>&)

Delete a list of properties for a device. The properties to delete are specified as a vector of strings.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.7 void DeviceProxy::delete_property(DbData&)

Delete property(ies) for a device. Properties to delete are specified using the DbData type. Refer to DbDevice::get_property() and DbData sections below for details.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.10.8 void DeviceProxy::get_property_list(const string &filter,vector<string> &prop_list)

Get the list of property names for the device. The parameter *filter* allows the user to filter the returned name list. The wildcard character is '*'. Only one wildcard character is allowed in the filter parameter. The name list is returned in the vector of strings passed as the method second argument.

Exception: NonDbDevice, WrongNameSyntax, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.1.11 Logging related methods

6.1.11.1 void DeviceProxy::add_logging_target(const string &target_type_target_name)

Adds a new logging target to the device. The target_type_target_name input parameter must follow the format: target_type::target_name. Supported target types are: *console*, *file* and *device*. For a device target, the target_name part of the target_type_target_name parameter must contain the name of a log consumer device (as defined in A.8). For a file target, target_name is the full path to the file to log to. If omitted, the device's name is used to build the file name (which is something like domain_family_member.log). Finally, the target_name part of the target_type_target_name input parameter is ignored in case of a console target and can be omitted.

Exception: DevFailed from device

6.1.11.2 void DeviceProxy::add_logging_target (const char *target_type_target_name)

Idem previous call

6.1.11.3 void DeviceProxy::remove_logging_target(const string &target_type_target_name)

Removes a logging target from the device's target list. The target_type_target_name input parameter must follow the format: target_type::target_name. Supported target types are: *console*, *file* and *device*. For a device target, the target_name part of the target_type_target_name parameter must contain the name of a log consumer device (as defined in). For a file target, target_name is the full path to the file to remove. If omitted, the default log file is removed. Finally, the target_name part of the target_type_target_name input parameter is ignored in case of a console target and can be omitted.

If target_name is set to "*", all targets of the specified target_type are removed.

6.1.11.4 void DeviceProxy::remove_logging_target (const char *target_type_target_name)

Idem previous call

6.1.11.5 vector<string> DeviceProxy::get_logging_target ()

Returns a vector of string containing the current device's logging targets. Each vector element has the following format: target_type::target_name. An empty vector is returned is the device has no logging targets.

6.1.11.6 int DeviceProxy::get_logging_level ()

Returns the current device's logging level (0=OFF, 1=FATAL, 2=ERROR, 3=WARNING, 4=INFO, 5=DEBUG).

6.1.11.7 void DeviceProxy::set_logging_level (int level)

Changes the device's logging level. (0=OFF, 1=FATAL, 2=ERROR, 3=WARNING, 4=INFO, 5=DEBUG).

6.1.12 Locking related methods

6.1.12.1 void DeviceProxy::lock(int lock_validity = 10)

Lock a device. The lock_validity is the time (in seconds) the lock is kept valid after the previous lock call. A default value of 10 seconds is provided and should be fine in most cases. In case it is necessary to change the lock validity, it's not possible to ask for a validity less than a minimum value set to 2 seconds. The library provided an automatic system to periodically re lock the device until an unlock call. No code is needed to start/stop this automatic re-locking system. The locking system is re-entrant. It is then allowed to call this method on a device already locked by the same process. The locking system has the following features:

- It is impossible to lock the database device or any device server process admin device
- Destroying a locked DeviceProxy unlocks the device
- Restarting a locked device keeps the lock
- It is impossible to restart a device locked by someone else
- Restarting a server breaks the lock

A locked device is protected against the following calls when executed by another client:

- *command_inout* call except for device state and status requested via command and for the set of commands defined as allowed following the definition of allowed command in the Tango control access schema.
- *write_attribute* call

- *write_read_attribute* call
- *set_attribute_config* call

6.1.12.2 void DeviceProxy::unlock(bool force = false)

Unlock a device. If used, the method argument provides a back door on the locking system. If this argument is set to true, the device will be unlocked even if the caller is not the locker. This feature is provided for administration purpopse and should be used very carefully. If this feature is used, the locker will receive a *DeviceUnlocked* during the next call which is normally protected by the locking Tango system.

6.1.12.3 string DeviceProxy::locking_status()

This method returns a plain string describing the device locking status. This string can be:

- "Device <device name> is not locked" in case the device is not locked
- "Device <device name> is locked by CPP or Python client with PID <pid> from host <host name>" in case the device is locked by a CPP client
- "Device <device name> is locked by JAVA client class <main class> from host <host name>" in case the device is locked by a JAVA client

6.1.12.4 bool DeviceProxy::is_locked()

Returns true if the device is locked. Otherwise, returns false.

6.1.12.5 bool DeviceProxy::is_locked_by_me()

Returns true if the device is locked by the caller. Otherwise, returns false (device not locked or locked by someone else)

6.1.12.6 bool DeviceProxy::get_locker(LockerInfo &li)

If the device is locked, this method returns true an set some locker process informations in the structure passed as argument. If the device is not locked, the method returns false. The LockerInfo structure definition is

---

```
typedef union
{
   pid_t           LockerPid;
   unsigned long   UUID[4];
}LockerId;

enum LockerLanguage
{
   CPP,
   JAVA
};

struct LockerInfo
{
   LockerLanguage  ll;
   LockerId        li;
   string          locker_host;
```

```
        string          locker_class;
    };
```

The structure *ll* field is set to either CPP or JAVA depending on the locker process language. In case of CPP client, the *li* union is set to the locker process pid (*LockerPid* field). In case of Java client, it is set to the Java client UUID (Universal Uniq IDentifier) in the *UUID* field. The *locker_host* field is initialised with the host name where the locker process is running (or its IP adress as a string if it is not possiblr to get the name associated with this address). The *locker_class* field is set to the Java virtual machine main class name when the locker client process is written in Java. For CPP client, it is set to the string "Not defined".

6.2 Tango::DeviceData

This is the fundamental type for sending and receiving data from device commands. The values can be inserted and extracted using the operators << and >> respectively and insert() for mixed data types. A status flag indicates if there is data in the DbDatum object or not. An additional flag allows the user to activate exceptions.

6.2.1 Constructors, assignement operators and C++11

This class has a default constructor (*DeviceData()*), a copy constructor (*DeviceData(const DeviceData &)*) and one assignement operator (*DeviceData & operator=(const DeviceData &)*). Nevertheless, the assignement operator and the copy constructor does not really copy the data included in the instance. For efficiency reasons, they rather move the data from one instance to another. Starting with Tango 8 and if Tango is compiled with gcc release 4.3 or above (or Windows VC 10 and above), some move semantics from C++11 has been added. This means that:

- A move constructor (*DeviceData(DeviceData &&)*) has been added.
- A move assignement operator (*DeviceData & operator=(DeviceData &&)*) has been added.
- The classical copy constructor and assignement operator really copy the data.

6.2.2 Operators

The insert and extract operators are specified for the following C++ types :

1. bool
2. short
3. unsigned short
4. DevLong
5. DevULong
6. DevLong64
7. DevULong64
8. float
9. double
10. string
11. char* (insert only)

12. const char *
13. vector<unsigned char>
14. vector<string>
15. vector<short>
16. vector<unsigned short>
17. vector<DevLong>
18. vector<DevULong>
19. vector<DevLong64>
20. vector<DevULong64>
21. vector<float>
22. vector<double>

Operators exist for inserting and extracting the native TANGO CORBA sequence types. These can be useful for programmers who want to use the TANGO api internally in their device servers and do not want to convert from CORBA to C++ types. Insert and extract operators exist for the following types :

1. DevVarUCharArray * (const DevVarUCharArray * for extraction)
2. DevVarShortArray * (const DevVarShortArray * for extraction)
3. DevVarUShortArray * (const DevVarUShortArray * for extraction)
4. DevVarLongArray * (const DevVarLongArray * for extraction)
5. DevVarULongArray * (const DevVarULongArray * for extraction)
6. DevVarLong64Array * (const DevVarLong64Array * for extraction)
7. DevVarULong64Array * (const DevVarULong64Array * for extraction)
8. DevVarFloatArray * (const DevVarFloatArray * for extraction)
9. DevVarDoubleArray * (const DevVarDoubleArray * for extraction)
10. DevVarStringArray * (const DevVarStringArray * for extraction)
11. DevVarLongStringArray * (const DevVarLongStringArray * for extraction)
12. DevVarDoubleStringArray * (const DevVarDoubleStringArray * for extraction)

Note :
Insertion by pointers takes full ownership of the pointed to memory. The insertion copy the data in the DeviceData object and delete the pointed to memory. Therefore, the memory is not more usable after the insertion. Also note that when using extraction by pointers, the pointed to memory is inside the DeviceData object and its lifetime is the same than the DeviceData object lifetime.

Operators also exist for inserting TANGO CORBA sequence type by reference. The insertion copy the data into the DeviceData object. Insert operator exist for the following types :

1. DevVarUCharArray &
2. DevVarShortArray &
3. DevVarUShortArray &
4. DevVarLongArray &
5. DevVarULongArray &
6. DevVarLong64Array &
7. DevVarULong64Array &
8. DevVarFloatArray &
9. DevVarDoubleArray &
10. DevVarStringArray &
11. DevVarLongStringArray &
12. DevVarDoubleStringArray &

Additional methods exist for inserting a mixture of strings and long (Tango::DevVarLongStringArray) and string and doubles (Tango::DevVarDoubleStringArray). These are :

1. insert(vector<long>&, vector<string>&)
2. insert(vector<double>&, vector<string>&)
3. extract(vector<long>&, vector<string>&)
4. extract(vector<double>&, vector<string>&)

All the extraction methods returns a boolean set to false if the extraction has failed (empty DeviceData, wrong data type...)
Special care has been taken to avoid memory copy between the network layer and the user application. Nevertheless, C++ vector types are not the CORBA native type and one copy is unavoidable when using vectors. Using the native TANGO CORBA sequence types avoid any copy. When using these TANGO CORBA sequence types, insertion into the DeviceData object consumes the memory pointed to by the pointer. After the insertion, it is not necessary to delete the memory. It will be done by the destruction of the DeviceData object. For extraction, the pointer used for the extraction points into memory inside the DeviceData object and you should not delete it
Here is an example of creating, inserting and extracting some data type from/into DeviceData object :

```
DeviceData my_short, my_long, my_string;
DeviceData my_float_vector, my_double_vector;
string a_string;
short a_short;
DevLong a_long;
vector<float> a_float_vector;
vector<double> a_double_vector;
my_short << 100; // insert a short
my_short >> a_short; // extract a short
my_long << 1000; // insert a long
my_long >> a_long; // extract a long
my_string << string(“estas lista a bailar el tango ?”); // insert a string
```

```
my_string >> a_string; // extract a string
my_float_vector << a_float_vector // insert a vector of floats
my_float_vector >> a_float_vector; // extract a vector of floats
my_double_vector << a_double_vector; // insert a vector of doubles
my_double_vector >> a_double_vector; // extract a vector of doubles
//
// Example of memory management with TANGO sequence types without memory leaks
//
for (int i = 0;i < 10;i++)
{
    DeviceData din,dout;
    DevVarLongArray *in = new DevVarLongArray();
    in->length(2);
    (*in)[0] = 2;
    (*in)[1] = 4;
    din << in;
    try
    {
        dout = device->command_inout("Cmd",din);
    }
    catch(DevFailed &e)
    {
       ....
    }
    const DevVarLongArray *out;
    dout >> out;
    cout << "Received value = " << (*out)[0];
}
```

Exception: WrongData if requested

6.2.3 bool DeviceData::is_empty()

is_empty() is a boolean method which returns true or false depending on whether the DeviceData object contains data or not. It can be used to test whether the DeviceData has been initialized or not e.g.

```
string string_read;
DeviceData sl_read = my_device->command_inout("ReadLine");
if (! sl_read.is_empty())
{
    sl_read >> string_read;
}
else
{
    cout << " no data read from serial line !" << endl;
}
```

Exception: WrongData if requested

6.2.4 int DeviceData::get_type()

This method returns the Tango data type of the data inside the DeviceData object

6.2.5 void DeviceData::exceptions(bitset<DeviceData::numFlags>)

Is a method which allows the user to switch on/off exception throwing when trying to extract data from an empty DeviceData object or using a wrong data type. The default is to not throw exception. The following flags are supported :

1. **isempty_flag** - throw a WrongData exception (reason = API_EmptyDeviceData) if user tries to extract data from an empty DeviceData object
2. **wrongtype_flag** - throw a WrongData exception (reason = API_IncompatibleCmdArgumentType) if user tries to extract data with a type different than the type used for insertion

6.2.6 bitset<DeviceData::numFlags> exceptions()

Returns the whole exception flags.

6.2.7 void DeviceData::reset_exceptions(DeviceData::except_flags fl)

Resets one exception flag

6.2.8 void DeviceData::set_exceptions(DeviceData::except_flags fl)

Sets one exception flag

The following is an example of how to use these exceptions related methods

```
DeviceData da;

bitset<DeviceData::numFlags> bs = da.exceptions();
cout << "bs = " << bs << endl;

da.set_exceptions(DeviceData::wrongtype_flag);
bs = da.exceptions();

cout << "bs = " << bs << endl;
```

6.2.9 ostream &operator<<(ostream &, DeviceData &)

Is an utility function to easily print the contents of a DeviceData object. This function knows all types which could be inserted in a DeviceData object and print them accordingly. A special string is printed if the DeviceData object is empty

```
DeviceProxy *dev = new DeviceProxy(“...”);
DeviceData out;
out = dev->command_inout(“MyCommand”);
cout << “Command returned: ” << out << endl;
```

6.3 Tango::DeviceDataHistory

This is the fundamental type for receiving data from device command polling buffers. This class inherits from the Tango::DeviceData class. One instance of this class is created for each command result history. Within this class, you find the command result data or the exception parameters, a flag indicating if the command has failed when it was invoked by the device server polling thread and the date when the command was executed. For history calls, it is not possible to returns command error as exception. See chapter on Advanced Features for all details regarding device polling. On top of the methods inherited from the DeviceData class, it offers the following methods

6.3.1 bool DeviceDataHistory::has_failed()

Returns true if the corresponding command has failed when it was executed by the device server polling thread. Otherwise returns false (amazing!)

Exception: none

6.3.2 TimeVal &DeviceDataHistory::get_date()

Returns the date when the device server polling thread has executed the command.

Exception: none

6.3.3 const DevErrorList &DeviceDataHistory::get_err_stack()

Return the error stack recorded by the device server polling thread in case of the command failed when it was invoked.

Exception: none

6.3.4 ostream &operator<<(ostream &, DeviceDataHistory &)

Is an utility function to easily print the contents of a DeviceDataHistory object. This function knows all types which could be inserted in a DeviceDataHistory object and print them accordingly. It also prints date and error stack in case the command returned an error.

```
DeviceProxy *dev = new DeviceProxy("...");
int hist_depth = 4;
vector<DeviceDataHistory> *hist;
hist = dev->command_history("MyCommand",hist_depth);
for (int i = 0;i < hist_depth;i++)
{
    cout << (*hist)[i] << endl;
}
delete hist;
```

6.4 Tango::DeviceAttribute

This is the fundamental type for sending and receiving data to and from device attributes. The values can be inserted and extracted using the operators << and >> respectively and insert() for mixed data types. There are two ways to check if the extraction operator succeed :

1. By testing the extractor operators return value. All the extractors operator returns a boolean value set to false in case of problem.

2. By asking the DeviceAttribute object to throw exception in case of problem. By default, DeviceAttribute throws exception :
   (a) when the user try to extract data and the server reported an error when the attribute was read.
   (b) When the user try to extract data from an empty DeviceAttribute

6.4.1 Constructors, assignement operators

Many constructors have been written for this class. The following constructors exist :

1. The C++ basic constructors
   (a) DeviceAttribute();
   (b) DeviceAttribute(const DeviceAttribute&);
2. Constructors for scalar type with name as C++ string or "const char *"
   (a) DeviceAttribute(string &, bool);
   (b) DeviceAttribute(string &, short);
   (c) DeviceAttribute(string &, DevLong);
   (d) DeviceAttribute(string &, DevLong64);
   (e) DeviceAttribute(string &, float);
   (f) DeviceAttribute(string &, double);
   (g) DeviceAttribute(string &, unsigned char);
   (h) DeviceAttribute(string &, unsigned short);
   (i) DeviceAttribute(string &, DevULong);
   (j) DeviceAttribute(string &, DevULong64);
   (k) DeviceAttribute(string &, string &);
   (l) DeviceAttribute(string &, DevState);
   (m) DeviceAttribute(string &, DevEncoded &);
   (n) DeviceAttribute(const char *, bool);
   (o) DeviceAttribute(const char *, short);
   (p) DeviceAttribute(const char *, DevLong);
   (q) DeviceAttribute(const char *, DevLong64);
   (r) DeviceAttribute(const char *, float);
   (s) DeviceAttribute(const char *, double);
   (t) DeviceAttribute(const char *, unsigned char);
   (u) DeviceAttribute(const char *, unsigned short);
   (v) DeviceAttribute(const char *, DevULong);
   (w) DeviceAttribute(const char *, DevULong64);
   (x) DeviceAttribute(const char *, string &);
   (y) DeviceAttribute(const char *, DevState);
   (z) DeviceAttribute(const char *,DevEncoded &);
3. Constructors for C++ vector types (for spectrum attribute) with name as C++ string or "const char *"
   (a) DeviceAttribute(string &, vector<bool> &);

(b) DeviceAttribute(string &, vector<short> &);
(c) DeviceAttribute(string &, vector<DevLong> &);
(d) DeviceAttribute(string &, vector<DevLong64> &);
(e) DeviceAttribute(string &, vector<float> &);
(f) DeviceAttribute(string &, vector<double> &);
(g) DeviceAttribute(string &, vector<unsigned char> &);
(h) DeviceAttribute(string &, vector<unsigned short> &);
(i) DeviceAttribute(string &, vector<DevULong> &);
(j) DeviceAttribute(string &, vector<DevULong64> &);
(k) DeviceAttribute(string &, vector<string> &);
(l) DeviceAttribute(string &, vector<DevState> &);
(m) DeviceAttribute(const char *, vector<bool> &);
(n) DeviceAttribute(const char *, vector<short> &);
(o) DeviceAttribute(const char *, vector<DevLong> &);
(p) DeviceAttribute(const char *, vector<DevLong64> &);
(q) DeviceAttribute(const char *, vector<float> &);
(r) DeviceAttribute(const char *, vector<double> &);
(s) DeviceAttribute(const char *, vector<unsigned char> &);
(t) DeviceAttribute(const char *, vector<unsigned short> &);
(u) DeviceAttribute(const char *, vector<DevULong> &);
(v) DeviceAttribute(const char *, vector<DevULong64> &);
(w) DeviceAttribute(const char *, vector<string> &);
(x) DeviceAttribute(const char *, vector<DevState> &);

4. Constructors for C++ vector types (for image attribute) with name as C++ string or "const char *". These constructors have two more parameters allowing the user to define the x and y image dimensions.

(a) DeviceAttribute(string &, vector<bool> &, int, int);
(b) DeviceAttribute(string &, vector<short> &, int, int);
(c) DeviceAttribute(string &, vector<DevLong> &, int, int);
(d) DeviceAttribute(string &, vector<DevLong64> &, int, int);
(e) DeviceAttribute(string &, vector<float> &, int, int);
(f) DeviceAttribute(string &, vector<double> &, int, int);
(g) DeviceAttribute(string &, vector<unsigned char> &, int, int);
(h) DeviceAttribute(string &, vector<unsigned short> &, int, int);
(i) DeviceAttribute(string &, vector<DevULong> &, int, int);
(j) DeviceAttribute(string &, vector<DevULong64> &, int, int);
(k) DeviceAttribute(string &, vector<string> &, int, int);
(l) DeviceAttribute(string &, vector<DevState> &, int, int);
(m) DeviceAttribute(const char *, vector<bool> &, int, int);
(n) DeviceAttribute(const char *, vector<short> &, int, int);
(o) DeviceAttribute(const char *, vector<DevLong> &, int, int);

(p) DeviceAttribute(const char *, vector<DevLong64> &, int, int);
(q) DeviceAttribute(const char *, vector<float> &, int, int);
(r) DeviceAttribute(const char *, vector<double> &, int, int);
(s) DeviceAttribute(const char *, vector<unsigned char> &, int, int);
(t) DeviceAttribute(const char *, vector<unsigned short> &, int, int);
(u) DeviceAttribute(const char *, vector<DevULong> &, int, int);
(v) DeviceAttribute(const char *, vector<DevULong64> &, int, int);
(w) DeviceAttribute(const char *, vector<string> & , int, int);
(x) DeviceAttribute(const char *, vector<DevState) &, int, int);

Note that the assignement operator and the copy constructor does not really copy the data included in the instance. For efficiency reasons, they rather move the data from one instance to another. Starting with Tango 8 and if Tango is compiled with gcc release 4.3 or above (or Windows VC 10 and above), some move semantics from C++11 has been added. This means that:

- A move constructor (*DeviceAttribute(DeviceAttribute &&)*) has been added.
- A move assignement operator (*DeviceAttribute & operator=(DeviceAttribute &&)*) has been added.
- The classical copy constructor and assignement operator really copy the data.

6.4.2 Data Extraction and Insertion : Operators and Methods

Special care has been taken to avoid memory copy between the network layer and the user application. Nevertheless, C++ vector types are not the CORBA native type and one copy is unavoidable when using vectors. Using the native TANGO CORBA sequence types in most cases avoid any copy but needs some more care about memory usage.

- **For insertion into DeviceAttribute instance from TANGO CORBA sequence pointers, the DeviceAttribute object takes ownership of the pointed to memory. This means that the pointed to memory will be freed when the DeviceAttribute object is destroyed or when another data is inserted into it.**
- **The insertion into DeviceAttribute instance from TANGO CORBA sequence reference copy the data into the DeviceAttribute object.**
- **For extraction into TANGO CORBA sequence types, the extraction method consumes the memory allocated to store the data and it is the caller responsibility to delete this memory.**

As it has been done for constructors, a lot of insertors operator for classical C++ data types have been defined :

1. Insert operators for the following scalar C++ types :

   (a) bool
   (b) short
   (c) DevLong
   (d) DevLong64
   (e) float
   (f) double
   (g) unsigned char
   (h) unsigned short

   (i) DevULong
   (j) DevULong64
   (k) string
   (l) DevState
   (m) DevEncoded
   (n) DevString
   (o) const char *

2. Insert operators for the following C++ vector types for spectrum attributes :
   (a) vector<bool>
   (b) vector<short>
   (c) vector<DevLong>
   (d) vector<DevLong64>
   (e) vector<float>
   (f) vector<double>
   (g) vector<unsigned char>
   (h) vector<unsigned short>
   (i) vector<DevULong>
   (j) vector<DevULong64>
   (k) vector<string>
   (l) vector<DevState>

3. Insert methods for the DevEncoded data type
   (a) insert(char *&, unsigned char *&, unsigned int)
       The last argument is the size of the buffer passed to the method as its second argument
   (b) insert(string &, vector<unsigned char &>)

4. Insert methods for the following C++ vector types for image attributes allowing the specification of the x and y image dimensions :
   (a) insert(vector<bool> &,int, int)
   (b) insert(vector<short> &,int, int)
   (c) insert(vector<DevLong> &,int, int)
   (d) insert(vector<DevLong64> &,int, int)
   (e) insert(vector<float> &,int, int)
   (f) insert(vector<double> &,int, int)
   (g) insert(vector<unsigned char> &,int, int)
   (h) insert(vector<unsigned short> &,int, int)
   (i) insert(vector<DevULong> &,int, int)
   (j) insert(vector<DevULong64> &,int, int)
   (k) insert(vector<string> &,int, int)
   (l) insert(vector<DevState> &,int, int)

Extractor operators are specified for the following C++ basic types

1. Extract operators for the following scalar C++ types :
   (a) bool
   (b) short
   (c) DevLong
   (d) DevLong64
   (e) float
   (f) double
   (g) unsigned char
   (h) unsigned short
   (i) DevULong
   (j) DevULong64
   (k) string
   (l) Tango::DevState
   (m) Tango::DevEncoded
2. Extract operators for the following C++ vector types for spectrum and image attributes :
   (a) vector<bool>
   (b) vector<short>
   (c) vector<DevLong>
   (d) vector<DevLong64>
   (e) vector<float>
   (f) vector<double>
   (g) vector<unsigned char>
   (h) vector<unsigned short>
   (i) vector<DevULong>
   (j) vector<DevULong64>
   (k) vector<string>
   (l) vector<DevState>
3. Extract methods to extract only the read value of an attribute into a C++ vector. The dimension of the read value can be read by using the methods get_dim_x() and get_dim_y() or get_r_dimension(). The methods use the same return values as the extraction operators with exceptions triggered by the exception flags:
   (a) bool DeviceAttribute::extract_read (vector<bool>&);
   (b) bool DeviceAttribute::extract_read (vector<short>&);
   (c) bool DeviceAttribute::extract_read (vector<DevLong>&);
   (d) bool DeviceAttribute::extract_read (vector<DevLong64>&);
   (e) bool DeviceAttribute::extract_read (vector<float>&);
   (f) bool DeviceAttribute::extract_read (vector<double>&);
   (g) bool DeviceAttribute::extract_read (vector<unsigned char>&);
   (h) bool DeviceAttribute::extract_read (vector<unsigned short>&);
   (i) bool DeviceAttribute::extract_read (vector<DevULong>&);

   (j) bool DeviceAttribute::extract_read (vector<DevULong64>&);
   (k) bool DeviceAttribute::extract_read (vector<string>&);
   (l) bool DeviceAttribute::extract_read (vector<DevState>&);
   (m) bool DeviceAttribute::extract_read(string &, vector<unsigned char> &);

4. Extract methods to extract only the set value of an attribute into a C++ vector. The dimension of the set value can be read by using the methods get_written_dim_x() and get_written_dim_y() or get_w_dimension(). The methods use the same return values as the extraction operators with exceptions triggered by the exception flags:

   (a) bool DeviceAttribute::extract_set (vector<bool>&);
   (b) bool DeviceAttribute::extract_set (vector<short>&);
   (c) bool DeviceAttribute::extract_set (vector<DevLong>&);
   (d) bool DeviceAttribute::extract_set (vector<DevLong64>&);
   (e) bool DeviceAttribute::extract_set (vector<float>&);
   (f) bool DeviceAttribute::extract_set (vector<double>&);
   (g) bool DeviceAttribute::extract_set (vector<unsigned char>&);
   (h) bool DeviceAttribute::extract_set (vector<unsigned short>&);
   (i) bool DeviceAttribute::extract_set (vector<DevULong>&);
   (j) bool DeviceAttribute::extract_set (vector<DevULong64>&);
   (k) bool DeviceAttribute::extract_set (vector<string>&);
   (l) bool DeviceAttribute::extract_set (vector<DevState>&);
   (m) bool DeviceAttribute::extract_set(string &, vector<unsigned char> &);

5. Special extract method for the Tango::DevEncoded data type

   (a) bool DeviceAttribute::extract(const char *&, unsigned char *&, unsigned int &);
       The last argument is the size of the buffer passed to the method as its second argument
   (b) bool DeviceAttribute::extract(string &, vector<unsigned char> &);

Operators also exist for extracting some native TANGO CORBA sequence types. These can be useful for programmers who want to use the TANGO api internally in their device servers and do not want to convert from CORBA to C++ types.

1. Insert operators for spectrum attribute and for the following types by pointer :

   (a) DevVarBooleanArray *
   (b) DevVarShortArray *
   (c) DevVarLongArray *
   (d) DevVarLong64Array *
   (e) DevVarFloatArray *
   (f) DevVarDoubleArray *
   (g) DevVarUCharArray *
   (h) DevVarUShortArray *
   (i) DevVarULongArray *
   (j) DevVarULong64Array *
   (k) DevVarStringArray *

(l) DevVarStateArray *

2. Insert operators for spectrum attribute and for the following types by reference :

   (a) const DevVarBooleanArray &
   (b) const DevVarShortArray &
   (c) const DevVarLongArray &
   (d) const DevVarLong64Array &
   (e) const DevVarFloatArray &
   (f) const DevVarDoubleArray &
   (g) const DevVarUCharArray &
   (h) const DevVarUShortArray &
   (i) const DevVarULongArray &
   (j) const DevVarULong64Array &
   (k) const DevVarStringArray &
   (l) const DevVarStateArray &

3. Insert methods for image attribute and pointers. These method allow the programmer to define the x and y image dimensions. The following methods are defined :

   (a) insert(DevVarBooleanArray *, int , int)
   (b) insert(DevVarShortArray *, int , int)
   (c) insert(DevVarLongArray *, int , int)
   (d) insert(DevVarLong64Array *, int, int)
   (e) insert(DevVarFloatArray *, int , int)
   (f) insert(DevVarDoubleArray *, int , int)
   (g) insert(DevVarUCharArray *, int , int)
   (h) insert(DevVarUShortArray *, int , int)
   (i) insert(DevVarULongArray *, int , int)
   (j) insert(DevVarULong64Array *, int, int)
   (k) insert(DevVarStringArray *, int , int)
   (l) insert(DevVarStateArray *, int, int)

4. Insert methods for image attribute and reference. These method allow the programmer to define the x and y image dimensions. The following methods are defined :

   (a) insert(const DevVarBooleanArray &, int , int)
   (b) insert(const DevVarShortArray &, int , int)
   (c) insert(const DevVarLongArray &, int , int)
   (d) insert(const DevVarLong64Array &, int, int)
   (e) insert(const DevVarFloatArray &, int , int)
   (f) insert(const DevVarDoubleArray &, int , int)
   (g) insert(const DevVarUCharArray &, int , int)
   (h) insert(const DevVarUShortArray &, int , int)
   (i) insert(const DevVarULongArray &, int , int)

(j) insert(const DevVarULong64Array &, int, int)
(k) insert(const DevVarStringArray &, int , int)
(l) insert(const DevVarStateArray &, int, int)

5. Extract operators for the following types :

(a) DevVarBooleanArray *
(b) DevVarShortArray *
(c) DevVarLongArray *
(d) DevVarLong64Array *
(e) DevVarFloatArray *
(f) DevVarDoubleArray *
(g) DevVarUCharArray *
(h) DevVarUShortArray *
(i) DevVarULongArray *
(j) DevVarULong64Array *
(k) DevVarStringArray *
(l) DevVarStateArray *
(m) DevVarEncodedArray *

Here is an example of creating, inserting and extracting some DeviceAttribute types :

```
DeviceAttribute my_short, my_long, my_string;
DeviceAttribute my_float_vector, my_double_vector;
string a_string;
short a_short;
DevLong a_long;
vector<float> a_float_vector;
vector<double> a_double_vector;
my_short << 100; // insert a short
my_short >> a_short; // extract a short
my_long << 1000; // insert a long
my_long >> a_long; // extract a DevLong
my_string << string("estas lista a bailar el tango ?"); // insert a string
my_string >> a_string; // extract a string
my_float_vector << a_float_vector // insert a vector of floats
my_float_vector >> a_float_vector; // extract a vector of floats
my_double_vector << a_double_vector; // insert a vector of doubles
my_double_vector >> a_double_vector; // extract a vector of doubles
//
// Extract read and set value of an attribute separately
// and get their dimensions
//
vector<float> r_float_vector, w_float_vector;
my_float_vector.extract_read (r_float_vector) // extract read values
int dim_x = my_float_vector.get_dim_x();      // get x dimension
int dim_y = my_float_vector.get_dim_y();      // get y dimension
my_float_vector.extract_set  (w_float_vector) // extract set values
```

```
int w_dim_x = my_float_vector.get_written_dim_x();  // get x dimension
int W_dim_y = my_float_vector.get_written_dim_y();  // get y dimension
//
// Example of memory management with TANGO sequence types without memory leaks
//
for (int i = 0;i < 10;i++)
{
    DeviceAttribute da;
    DevVarLongArray *out;
    try
    {
        da = device->read_attribute("Attr");
        da >> out;
    }
    catch(DevFailed &e)
    {
       ....
    }
    cout << "Received value = " << (*out)[0];
    delete out;
}
```

Exception: WrongData if requested

6.4.3 bool DeviceAttribute::is_empty()

is_empty() is a boolean method which returns true or false depending on whether the DeviceAttribute object contains data or not. It can be used to test whether the DeviceAttribute has been initialized or not e.g.

```
string parity;
DeviceAttribute sl_parity = my_device->read_attribute("parity");
if (! sl_read.is_empty())
{
    sl_parity >> parity;
}
else
{
    cout << " no parity attribute defined for serial line !" << endl;
}
```

Exception: WrongData if requested

6.4.4 void DeviceAttribute::exceptions(bitset<DeviceAttribute::numFlags>)

Is a method which allows the user to switch on/off exception throwing when trying to extract data from an empty DeviceAttribute object or with a wrong data type. The following flags are supported :

1. **isempty_flag** - throw a WrongData exception (reason= API_EmptyDeviceAttribute) if user tries to extract data from an empty DeviceAttribute object. By default, this flag is set.

2. **wrongtype_flag** - throw a WrongData exception (reason = API_IncompatibleAttrArgumentType) if user tries to extract data with a type different than the type used for insertion. By default, this flag is not set.

3. **failed_flag** - throw an exception when the user try to extract data from the DeviceAttribute object and an error was reported by the server when the user try to read the attribute. The type of the exception thrown is the type of the error reported by the server. By default, this flag is set.

4. **unknown_format_flag** - throw an exception when the user try to get the attribute data format from the DeviceAttribute object when this information is not yet available. This information is available only after the *read_attribute* call has been sucessfully executed. The type of the exception thrown is WrongData exception (reason = API_EmptyDeviceAttribute). By default, this flag is not set.

6.4.5 bitset<DeviceAttribute::numFlags> exceptions()

Return the whole exception flags.

6.4.6 void DeviceAttribute::reset_exceptions(DeviceAttribute::except_flags fl)

Reset one exception flag

6.4.7 void DeviceAttribute::set_exceptions(DeviceAttribute::except_flags fl)

Set one exception flag

The following is an example of how to use these exceptions related methods

```
DeviceAttribute da;

bitset<DeviceAttribute::numFlags> bs = da.exceptions();
cout << "bs = " << bs << endl;

da.set_exceptions(DeviceAttribute::wrongtype_flag);
bs = da.exceptions();

cout << "bs = " << bs << endl;
```

6.4.8 bool DeviceAttribute::has_failed()

Returns a boolean set to true if the server report an error when the attribute was read.

6.4.9 const DevErrorList &DeviceAttribute::get_err_stack()

Returns the error stack reported by the server when the attribute was read.

The following is an example of the three available ways to get data out of a DeviceAttribute object.

```
DeviceAttribute da;
vector<short> attr_data;

try
{
```

```
    da = device->read_attribute("Attr");
    da >> attr_data;
}
catch (DevFailed &e)
{
    ....
}


------------------------------------------------------------------------

DeviceAttribute da;
vector<short> attr_data;

da.reset_exceptions(DeviceAttribute::failed_flag);

try
{
    da = device->read_attribute("Attr");
}
catch (DevFailed &e)
{
    .....
}

if (!(da >> attr_data))
{
    DevErrorList &err = da.get_err_stack();
    .....
}
else
{
    .....
}

----------------------------------------------------------------------

DeviceAttribute da;
vector<short> attr_data;

try
{
    da = device->read_attribute("Attr");
}
catch (DevFailed &e)
{
    ......
}

if (da.has_failed())
{
    DevErrorList &err = da.get_err_stack();
    ....
}
```

```
60  else
61  {
62      da >> attr_data;
63  }
```

The first way is coded between lines 1 and 13. It uses the default behaviour of the DeviceAttribute object which is to throw an exception when the user try to extract data when the server reports an error when the attribute was read. The second way is coded between line 17 and 40. The DeviceAttribute object now does not throw "failed" exception any more and the return value of the extractor operator is checked. The third way is coded between line 43 and 63. In this case, the attribute data validity is checked before trying to extract them.

6.4.10 string &DeviceAttribute::get_name()

Returns the name of the attribute

6.4.11 void DeviceAttribute::set_name(string &)

Sets attribute name

6.4.12 void DeviceAttribute::set_name(const char *)

Sets attribute name

6.4.13 AttrQuality &DeviceAttribute::get_quality()

Returns the quality of the attribute: an enumerate type which can be one of {ATTR_VALID, ATTR_INVALID, ATTR_ALARM, ATTR_CHANGING or ATTR_WARNING}.

6.4.14 int DeviceAttribute::get_dim_x()

Returns the attribute read x dimension

6.4.15 int DeviceAttribute::get_dim_y()

Returns the attribute read y dimension

6.4.16 int DeviceAttribute::get_written_dim_x()

Returns the attribute write x dimension

6.4.17 int DeviceAttribute::get_written_dim_y()

Returns the attribute write y dimension

6.4.18 AttributeDimension DeviceAttribute::get_r_dimension()

Returns the attribute read dimension

6.4.19 AttributeDimension DeviceAttribute::get_w_dimension()

Returns the attribute write dimension

6.4.20 long DeviceAttribute::get_nb_read()

Returns the number of read values

6.4.21 long DeviceAttribute::get_nb_written()

Returns the number of written values. Here is an example of these last methods usage.

```
1  DeviceAttribute da;
2  vector<short> attr_data;
3
4  try
5  {
6     da = device->read_attribute("Attr");
7     da >> attr_data;
8  }
9  catch (DevFailed &e)
10 {
11    ....
12 }
13
14 long read = da.get_nb_read();
15 long written = da.get_nb_written();
16 size_t size = attr_data.size();
17
18 for (long i = 0;i < read;i++)
19    cout << "Read value " << i+1 << " = " << attr_data[i] <<  endl;
20
21 int ind;
22 for (long j = 0; j < written;j++)
23 {
24    size == 1 ? ind = j : ind = j + read;
25    cout << "Last written value " << j+1 << " = " << attr_data[ind] << endl;
26 }
```

Line 24 is needed to cover the case of scalar write attribute where only one value is returned but both get_nb_read() and get_nb_written() methods return 1.

6.4.22 TimeVal &DeviceAttribute::get_date()

Returns a reference to the time when the attribute was read in server

6.4.23 int DeviceAttribute::get_type()

Returns the type of the attribute data.

6.4.24 AttrDataFormat DeviceAttribute::get_data_format()

Returns the attribute data format. Note that this information is valid only after the call to the device has been executed. Otherwise the FMT_UNKNOWN value of the AttrDataFormat enumeration is returned or an exception is thrown according to the object exception flags.

6.4.25 ostream &operator<<(ostream &, DeviceAttribute &)

Is an utility function to easily print the contents of a DeviceAttribute object. This function knows all types which could be inserted in a DeviceAttribute object and print them accordingly if the data are valid. It also prints the date returned within the attribute, the attribute name, the dim_x and dim_y attribute parameter and its quality factor.

```
DeviceProxy *dev = new DeviceProxy("...");
DeviceAttribute attr;
attr = dev->read_attribute("MyAttribute");
cout << "Attribute returned: " << attr << endl;
```

6.5 Tango::DeviceAttributeHistory

This is the fundamental type for receiving data from device attribute polling buffers. This class inherits from the Tango::DeviceAttribute class. One instance of this class is created for each attribute result history. Within this class, you find the attribute result data or the exception parameters and a flag indicating if the attribute has failed when it was invoked by the device server polling thread. For history calls, it is not possible to returns attribute error as exception. See chapter on Advanced Features for all details regarding device polling. On top of the methods inherited from the DeviceAttribute class, it offers the following methods

6.5.1 ostream &operator<<(ostream &, DeviceAttributeHistory &)

Is an utility function to easily print the contents of a DeviceAttributeHistory object. This function knows all types which could be inserted in a DeviceAttributeHistory object and print them accordingly. It also prints date, attribute name, attribute dim_x and dim_y parameters, attribute quality factor and error stack in case the attribute returned an error.

```
DeviceProxy *dev = new DeviceProxy("...");
int hist_depth = 4;
vector<DeviceAttributeHistory> *hist;
hist = dev->attribute_history("MyAttribute",hist_depth);
for (int i = 0;i < hist_depth;i++)
{
    cout << (*hist)[i] << endl;
}
delete hist;
```

6.6 Tango::AttributeProxy()

The high level object which provides the client with an easy-to-use interface to TANGO device attributes. AttributeProxy is a handle to the real Attribute (hence the name Proxy) and is not the real Attribute (of course). The AttributeProxy manages timeouts, stateless connections (new AttributeProxy() nearly always works), and reconnection if the device server is restarted.

6.6.1 Constructors

6.6.1.1 AttributeProxy::AttributeProxy(string &name)

Create an AttributeProxy to an attribute of the specified name. The constructor will connect to the TANGO database, query for the device to which the attribute belongs to network address and build a connection to this device. If the device to which the attribute belongs to is defined in the TANGO database but the device server is not running, AttributeProxy will try to build a connection every time the client tries to access the attribute. If an alias name is defined for the attribute, this alias name can be used to create the AttributeProxy instance. If a device name alias is defined for the device, it can be used instead of the three fields device name. If the device to which the attribute belongs to is not defined in the database, an exception is thrown. Examples :

```
AttributeProxy *my_attr = new AttributeProxy("my/own/device/attr");
AttributeProxy *my_attr_bis = new AttributeProxy("attr_alias");
AttributeProxy *my_attr_ter = new AttributeProxy("dev_alias/attr");
```

See appendix on device/attribute naming for all details about Tango device or attribute naming syntax.
Exception: WrongNameSyntax, ConnectionFailed

6.6.1.2 AttributeProxy::AttributeProxy(const char *name)

Idem previous call

6.6.2 Miscellaneous methods

6.6.2.1 DevState AttributeProxy::state()

A method which returns the state of the device to which the attribute belongs to. This state is returned as a Tango::DevState type. Example :

```
dev_state = my_attr->state() << endl;
```

Exception: ConnectionFailed, CommunicationFailed

6.6.2.2 string AttributeProxy::status()

A method which return the status of the device to which the attribute belongs to. The status is returned as a string. Example :

```
cout << "device status" << my_attr->status() << endl;
```

Exception: ConnectionFailed, CommunicationFailed

6.6.2.3 int AttributeProxy::ping()

A method which sends a ping to the device to which the attribute belongs and returns the time elapsed in microseconds. Example :

```
cout << "device ping took " << my_device->ping() << " microseconds" << endl;
```

Exception: ConnectionFailed, CommunicationFailed

6.6.2.4 string AttributeProxy::name()

Returns the attribute name

6.6.2.5 DeviceProxy *get_device_proxy()

Returns the DeviceProxy instance used to communicate with the device to which the attributes belongs.

6.6.3 Synchronous related methods

6.6.3.1 AttributeInfo AttributeProxy::get_config()

Return the attribute configuration. AttributeInfo is a struct defined as follows :

```
typedef struct _AttributeInfo {
   string name;
   AttrWriteType writable;
   AttrDataFormat data_format;
   int data_type;
   int max_dim_x;
   int max_dim_y;
   string description;
   string label;
   string unit;
   string standard_unit;
   string display_unit;
   string format;
   string min_value;
   string max_value;
   string min_alarm;
   string max_alarm;
   string writable_attr_name;
   vector<string> extensions;
   Tango::DispLevel disp_level;
} AttributeInfo;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.6.3.2 void AttributeProxy::set_config(AttributeInfo &)

Change the attribute configuration.

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.6.3.3 DeviceAttribute AttributeProxy::read()

Read the attribute. To extract the value you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double (this will return 0) you have to extract it as a short.

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.6.3.4 void AttributeProxy::write(DeviceAttribute&)

Write the attribute. To insert the value to write you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short.

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.6.3.5 DeviceAttribute AttributeProxy::write_read(DeviceAttribute&)

Write then read a single attribute in a single network call. By default (serialisation by device), the execution of this call in the server can't be interrupted by other clients. To insert/extract the value to write/read you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short.

Exception: ConnectionFailed, CommunicationFailed, DeviceUnlocked, DevFailed from device

6.6.3.6 vector<DeviceAttributeHistory> *AttributeProxy::history(int)

Retrieve attribute history from the attribute polling buffer. The argument is the wanted history depth. This method returns a vector of DeviceAttributeHistory types. This method allocates memory for the vector of DeviceAttributeHistory returned to the caller. It is the caller responsibility to delete this memory. Class DeviceAttributeHistory is detailed on chapter 6.5See chapter on Advanced Feature for all details regarding polling.

```
AttributeProxy attr = new AttributeProxy("my/own/device/Current");
vector<DeviceAttributeHistory> *hist;
hist = attr->history(5);
for (int i = 0;i < 5;i++)
{
    bool fail = (*hist)[i].has_failed();
    if (fail == false)
    {
        cout << "Attribute name = " << (*hist)[i].get_name() << endl;
        cout << "Attribute quality factor = " << (*hist)[i].get_quality() << endl;
        long value;
        (*hist)[i] >> value;
        cout << "Current = " << value << endl;
```

```
        }
        else
        {
            cout << "Attribute failed !" << endl;
            cout << "Error level 0 desc = " << ((*hist)[i].get_err_stack())[0].desc << endl;
        }
        cout << "Date = " << (*hist)[i].get_date().tv_sec << endl;
    }
    delete hist;
```

Exception: NonSupportedFeature, ConnectionFailed, CommunicationFailed, DevFailed from device

6.6.4 Asynchronous methods

6.6.4.1 long AttributeProxy::read_asynch()

Read the attribute asynchronously (polling model). This call returns an *asynchronous call identifier* which is needed to get the attribute value.

Exception: ConnectionFailed

6.6.4.2 DeviceAttribute *AttributeProxy::read_reply(long id)

Check if the answer of an asynchronous read is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceAttribute object. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived. To extract attribute value, you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the DeviceAttribute object returned to the caller. This is the caller responsibility to delete this memory.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.6.4.3 DeviceAttribute *AttributeProxy::read_reply(long id, long timeout)

Check if the answer of an asynchronous read is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, it is returned to the caller in a DeviceAttribute object. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived. To extract attribute value, you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the DeviceAttribute object returned to the caller. This is the caller responsibility to delete this memory.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.6.4.4 long AttributeProxy::write_asynch(DeviceAttribute &argin)

Write the attribute asynchronously (polling model). To insert the value to write you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the user type to the attribute native type e.g. if an attribute expects a short you cannot insert it as a double (this will throw an exception) you have to insert it as a short. This call returns an *asynchronous call identifier* which is needed to get the server reply.

Exception: ConnectionFailed

6.6.4.5 void AttributeProxy::write_reply(long id)

Check if the answer of an asynchronous write is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. An exception is also thrown in case of the reply is not yet arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.6.4.6 void AttributeProxy::write_reply(long id, long timeout)

Check if the answer of an asynchronous write is arrived (polling model). id is the asynchronous call identifier. If the reply is arrived and if it is a valid reply, the call returned. If the reply is an exception, it is re-thrown by this call. If the reply is not yet arrived, the call will wait (blocking the process) for the time specified in timeout. If after timeout milliseconds, the reply is still not there, an exception is thrown. If timeout is set to 0, the call waits until the reply arrived.

Exception: AsynCall, AsynReplyNotArrived, CommunicationFailed, DevFailed from device

6.6.4.7 void AttributeProxy::read_asynch(CallBack &cb)

Read the attribute asynchronously using the callback model. The argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *attr_read()* method overloaded.

Exception: ConnectionFailed

6.6.4.8 void AttributeProxy::write_asynch(DeviceAttribute &argin, CallBack &cb)

Write the attribute asynchronously using the callback model. The argument is a reference to a callback object. This callback object should be an instance of a user class inheriting from the *Tango::CallBack* class with the *attr_written()* method overloaded.

Exception: ConnectionFailed

6.6.5 Polling related methods

6.6.5.1 bool AttributeProxy::is_polled()

Returns true if the attribute is polled. Otherwise, returns false.

6.6.5.2 int AttributeProxy::get_poll_period()

Returns the attribute polling period in mS. If the attribute is not polled, it returns 0.

6.6.5.3 void AttributeProxy::poll(int period)

Add the attribute to the list of polled attributes. The polling period is specified by "period" (in mS). If the attribute is already polled, this method will update the polling period according to "period".

6.6.5.4 void AttributeProxy::stop_poll()

Remove attribute from the list of polled attributes.

6.6.6 Event related methods

6.6.6.1 int AttributeProxy::subscribe_event(EventType event, CallBack *cb)

The client call to subscribe for event reception in the **push model**. The client implements a callback method which is triggered when the event is received either by polling or a dedicated thread. Filtering is done based on the reason specified and the event type. For example when reading the state and the reason specified is "change" the event will be fired only when the state changes. Events consist of an attribute name and the event reason. A standard set of reasons are implemented by the system, additional device specific reasons can be implemented by device servers programmers.

The *event* parameter is the event reason and must be on the enumerated values:

- Tango::CHANGE_EVENT
- Tango::PERIODIC_EVENT
- Tango::ARCHIVE_EVENT
- Tango::ATTR_CONF_EVENT

cb is a pointer to a class inheriting from the Tango CallBack class and implementing a *push_event()* method, *filters* is a variable list of name,value pairs which define additional filters for events. The *subscribe_event()* call returns an event id which has to be specified when unsubscribing from this event.

Exception: EventSystemFailed

Note: For releases prior to Tango 8, a similar call with a third argument (const vector<string> &filters) was available. This extra argument gave the user a way to define extra event filtering. For compatibility reason, this call still exist but the extra filtering features is not implemented.

6.6.6.2 int AttributeProxy::subscribe_event(EventType event, CallBack *cb, bool stateless)

This subscribe event method has the same functionality as described in the last section. It adds an additional flag called *stateless*. When the *stateless* flag is set to *false*, an exception will be thrown when the event subscription encounters a problem.

With the *stateless* flag set to *true*, the event subscription will always succeed, even if the corresponding device server is not running. The keep alive thread will try every 10 seconds to subscribe for the specified event. At every subscription retry, a callback is executed which contains the corresponding exception.

Exception: EventSystemFailed

Note: For releases prior to Tango 8, a similar call with a forth argument (const vector<string> &filters) was available. This extra argument gave the user a way to define extra event filtering and it was the third argument in the argument list. For compatibility reason, this call still exist but the extra filtering features is not implemented.

6.6.6.3 int AttributeProxy::subscribe_event(EventType event, int event_queue_size, bool stateless)

The client call to subscribe for event reception in the **pull model**. Instead of a callback method the client has to specify the size of the event reception buffer.

The event reception buffer is implemented as a round robin buffer. This way the client can set-up different ways to receive events.

Event reception buffer size = 1 : The client is interested only in the value of the last event received. All other events that have been received since the last reading are discarded.

Event reception buffer size > 1 : The client has chosen to keep an event history of a given size. When more events arrive since the last reading, older events will be discarded.

Event reception buffer size = ALL_EVENTS : The client buffers all received events. The buffer size is unlimited and only restricted by the available memory for the client.

All other parameters are similar to the descriptions given in the last two sections.

Exception: EventSystemFailed

Note: For releases prior to Tango 8, a similar call with a forth argument (const vector<string> &filters) was available. This extra argument gave the user a way to define extra event filtering and it was the third argument in the argument list. For compatibility reason, this call still exist but the extra filtering features is not implemented.

6.6.6.4 void AttributeProxy::unsubscribe_event(int event_id)

Unsubscribe a client from receiving the event specified by *event_id*. *event_id* is the event identifier returned by the *AttributeProxy::subscribe_event()* method.

Exception: EventSystemFailed

6.6.6.5 void AttributeProxy::get_events(int event_id, CallBack *cb)

The method extracts all waiting events from the event reception buffer and executes the callback method cb for every event. During event subscription the client must have chosen the pull model for this event. event_id is the event identifier returned by the AttributeProxy::subscribe_event() method.

Exception: EventSystemFailed

6.6.6.6 void AttributeProxy::get_events(int event_id, EventDataList &event_list)

The method extracts all waiting events from the event reception buffer. The returned event_list is a vector of EventData pointers. The EventData object contains the event information as for the callback methods.

During event subscription the client must have chosen the pull model for this event. event_id is the event identifier returned by the AttributeProxy::subscribe_event() method.

Exception: EventSystemFailed

6.6.6.7 void AttributeProxy::get_events(int event_id, AttrConfEventDataList &event_list)

The method extracts all waiting attribute configuration events from the event reception buffer. The returned event_list is a vector of AttrConfEventData pointers. The AttrConfEventData object contains the event information as for the callback methods.

During event subscription the client must have chosen the pull model for this event. event_id is the event identifier returned by the AttributeProxy::subscribe_event() method.

Exception: EventSystemFailed

6.6.6.8 int AttributeProxy::event_queue_size(int event_id)

Returns the number of stored events in the event reception buffer. After every call to DeviceProxy:get_events(), the event queue size is 0.

During event subscription the client must have chosen the pull model for this event. event_id is the event identifier returned by the AttributeProxy::subscribe_event() method.

Exception: EventSystemFailed

6.6.6.9 TimeVal AttributeProxy::get_last_event_date(int event_id)

Returns the arrival time of the last event stored in the event reception buffer. After every call to DeviceProxy:get_events(), the event reception buffer is empty. In this case an exception will be returned.

During event subscription the client must have chosen the pull model for this event. event_id is the event identifier returned by the AttributeProxy::subscribe_event() method.

Exception: EventSystemFailed

6.6.6.10 bool AttributeProxy::is_event_queue_empty(int event_id)

Returns true when the event reception buffer is empty.

During event subscription the client must have chosen the pull model for this event. event_id is the event identifier returned by the AttributeProxy::subscribe_event() method.

Exception: EventSystemFailed

6.6.7 Property related methods

6.6.7.1 void AttributeProxy::get_property (string&, DbData&)

Get a single property for the attribute. The property to get is specified as a string. Refer to DbDevice::get_property() and DbData sections below for details on the DbData type.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.6.7.2 void AttributeProxy::get_property (vector<string>&, DbData&)

Get a list of properties for the attribute. The properties to get are specified as a vector of strings. Refer to DbDevice::get_property() and DbData sections below for details on the DbData type.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.6.7.3 void AttributeProxy::get_property(DbData&)

Get property(ies) for the attribute. Properties to get are specified using the DbData type. Refer to DbDevice::get_property() and DbData sections below for details.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.6.7.4 void AttributeProxy::put_property(DbData&)

Put property(ies) for an attribute. Properties to put are specified using the DbData type. Refer to DbDevice::put_property() and DbData sections below for details.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.6.7.5 void AttributeProxy::delete_property (string&, DbData&)

Delete a single property for an attribute. The property to delete is specified as a string. Refer to DbDevice::delete_property() and DbData sections below for details on the DbData type.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.6.7.6 void AttributeProxy::delete_property (vector<string>&, DbData&)

Delete a list of properties for an attribute. The properties to delete are specified as a vector of strings. Refer to DbDevice::get_property() and DbData sections below for details on the DbData type.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.6.7.7 void AttributeProxy::delete_property(DbData&)

Delete property(ies) for an attribute. Properties to delete are specified using the DbData type. Refer to DbDevice::get_property() and DbData sections below for details.

Exception: NonDbDevice, ConnectionFailed (with database), CommunicationFailed (with database), DevFailed from database device

6.7 Tango::ApiUtil

This class is a singleton. Therefore, it is not necessary to create it. It will be automatically done. A static method allows a user to retrieve the instance

6.7.1 static ApiUtil *ApiUtil::instance()

Return the ApiUtil singleton instance.

6.7.2 static void ApiUtil::cleanup()

Destroy the ApiUtil singleton instance.

6.7.3 long ApiUtil::pending_asynch_call(asyn_req_type req)

Return number of asynchronous pending requests (any device). The input parameter is an enumeration with three values which are:

POLLING : Return only polling model asynchronous request number

CALL_BACK : Return only callback model asynchronous request number

ALL_ASYNCH : Return all asynchronous request number

Exception: None

6.7.4 void ApiUtil::get_asynch_replies()

Fire callback methods for all (any device) asynchronous requests (command and attribute) with already arrived replied. Returns immediately if there is no replies already arrived or if there is no asynchronous requests.

Exception: None, all errors are reported using the err and errors fields of the parameter passed to the callback method. See chapter 6.8 for details.

6.7.5 void ApiUtil::get_asynch_replies(long timeout)

Fire callback methods for all (any device) asynchronous requests (command and attributes) with already arrived replied. Wait and block the caller for timeout milliseconds if they are some device asynchronous requests which are not yet arrived. Returns immediately if there is no asynchronous request. If timeout is set to 0, the call waits until all the asynchronous requests sent has received a reply.

Exception: AsynReplyNotArrived. All other errors are reported using the err and errors fields of the object passed to the callback methods. See chapter 6.8 for details.

6.7.6 void ApiUtil::set_asynch_cb_sub_model(cb_sub_model model)

Set the asynchronous callback sub-model between the pull and push sub-model. See chapter 4.5 to read the definition of these sub-model. The cb_sub_model data type is an enumeration with two values which are :

PUSH_CALLBACK : The push sub-model

PULL_CALLBACK : The pull sub-model

By default, all Tango client using asynchronous callback model are in pull sub-model. This call must be used to switch to the push sub-model. NOTE that in push sub-model, a separate thread is spawned to deal with server replies.

Exception: None

6.7.7 cb_sub_model ApiUtil::get_asynch_cb_sub_model()

Get the asynchronous callback sub-model.

Exception: None

6.7.8 static int ApiUtil::get_env_var(const char *name,string &value)

Get environment variable. On Unixes OS, this call tries to get the variable in the caller environment then in a file *.tangorc* in the user home directory and finally in a file */etc/tangorc*. On Windows, this call looks in the user environment then in a file stored in %TANGO_HOME%/tangorc. This method returns 0 of the environment variable is found. Otherwise, it returns -1.

Exception: None

6.7.9 void set_event_buffer_hwm(DevLong val)

Set the client event buffer high water mark (HWM)

6.8 Asynchronous callback related classes

6.8.1 Tango::CallBack

When using the event push model (callback automatically executed), there are some cases (same callback used for events coming from different devices hosted in device server process running on different hosts) where the callback method could be executed concurently by different threads started by the ORB. The user has to code his callback method in a **thread safe** manner.

6.8.1.1 void CallBack::cmd_ended(CmdDoneEvent *event)

This method is defined as being empty and must be overloaded by the user when the asynchronous callback model is used. This is the method which will be executed when the server reply from a command_inout is received in both push and pull sub-mode.

6.8.1.2 void CallBack::attr_read(AttrReadEvent *event)

This method is defined as being empty and must be overloaded by the user when the asynchronous callback model is used. This is the method which will be executed when the server reply from a read_attribute(s) is received in both push and pull sub-mode.

6.8.1.3 void CallBack::attr_written(AttrWrittenEvent *event)

This method is defined as being empty and must be overloaded by the user when the asynchronous callback model is used. This is the method which will be executed when the server reply from a write_attribute(s) is received in both push and pull sub-mode.

6.8.1.4 void CallBack::push_event(EventData *event)

This method is defined as being empty and must be overloaded by the user when events are used. This is the method which will be executed when the server send event(s) to the client.

6.8.1.5 void CallBack::push_event(AttrConfEventData *event)

This method is defined as being empty and must be overloaded by the user when events are used. This is the method which will be executed when the server send attribute configuration change event(s) to the client.

6.8.1.6 void CallBack::push_event(DataReadyEventData *event)

This method is defined as being empty and must be overloaded by the user when events are used. This is the method which will be executed when the server send attribute data ready event(s) to the client.

6.8.2 Tango::CmdDoneEvent

This class is used to pass data to the callback method in asynchronous callback model for command execution. It contains the following public field

device : The DeviceProxy object on which the call was executed (Tango::DeviceProxy *)

cmd_name : The command name (string &)

argout : The command argout (DeviceData &)

err : A boolean flag set to true if the command failed. False otherwise (bool)

errors : The error stack (DevErrorList &)

6.8.3 Tango::AttrReadEvent

This class is used to pass data to the callback method in asynchronous callback model for read_attribute(s) execution. It contains the following public field

device : The DeviceProxy object on which the call was executed (Tango::DeviceProxy *)

attr_names : The attribute name list (vector<string> &)

argout : The attribute data (vector<DeviceAttribute> *)

err : A boolean flag set to true if the request failed. False otherwise (bool)

errors : The error stack (DevErrorList &)

To extract attribute value(s), you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the vector of DeviceAttribute objects passed to the caller. This is the caller responsibility to delete this memory.

6.8.4 Tango::AttrWrittenEvent

This class is used to pass data to the callback method in asynchronous callback model for write_attribute(s) execution. It contains the following public field

device : The DeviceProxy object on which the call was executed (Tango::DeviceProxy *)

attr_names : The attribute name list (vector<string> &)

err : A boolean flag set to true if the request failed. False otherwise (bool)

errors : The error stack (DevErrorList &)

6.8.5 Tango::EventData

This class is used to pass data to the callback method when an event is sent to the client. It contains the following public field

device : The DeviceProxy object on which the call was executed (Tango::DeviceProxy *)

attr_name : The attribute name (std::string &)

event : The event name (std::string &)

attr_value : The attribute data (DeviceAttribute *)

err : A boolean flag set to true if the request failed. False otherwise (bool)

errors : The error stack (DevErrorList &)

To extract attribute value(s), you have to use the operator of the class DeviceAttribute which corresponds to the data type of the attribute. NOTE: There is no automatic type conversion from the attribute native type to user type e.g. if an attribute returns a short you cannot extract it as a double, you have to extract it as a short. Memory has been allocated for the vector of DeviceAttribute objects passed to the caller. This is the caller responsibility to delete this memory.

6.8.6 Tango::AttrConfEventData

This class is used to pass data to the callback method when an attribute configuration event is sent to the client. It contains the following public field

device : The DeviceProxy object on which the call was executed (Tango::DeviceProxy *)

attr_name : The attribute name (std::string &)

event : The event name (std::string &)

attr_conf : The attribute configuration (AttributeInfoEx *)

err : A boolean flag set to true if the request failed. False otherwise (bool)

errors : The error stack (DevErrorList &)

6.8.7 Tango::DataReadyEventData

This class is used to pass data to the callback method when an attribute data ready event is sent to the client. It contains the following public field

device : The DeviceProxy object on which the call was executed (Tango::DeviceProxy *)

attr_name : The attribute name (std::string &)

event : The event name (std::string &)

attr_data_type :The attribute data type (int)

ctr : The user counter. Set to 0 if not defined when sent by the server (int)

err : A boolean flag set to true if the request failed. False otherwise (bool)

errors : The error stack (DevErrorList &)

6.9 Tango::Group

6.9.1 Constructor and Destructor

6.9.1.1 Group::Group (const std::string& name)

Instanciate an empty group. The group name allows retrieving a sub-group in the hierarchy.

See also: Group::~Group(), Group::get_group().

6.9.1.2 Group::~Group ()

Delete a group and all its elements.

Be aware that a group always gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a Group (respectively a DeviceProxy) returned by a call to *Tango::Group::get_group()* (respectively to *Tango::Group::get_device()*). Use the *Tango::Group::remove()* method instead.

See also: Group::Group(), Group::remove(), Group::remove_all().

6.9.2 Group Management Related Methods

6.9.2.1 void Group::add (Group* group, int timeout_ms = -1)

Attaches a (sub)group.

Be aware that a group always gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a Group attached to a Group. Use the *Group::remove()* method instead.

If *timeout_ms* parameter is different from -1, the client side timeout associated to each device composing the group added is set to *timeout_ms* milliseconds. If *timeout_ms* is -1, timeouts are not changed.

This method does nothing if the specified group is already attached (i.e. it is silently ignored) and *timeout_ms* = -1.

If the specified group is already attached and *timeout_ms* is different from -1, the client side timeout of each device composing the group given in parameter is set to *timeout_ms* milliseconds.

See also: all other forms of Group::add() and Group::set_timeout_millis().

6.9.2.2 void Group::add (const std::string& pattern, int timeout_ms = -1)

Attaches any device which name matches the specified *pattern*.

The pattern parameter can be a simple device name or a device name pattern (e.g. domain_*/ family/member_*).

This method first asks to the Tango database the list of device names matching the pattern. Devices are then attached to the group in the order in which they are returned by the database.

Any device already present in the hierarchy (i.e. a device belonging to the group or to one of its subgroups) is silently ignored but its client side timeout is set to *timeout_ms* milliseconds if *timeout_ms* is different from -1.

Set the client side timeout of each device matching the specified *pattern* to *timeout_ms* milliseconds if *timeout_ms* is different from -1.

See also: all other forms of Group::add() and Group::set_timeout_millis().

6.9.2.3 void Group::add (const std::vector<std::string>& patterns, int timeout_ms = -1)

Attaches any device which name matches one of the specified patterns.

The *patterns* parameter can be an array of device names and/or device name patterns.

This method first asks to the Tango database the list of device names matching one the patterns. Devices are then attached to the group in the order in which they are returned by the database.

Any device already present in the hierarchy (i.e. a device belonging to the group or to one of its subgroups), is silently ignored but its client side timeout is set to *timeout_ms* milliseconds if *timeout_ms* is different from -1.

If *timeout_ms* is different from -1, the client side timeouts of all devices matching the specified *patterns* are set to *timeout_ms* milliseconds.

See also: all other forms of Group::add() and Group::set_timeout_millis().

6.9.2.4 void Group::remove (const std::string& pattern, bool fwd = true)

Removes any group or device which name matches the specified pattern.

The *pattern* parameter can be a group name, a device name or a device name pattern (e.g domain_*/family/member_*).

Since we can have groups with the same name in the hierarchy, a group name can be fully qualified to specify which group should be removed. Considering the following group:

```
-> gauges
   | -> cell-01
   |      |-> penning
   |      |     |-> ...
   |      |-> pirani
   |            |-> ...
   | -> cell-02
   |      |-> penning
   |      |     |-> ...
   |      |-> pirani
   |            |-> ...
   | -> cell-03
   |      |-> ...
   |
   | -> ...
```

A call to gauges->remove("penning") will remove any group named "penning" in the hierarchy while gauges->remove("gauges.cell-02.penning") will only remove the specified group.

If *fwd* is set to true (the default), the remove request is also forwarded to subgroups. Otherwise, it is only applied to the local set of elements. For instance, the following code remove any stepper motor in the hierarchy:

```
root_group->remove("*/stepper_motor/*");
```

See also: all other forms of Group::remove().

6.9.2.5 void Group::remove (const std::vector<std::string>& patterns, bool fwd = true)

Removes any group or device which name matches the specified patterns.

The *patterns* parameter can be an array of group names and/or device names and/or device name patterns.

Since we can have groups with the same name in the hierarchy, a group name can be fully qualified to specify which group should be removed. See previous method for details.

If *fwd* is set to true (the default), the remove request is also forwarded to subgroups. Otherwise, it is only applied to the local set of elements.

See also: all other forms of Group::remove().

6.9.2.6 void Group::remove_all (void)

Removes all elements in the group. After such a call, the group is empty.

See also: all forms of Group::remove().

6.9.2.7 bool Group::contains (const std::string& pattern, bool fwd = true)

Returns true if the hierarchy contains groups and/or devices which name matches the specified *pattern*. Returns false otherwise.

The pattern can be a fully qualified or simple group name, a device name or a device name pattern.

If *fwd* is set to true (the default), the request is also forwarded to subgroups. Otherwise, it is only applied to the local set of elements.

See also: Group::get_device(), Group::get_group().

6.9.2.8 DeviceProxy* Group::get_device (const std::string& device_name)

Returns a reference to the specified device or NULL if there is no device by that name in the group. This method may throw an exception in case the specified device belongs to the group but can't be reached (not registered, down...). See example below. See also the Tango::DeviceProxy class documentation for details.

```
try
{
  Tango::DeviceProxy *dp = g->get_device("my/device/01");
  if (dp == 0)
  {
// my/device/01 doe snot belongs to the group
  {
}
catch (const Tango::DevFailed &df)
{
// my/device/01 belongs to the group but can't be reached
}
```

The request is systematically forwarded to subgroups (i.e. if no device named device_name could be found in the local set of devices, the request is forwarded to subgroups).

Be aware that a group always gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a DeviceProxy returned by the *Group::get_device()* method. Use the *Tango::Group::remove()* method instead.

See also: other form of Group::get_device(), Group::get_size(), Group::get_group(), Group::contains().

6.9.2.9 DeviceProxy* Group::get_device (long idx)

Returns a reference to the "idx-th" device in the hierarchy or NULL if the hierarchy contains less than "*idx*" devices. This method may throw an exception in case the specified device belongs to the group but can't be reached (not registered, down...). See previous example. See also the Tango::DeviceProxy class documentation for details.

The request is systematically forwarded to subgroups (i.e. if the local set of devices contains less than "idx" devices, the request is forwarded to subgroups).

Be aware that a group always gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a DeviceProxy returned by the *Group::get_device()* method. Use the *Tango::Group::remove()* method instead.

See also: other form of Group::get_device(), Group::get_size(), Group::get_group, Group::contains().

6.9.2.10 DeviceProxy* Group::operator[] (long i)

Returns a reference to the "idx-th" device in the hierarchy or NULL if the hierarchy contains less than "idx" devices. See the Tango::DeviceProxy class documentation for details.

The request is systematically forwarded to subgroups (i.e. if the local set of devices contains less than "idx" devices, the request is forwarded to subgroups).

Be aware that a group always gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a DeviceProxy returned by the *Group::get_device()* method. Use the *Tango::Group::remove()* method instead.

See also: other form of Group::get_device(), Group::get_size(), Group::get_group(), Group::contains().

6.9.2.11 Group* Group::get_group (const std::string& group_name)

Returns a reference to the specified group or NULL if there is no group by that name. The group_name can be a fully qualified name.

Considering the following group:

```
-> gauges
    |-> cell-01
    |    |-> penning
    |    |    |-> ...
    |    |-> pirani
    |    |-> ...
    |-> cell-02
    |    |-> penning
    |    |    |-> ...
    |    |-> pirani
    |    |-> ...
    | -> cell-03
    |    |-> ...
    |
    | -> ...
```

A call to gauges->get_group("penning") returns the first group named "penning" in the hierarchy (i.e. gauges.cell-01.penning) while gauges->get_group("gauges.cell-02.penning") returns the specified group.

The request is systematically forwarded to subgroups (i.e. if no group named group_name could be found in the local set of elements, the request is forwarded to subgroups).

Be aware that a group always gets the ownership of its children and deletes them when it is itself deleted. Therefore, never try to delete a Group returned by the *Group::get_group()* method. Use the *Tango::Group::remove()* method instead.

See also: Group::get_device(), Group::contains().

6.9.2.12 long Group::get_size (bool fwd = true)

Return the number of devices in the hierarchy (respectively the number of device in the group) if the forward option is set to true (respectively set to false).

6.9.2.13 std::vector<std::string> Group::get_device_list (bool fwd = true)

Returns the list of devices currently in the hierarchy.

If *fwd* is set to true (the default) the request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Considering the following hierarchy:

```
g2->add("my/device/04"); g2->add("my/device/05");

g4->add("my/device/08"); g4->add("my/device/09");

g3->add("my/device/06");
g3->addg(g4);
g3->add("my/device/07");

g1->add("my/device/01");
g1->add(g2);
g1->add("my/device/03");
g1->add(g3);
g1->add("my/device/02");
```

The returned vector content depends on the value of the forward option. If set to true, the results will be organized as follows:

```
std::vector<std::string> dl = g1->get_device_list(true);
```

dl[0] contains "my/device/01" which belongs to g1
dl[1] contains "my/device/04" which belongs to g1.g2
dl[2] contains "my/device/05" which belongs to g1.g2
dl[3] contains "my/device/03" which belongs to g1
dl[4] contains "my/device/06" which belongs to g1.g3
dl[5] contains "my/device/08" which belongs to g1.g3.g4
dl[6] contains "my/device/09" which belongs to g1.g3.g4
dl[7] contains "my/device/07" which belongs to g1.g3
dl[8] contains "my/device/02" which belongs to g1

If the forward option is set to false, the results are:

```
std::vector<std::string> dl = g1->get_device_list(false);
```

dl[0] contains "my/device/01" which belongs to g1
dl[1] contains "my/device/03" which belongs to g1
dl[2] contains "my/device/02" which belongs to g1

6.9.3 "A la" DeviceProxy Methods

6.9.3.1 bool Group::ping (bool fwd = true)

Ping all devices in a group. This method returns true if all devices in the group are alive, false otherwise.

If *fwd* is set to true (the default), the request is also forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

6.9.3.2 void Group::set_timeout_millis(int timeout_ms)

Set client side timeout for all devices composing the group in milliseconds. Any method which takes longer than this time to execute will throw an exception.

Exception: none (errors are ignored).

6.9.3.3 GroupCmdReplyList Group::command_inout (const std::string& c, bool fwd = true)

Executes a Tango command on a group. This method is synchronous and does not return until replies are obtained or timeouts occurred.

The parameter *c* is the name of the command.

If *fwd* is set to true (the default), the request is also forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Command results are returned in a GroupCmdReplyList. See Obtaining command result for details (4.7.3.1). See also Case 1 of executing a command (4.7.3.2) for an example.

6.9.3.4 GroupCmdReplyList Group::command_inout (const std::string& c, const DeviceData& d, bool fwd = true)

Executes a Tango command on each device in the group. This method is synchronous and does not return until replies are obtained or timeouts occurred.

The parameter *c* is the name of the command.

The second parameter *d* is a Tango generic container for command carrying the command argument. See the Tango::DeviceData documentation.

If *fwd* is set to true (the default), the request is also forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Command results are returned in a GroupCmdReplyList. See Obtaining command results (4.7.3.1) for details. See also Case 2 of executing a command (4.7.3.4) for an example.

6.9.3.5 template<typename T> GroupCmdReplyList Group::command_inout (const std::string& c, const std::vector<T>& d, bool fwd = true)

Executes a Tango command on each device in the group. This method is synchronous and does not return until replies are obtained or timeouts occurred.

This implementation of command_inout allows passing a specific input argument to each device in the group. In order to use this form of command_inout, the user must have an "a priori" and "perfect" knowledge of the devices order in the group.

The parameter *c* is the name of the command.

The std::vector *d* contains a specific argument value for each device in the group. Since this method is a template, d is able to contain any Tango command argument type. Its size must equal Group::get_size(fwd). Otherwise, an exception is thrown. The order of the argument values must follows the order of the devices in the group (d[0] => 1st device, d[1] => 2nd device and so on).

If *fwd* is set to true (the default), the request is also forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Command results are returned in a GroupCmdReplyList. See Obtaining command results (4.7.3.1) for details. See also Case 3 of executing a command (4.7.3.5) for an example of this special form of command_inout.

6.9.3.6 long Group::command_inout_asynch (const std::string& c, bool fgt = false, bool fwd = true, long rsv = -1)

Executes a Tango command on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to *Group::command_inout_reply()* to obtain the results.

The parameter *c* is the name of the command.

The parameter *fgt* is a fire and forget flag. If set to true, it means that no reply is expected (i.e. the caller does not care about it and will not even try to get it). A false default value is provided.

If the parameter *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Finally, *rsv* is reserved for internal purpose and should not be modify. This parameter may disappear in a near future.

See Case 1 of Executing a command (4.7.3.2) for an example.

6.9.3.7 long Group::command_inout_asynch (const std::string& c, const DeviceData& d, bool fgt = false, bool fwd = true, long rsv = -1)

Executes a Tango command on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to *Group::command_inout_reply()* to obtain the results.

The parameter *c* is the name of the command.

The second parameter *d* is a Tango generic container for command carrying the command argument. See the Tango::DeviceData documentation for details.

The parameter *fgt* is a fire and forget flag. If set to true, it means that no reply is expected (i.e. the caller does not care about it and will not even try to get it). A false default value is provided.

If the parameter *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Finally, *rsv* is reserved for internal purpose and should not be modify. This parameter may disappear in a near future.

See Case 2 of Executing a command (4.7.3.4) for an example.

6.9.3.8 long Group::command_inout_asynch (const std::string& c, const std::vector<T>& d, fgt = false, bool fwd = true)

Executes a Tango command on each device in the group asynchronously. The method send the request to all devices and return immediately. Pass the returned request id to Group::command_inout_reply to obtain the results.

This implementation of command_inout allows passing a specific input argument to each device in the group. In order to use this form of command_inout_asynch, the user must have an "a priori" and "perfect" knowledge of the devices order in the group.

The parameter *c* is the name of the command.

The std::vector *d* contains a specific argument value for each device in the group. d is able to contain any Tango command argument type. Its size must equal Group::get_size(fwd). Otherwise, an exception is thrown. The order of the argument values must follows the order of the devices in the group (d[0] => 1st device, d[1] => 2nd device and so on).

The parameter *fgt* is a fire and forget flag. If set to true, it means that no reply is expected (i.e. the caller does not care about it and will not even try to get it). A false default value is provided.

If *fwd* is set to true (the default), the request is also forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

See Case 3 of Executing a command (4.7.3.5) for an example of this special form of command_inout.

6.9.3.9 GroupCmdReplyList Group::command_inout_reply (long req_id, long timeout_ms = 0)

Returns the results of an asynchronous command.

The first parameter *req_id* is a request identifier previously returned by one of the command_inout_asynch methods.

For each device in the hierarchy, if the command result is not yet available, command_inout_reply wait *timeout_ms* milliseconds before throwing an exception. This exception will be part of the global reply. If *timeout_ms* is set to 0, command_inout_reply waits "indefinitely".

Command results are returned in a GroupCmdReplyList. See Obtaining command results (4.7.3.1) for details.

6.9.3.10 GroupAttrReplyList Group::read_attribute (const std::string& a, bool fwd = true)

Reads an attribute on each device in the group. This method is synchronous and does not return until replies are obtained or timeouts occurred.

The parameter *a* is the name of the attribute to read.

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Attribute values are returned in a GroupAttrReplyList. See Obtaining attribute values (4.7.4.1) for details. See also Reading an attribute (4.7.4) for an example.

6.9.3.11 long Group::read_attribute_asynch (const std::string& a, bool fwd = true, long rsv = -1)

Reads an attribute on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to *Group::read_attribute_reply()* to obtain the results.

The parameter *a* is the name of the attribute to read.

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

The last parameter (rsv) is reserved for internal purpose and should not be modify. It may disappear in a near future.

See Reading an attribute (4.7.4) for an example.

6.9.3.12 GroupAttrReplyList Group::read_attribute_reply (long req_id, long timeout_ms = 0)

Returns the results of an asynchronous attribute reading.

The first parameter *req_id* is a request identifier previously returned by read_attribute_asynch.

For each device in the hierarchy, if the attribute value is not yet available, read_attribute_reply wait *timeout_ms* milliseconds before throwing an exception. This exception will be part of the global reply. If *timeout_ms* is set to 0, read_attribute_reply waits "indefinitely".

Replies are returned in a GroupAttrReplyList. See Obtaining attribute values (4.7.4.1) for details.

6.9.3.13 GroupReplyList Group::write_attribute (const DeviceAttribute& d, bool fwd = true)

Writes an attribute on each device in the group. This method is synchronous and does not return until acknowledgements are obtained or timeouts occurred.

The first parameter *d* is a Tango generic container for attribute carrying both the attribute name and the value. See the Tango::DeviceAttribute documentation for details.

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Acknowledgements are returned in a GroupReplyList. See Obtaining acknowledgements (4.7.5.1) for details. See also Case 1 of Writing an attribute (4.7.5.2) for an example.

6.9.3.14 GroupReplyList Group::write_attribute (const std::string& a, const std::vector<T>& d, bool fwd = true)

Writes an attribute on each device in the group. This method is synchronous and does not return until replies are obtained or timeouts occurred.

This implementation of write_attribute allows writing a specific value to each device in the group. In order to use this form of write_attribute, the user must have an "a priori" and "perfect" knowledge of the devices order in the group.

The parameter *a* is the name of the attribute.

The std::vector *d* contains a specific value for each device in the group. Since this method is a template, d is able to contain any Tango attribute type. Its size must equal *Group::get_size(fwd)*. Otherwise, an exception is thrown. The order of the attribute values must follows the order of the devices in the group (d[0] => 1st device, d[1] => 2nd device and so on).

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Acknowledgements are returned in a GroupReplyList. See Obtaining acknowledgements (4.7.5.1) for details. See also Case 2 of Writing an attribute (4.7.5.3) for an example.

6.9.3.15 long Group::write_attribute_asynch (const DeviceAttribute& d, bool fwd = true, long rsv = -1)

Writes an attribute on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to *Group::write_attribute_reply()* to obtain the acknowledgements.

The first parameter *d* is a Tango generic container for attribute carrying both the attribute name and the value. See the Tango::DeviceAttribute documentation for details.

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

The last parameter *rsv* is reserved for internal purpose and should not be modify. It may disappear in a near future.

See Case 1 of Writing an attribute (4.7.5.2) for an example.

6.9.3.16 long Group::write_attribute_asynch (const std::string& a, const std::vector<T>& d, bool fwd = true)

Writes an attribute on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to *Group::write_attribute_reply()* to obtain the acknowledgements.

This implementation of write_attribute_asynch allows writing a specific value to each device in the group. In order to use this form of write_attribute_asynch, the user must have an "a priori" and "perfect" knowledge of the devices order in the group.

The parameter *a* is the name of the attribute.

The std::vector *d* contains a specific value for each device in the group. Since this method is a template, d is able to contain any Tango attribute type. Its size must equal *Group::get_size(fwd)*. Otherwise, an exception is thrown. The order of the attribute values must follows the order of the devices in the group (d[0] => 1st device, d[1] => 2nd device and so on).

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

See Case2 of Writing an attribute (4.7.5.3) for an example.

6.9.3.17 GroupReplyList Group::write_attribute_reply (long req_id, long timeout_ms = 0)

Returns the acknowledgements of an asynchronous attribute writing.

The first parameter *req_id* is a request identifier previously returned by one of the write_attribute_asynch implementation.

For each device in the hierarchy, if the acknowledgement is not yet available, write_attribute_reply wait *timeout_ms* milliseconds before throwing an exception. This exception will be part of the global reply. If *timeout_ms* is set to 0, write_attribute_reply waits "indefinitely".

Acknowledgements are returned in a GroupReplyList. See Obtaining acknowledgements 4.7.5.1 for details.

6.9.3.18 GroupAttrReplyList Group::read_attributes (const std::vector<std::string>& al, bool fwd = true)

Reads several attributes on each device in the group. This method is synchronous and does not return until replies are obtained or timeouts occurred.

The parameter *al* is a vector containing the name of the attributes to be read.

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

Attribute values are returned in a GroupAttrReplyList. See Obtaining attribute values (4.7.4.1) for details. See also Reading an attribute (4.7.4) for an example. The order of attribute value returned in the GroupAttrReplyList is all attributes for first element in the group followed by all attributes for the second group element and so on.

6.9.3.19 long Group::read_attributes_asynch (const std::vector<std::string>& al, bool fwd = true, long rsv = -1)

Reads several attributes on each device in the group asynchronously. The method sends the request to all devices and returns immediately. Pass the returned request id to *Group::read_attributes_reply()* to obtain the results.

The parameter *al* is a vector containing the name of the attributes to be read.

If *fwd* is set to true (the default) request is forwarded to subgroups. Otherwise, it is only applied to the local set of devices.

The last parameter (rsv) is reserved for internal purpose and should not be modify. It may disappear in a near future.

See Reading an attribute (4.7.4) for an example.

6.9.3.20 GroupAttrReplyList Group::read_attributes_reply (long req_id, long timeout_ms = 0)

Returns the results of an asynchronous attribute reading.

The first parameter *req_id* is a request identifier previously returned by read_attributes_asynch.

For each device in the hierarchy, if the attribute value is not yet available, read_attributes_reply wait *timeout_ms* milliseconds before throwing an exception. This exception will be part of the global reply. If *timeout_ms* is set to 0, read_attributes_reply waits "indefinitely".

Replies are returned in a GroupAttrReplyList. See Obtaining attribute values (4.7.4.1) for details. The order of attribute value returned in the GroupAttrReplyList is all attributes for first element in the group followed by all attributes for the second group element and so on.

6.10 Tango::Database

A high level object which contains the link to the database. It has methods for all database commands e.g. get_device_property(), device_list(), info(), etc.

6.10.1 Database::Database()

Create a TANGO Database object. The constructor uses the environment variable “TANGO_HOST” to determine which instance of the TANGO database to connect to. Example :

```
using namespace Tango;
Database *db = new Database();
```

The Database class also has a copy constructor and one assignement operator defined.

6.10.2 string Database::get_info()

Query the database for some general info about the tables in the database. Result is returned as a string. Example :

```
cout << db->get_info() << endl;
```

will return information like this :

```
Running since 2000-11-06 14:10:46

Devices defined  = 115
Devices exported  = 41
Device servers defined  = 47
Device servers exported  = 17

Class properties defined  = 5
Device properties defined  = 130
Class attribute properties defined  = 20
Device attribute properties defined  = 92
```

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)

6.10.3 void Database::add_device(DbDevInfo&)

Add a device to the database. The device name, server and class are specified in the DbDevInfo structure. Example :

```
DbDevInfo my_device_info;
my_device_info.name = "my/own/device";
my_device_info._class = "MyDevice";
my_device_info.server = "MyServer/test";
db->add_device(my_device_info);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.4 void Database::delete_device(string)

Delete the device of the specified name from the database. Example

```
db->delete_device("my/own/device");
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError, DB_DeviceNotDefined)*

6.10.5 DbDevImportInfo Database::import_device(string &)

Query the database for the export info of the specified device. The command returns the information in a DbDevImportInfo structure. Example :

```
DbDevImportInfo my_device_import;
my_device_import = db->import_device("my/own/device");
cout << " device " << my_device_import.name;
cout << "exported " << my_device_import.exported;
cout << "ior " << my_device_import.ior;
cout << "version " << my_device_import.version;
cout << endl;
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError, DB_DeviceNotDefined)*

6.10.6 void Database::export_device(DbDevExportInfo&)

Update the export info for this device in the database. Device name, server, class, pid and version are specified in the DbDevExportInfo structure. Example :

```
DbDevExportInfo my_device_export;
my_device_export.name = "my/own/device";
my_device_export.ior = "the real ior";
my_device_export.host = "dumela";
my_device_export.version = "1.0";
my_device_export.pid = get_pid();
db->export_device(my_device_export);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError, DB_DeviceNotDefined)*

6.10.7 void Database::unexport_device(string)

Mark the specified device as un-exported in the database. Example :

```
db->unexport_device("my/own/device");
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.8 void Database::add_server(string &, DbDevInfos&)

Add a group of devices to the database. The device names, server names and classes are specified in the vector of DbDevInfo structures.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.9 void Database::delete_server(string &)

Delete the device server and its associated devices from the database.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.10 void Database::export_server(DbDevExportInfos &)

Export a group of devices to the database. The device names, IOR, class, server name, pid etc. are specified in the vector of DbDevExportInfo structures.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.11 void Database::unexport_server(string &)

Mark all devices exported for this server as unexported.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.12 DbDatum Database::get_services(string &servicename,string &instname)

Query database for specified services.The instname parameter can be a wildcard character ("*").

```
string servicename("HdbManager");
string instname("ctrm");
DbDatum db_datum = db->get_services(servicename,instname);
vector<string> service_list;
db_datum >> service_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.13 void Database::register_service(string &servicename,string &instname,string &devname)

Register the specified service wihtin the database.

```
string servicename("HdbManager");
string instname("ctrm");
string devname("sys/hdb/1");
db->register_service(servicename,instname,devname);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.14 void Database::unregister_service(string &servicename,string &instname)

Unregister the specified service from the database.

```
string servicename("HdbManager");
string instname("ctrm");
db->unregister_service(servicename,instname);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.15 DbDatum Database::get_host_list()

Returns the list of all host names registered in the database.

```
DbDatum db_datum = db->get_host_list();
vector<string> host_list;
db_datum >> host_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.16 DbDatum Database::get_host_list(string &wildcard)

Returns the list of all host names registered in the database which match the specified wildcard (eg: "l-c0*").

```
string wildcard("l-c0*");
DbDatum db_datum = db->get_host_list(wildcard);
vector<string> host_list;
db_datum >> host_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.17 DbDatum Database::get_server_class_list(string &server)

Query the database for a list of classes instancied by the specified server. The DServer class exists in all TANGO servers and for this reason this class is removed from the returned list.

```
string server("Serial/1");
DbDatum db_datum = db->get_server_class_list(server);
vector<string> class_list;
db_datum >> class_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.18 DbDatum Database::get_server_name_list()

Return the list of all server names registered in the database.

```
DbDatum db_datum = db->get_server_name_list();
vector<string> server_list;
db_datum >> server_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.19 DbDatum Database::get_instance_name_list(string &servername)

Return the list of all instance names existing in the database for the specifed server.

```
string servername("Serial");
DbDatum db_datum = db->get_instance_name_list(servername);
vector<string> instance_list;
db_datum >> instance_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.20 DbDatum Database::get_server_list()

Return the list of all servers registered in the database.

```
DbDatum db_datum = db->get_server_list();
vector<string> server_list;
db_datum >> server_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.21 DbDatum Database::get_server_list(string &wildcard)

Return the list of all servers registered in the database which match the specified wildcard (eg: "Serial/*").

```
string wildcard("Serial/*");
DbDatum db_datum = db->get_server_list(wildcard);
vector<string> server_list;
db_datum >> server_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.22 DbDatum Database::get_host_server_list(string &hostname)

Query the database for a list of servers registred on the specified host.

```
string host("kidiboo");
DbDatum db_datum = db->get_host_server_list(wildcard);
vector<string> server_list;
db_datum >> server_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.23 DbServerInfo Database::get_server_info(string &server)

Query the database for server information.

```
string server("Serial/1");
DbServerInfo info = db->get_server_info(server);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.24 void Database::put_server_info(DbServerInfo &info)

Add/update server information in the database.

```
DbServerInfo info;
info.name = "Serial/1"; // Server (name/instance)
info.host = "kidiboo"; // Register on host kidiboo
info.mode = 1; // Controlled by Astor flag (0 or 1)
info.level = 3; // Startup level (Used by Astor)
db->put_server_info(info);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.25 void Database::delete_server_info(string &server)

Delete server information of the specifed server from the database.

```
string server("Serial/1");
db->delete_server_info(server);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.26 DbDatum Database::get_device_name(string &, string &)

Query the database for a list of devices served by the specified server (1st parameter) and of the specified class (2nd parameter). The method returns a DbDatum type. The device names are stored as an array of strings. Here is two code example of how to extract the names from the DbDatum type :

```
vector<string> device_names;
device_names << db_datum;
```

or :

```
for (int i=0; i< db_datum.size(); i++)
{
device_name[i] = db_datum.value_string[i];
}
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.27 DbDatum Database::get_device_exported(string &)

Query the database for a list of exported devices whose names satisfy the supplied filter (* is wildcard for any character(s)). This method returns a DbDatum type. See the method get_device_name() for an example of how to extract the list of aliases from the DbDatum type.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.28 DbDatum Database::get_device_domain(string &)

Query the database for a list of device domain names which match the wildcard provided. Wildcard character is * and matches any number of characters. Domain names are case insensitive. This method returns a DbDatum type. See the method get_device_name() for an example of how to extract the list of aliases from the DbDatum type.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.29 DbDatum Database::get_device_family(string &)

Query the database for a list of device family names which match the wildcard provided. Wildcard character is * and matches any number of characters. Family names are case insensitive. This method returns a DbDatum type. See the method get_device_name() for an example of how to extract the list of aliases from the DbDatum type.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.30 DbDatum Database::get_device_member(string &)

Query the database for a list of device member names which match the wildcard provided. Wildcard characters is * and matches any number of characters. Member names are case insensitive. This method returns a DbDatum type. See the method get_device_name() for an example of how to extract the list of aliases from the DbDatum type.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.31 DbDatum Database::get_device_class_list(string &server)

Query the database for a list of devices and classes served by the specified server. Return a list with the following structure: {device name,class name,device name,class name,...}

```
string server("Serial/1");
DbDatum db_datum = db->get_device_class_list(server);
vector<string> dev_list;
db_datum >> dev_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.32 string Database::get_class_for_device(string &devname)

Return the class of the specified device.

```
string devname("sr/rf-cavity/1");
string classname = db->get_class_for_device(devname);
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.33 DbDatum Database::get_class_inheritance_for_device(string &devname)

Return the class inheritance scheme of the specified device.

```
string devname("sr/rf-cavity/1");
DbDatum db_datum = db->get_class_inheritance_for_device(devname);
vector<string> class_list;
db_datum >> class_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.34 DbDatum Database::get_device_exported_for_class(string &classname)

Query database for list of exported devices for the specified class.

```
string classname("MyClass");
DbDatum db_datum = db->get_device_exported_for_class(classname);
vector<string> dev_list;
db_datum >> dev_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.35 DbDatum Database::get_object_list(string &wildcard)

Query the database for a list of object (free properties) for which properties are defined and which match the specified wildcard.

```
string wildcard("Optic*");
DbDatum db_datum = db->get_object_list(wildcard);
vector<string> obj_list;
db_datum >> obj_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.36 DbDatum Database::get_object_property_list(string &objectname,string &wildcard)

Query the database for a list of properties defined for the specified object and which match the specified wildcard.

```
string objname("OpticID9");
string wildcard("Delta*");
DbDatum db_datum = db->get_object_property_list(objname,wildcard);
vector<string> prop_list;
db_datum >> prop_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.37 void Database::get_property(string, DbData&)

Query the database for a list of object (i.e. non-device) properties for the specified object. The property names are specified by the vector of DbDatum structures. The method returns the properties in the same DbDatum structures. To retrieve the properties use the extract operator >>. Here is an example of how to use the DbData type to specify and extract properties :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));
db->get_property("mymotor", db_data);
float velocity, acceleration;
db_data[0] >> velocity;
db_data[1] >> acceleration;
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.38 void Database::put_property(string, DbData&)

Insert or update a list of properties for the specified object. The property names and their values are specified by the vector of DbDatum structures. Use the insert operator >> to insert the properties into the DbDatum structures. Here is an example of how to insert properties into the database using this method :

```
DbDatum velocity("velocity"), acceleration("acceleration");
DbData db_data;
velocity << 100000.0;
acceleration << 500000.0;
db_data.push_back(velocity);
db_data.push_back(acceleration);
db->put_property("mymotor", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.39 void Database::delete_property(string, DbData&)

Delete a list of properties for the specified object. The property names are specified by the vector of DbDatum structures. Here is an example of how to delete properties from the database using this method :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));
db->delete_property("mymotor", db_data);
```

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.40 vector<DbHistory> Database::get_property_history(string &objname, string &propname)

Get the list of the last 10 modifications of the specifed object property. Note that propname can contain a wildcard character (eg: "prop*").

```
vector<DbHistory> hist;
DbDatum result;
string objname("jlptest");
string propname("test_prop");
hist = db->get_property_history(objname,propname);
// Print the modification history of the specified property
for(int i=0;i<hist.size();i++) {
  cout << "Name:" << hist[i].get_name() << endl;
  cout << "Date:" << hist[i].get_date() << endl;
  if( hist[i].is_deleted() ) {
    cout << "Deleted !" << endl;
  } else {
    hist[i].get_value() >> result;
    for (int j=0; j<result.size(); j++)
      cout << "Value:" << result[j] << endl;
  }
}
```

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.41 void Database::get_device_property(string, DbData&)

Query the database for a list of device properties for the specified object. The property names are specified by the vector of DbDatum structures. The method returns the properties in the same DbDatum structures. To retrieve the properties use the extract operator >>. Here is an example of how to use the DbData type to specify and extract properties :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));
db->get_device_property("id11/motor/1", db_data);
float velocity, acceleration;
db_data[0] >> velocity;
db_data[1] >> acceleration;
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.42 void Database::put_device_property(string, DbData&)

Insert or update a list of properties for the specified device. The property names and their values are specified by the vector of DbDatum structures. Use the insert operator >> to insert the properties into the DbDatum structures. Here is an example of how to insert properties into the database using this method :

```
DbDatum velocity("velocity"), acceleration("acceleration");
DbData db_data;
velocity << 100000.0;
acceleration << 500000.0;
db_data.push_back(velocity);
db_data.push_back(acceleration);
db->put_device_property("id11/motor/1", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.43 void Database::delete_device_property(string, DbData&)

Delete a list of properties for the specified device. The property names are specified by the vector of DbDatum structures. Here is an example of how to delete properties from the database using this method :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));
db->delete_device_property("id11/motor/1", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.44 vector<DbHistory> Database::get_device_property_history(string &devname, string &propname)

Get the list of the last 10 modifications of the specifed device property. Note that propname can contain a wildcard character (eg: "prop*"). An example of usage of a similar function can be found in the documentation of the get_property_history() function.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.45 void Database::get_device_attribute_property(string, DbData&)

Query the database for a list of device attribute properties for the specified object. The attribute names are specified by the vector of DbDatum structures. The method returns all the properties for the specified attributes. The attribute names are returned with the number of properties specified as their value. The first DbDatum element of the returned DbData vector contains the first attribute name and the first attribute property number. The following DbDatum element contains the first attribute property name and property values. To retrieve the properties use the extract operator >>. Here is an example of how to use the DbData type to specify and extract attribute properties :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));

db->get_device_attribute_property("id11/motor/1", db_data);

float vel_max, vel_min, acc_max, acc_min;

for (int i=0;i < db_data.size();)
{
     long nb_prop;
     string &att_name = db_data[i].name;
     db_data[i] >> nb_prop;
     i++;
     for (int k=0;k < nb_prop;k++)
     {
         string &prop_name = db_data[i].name;
         if (att_name == "velocity")
         {
             if (prop_name == "min")
                 db_data[i] >>  vel_min;
             else if (att_name == "max")
                 db_data[i] >> vel_max;
         }
         else
         {
             if (prop_name == "min")
                 db_data[i] >> acc_min;
             else
                 db_data[i] >> acc_max;
         }
         i++;
     }
}
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.46 void Database::put_device_attribute_property(string, DbData&)

Insert or update a list of attribute properties for the specified device. The attribute property names and their values are specified by the vector of DbDatum structures. Use the insert operator >> to insert the properties into the DbDatum structures. Here is an example of how to insert/update properties *min*, *max* for attribute *velocity* and properties *min*, *max* for attribute *acceleration* of device *id11/motor/1* into the database using this method :

```
DbDatum velocity("velocity"), vel_min("min"), vel_max("max");
DbDatum acceleration("acceleration"), acc_min("min"), acc_max("max");
DbData db_data;
velocity << 2;
vel_min << 0.0;
vel_max << 1000000.0;
db_data.push_back(velocity);
db_data.push_back(vel_min);
db_data.push_back(vel_max);
acceleration << 2;
acc_min << 0.0;
acc_max << 8000000;
db_data.push_back(acceleration);
db_data.push_back(acc_min);
db_data.push_back(acc_max);
db->put_device_attribute_property("id11/motor/1", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.47 void Database::delete_device_attribute_property(string, DbData&)

Delete a list of attribute properties for the specified device. The attribute names are specified by the vector of DbDatum structures. Here is an example of how to delete the *unit* property of the *velocity* attribute of the *id11/motor/1* device using this method :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("unit"));
db->delete_device_attribute_property("id11/motor/1", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.48 vector<DbHistory> Database::get_device_attribute_property_history(string &devname, string &attname, string &propname)

Get the list of the last 10 modifications of the specifed device attribute property. Note that propname and attname can contain a wildcard character (eg: "prop*"). An example of usage of a similar function can be found in the documentation of the get_property_history() function.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.49 DbDatum Database::get_class_list(string &wildcard)

Query the database for a list of classes which match the specified wildcard.

```
string wildcard("Motor*");
DbDatum db_datum = db->get_class_list(wildcard);
vector<string> class_list;
db_datum >> class_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.50 DbDatum Database::get_class_property_list(string &classname)

Query the database for a list of properties defined for the specified class.

```
string classname("MyClass");
DbDatum db_datum = db->get_class_property_list(classname);
vector<string> prop_list;
db_datum >> prop_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.51 void Database::get_class_property(string, DbData&)

Query the database for a list of class properties. The property names are specified by the vector of DbDatum structures. The method returns the properties in the same DbDatum structures. To retrieve the properties use the extract operator >>. Here is an example of how to use the DbData type to specify and extract properties :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));
db->get_class_property("StepperMotor", db_data);
float velocity, acceleration;
db_data[0] >> velocity;
db_data[1] >> acceleration;
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.52 void Database::put_class_property(string, DbData&)

Insert or update a list of properties for the specified class. The property names and their values are specified by the vector of DbDatum structures. Use the insert operator >> to insert the properties into the DbDatum structures. Here is an example of how to insert properties into the database using this method :

```
DbDatum velocity("velocity"), acceleration("acceleration");
DbData db_data;
velocity << 100000.0;
acceleration << 500000.0;
db_data.push_back(velocity);
db_data.push_back(acceleration);
db->put_class_property("StepperMotor", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.53 void Database::delete_class_property(string, DbData&)

Delete a list of properties for the specified class. The property names are specified by the vector of DbDatum structures. Here is an example of how to delete properties from the database using this method :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));
db->delete_class_property("StepperMotor", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.54 vector<DbHistory> Database::get_class_property_history(string &classname, string &propname)

Get the list of the last 10 modifications of the specifed class property. Note that propname can contain a wildcard character (eg: "prop*"). An example of usage of a similar function can be found in the documentation of the get_property_history() function.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.55 DbDatum Database::get_class_attribute_list(string &classname,string &wildcard)

Query the database for a list of attributes defined for the specified class which match the specified wildcard.

```
string classname("MyClass");
string wildcard("*");
DbDatum db_datum = db->get_class_attribute_list(classname,wildcard);
vector<string> att_list;
db_datum >> att_list;
```

Exception: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.56 void Database::get_class_attribute_property(string, DbData&)

Query the database for a list of class attribute properties for the specified object. The attribute names are specified by the vector of DbDatum structures. The method returns all the properties for the specified attributes. The attribute names are returned with the number of properties specified as their value. The first DbDatum element of the returned DbData vector contains the first attribute name and the first attribute property number. The following DbDatum element contains the first attribute property name and property values. To retrieve the properties use the extract operator >>. Here is an example of how to use the DbData type to specify and extract attribute properties :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("acceleration"));

db->get_class_attribute_property("StepperMotor", db_data);

float vel_max, vel_min, acc_max, acc_min;

for (int i=0; i< db_data.size(); i++)
{
     long nb_prop;
     string &att_name = db_data[i].name;
     db_data[i] >> nb_prop;
     i++;
     for (int k=0;k < nb_prop;k++)
     {
         string &prop_name = db_data[i].name;
         if (att_name == "velocity")
         {
             if (prop_name == "min")
                 db_data[i] >>  vel_min;
             else if (att_name == "max")
                 db_data[i] >> vel_max;
         }
         else
         {
             if (prop_name == "min")
                 db_data[i] >> acc_min;
             else
                 db_data[i] >> acc_max;
         }
         i++;
     }
}
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.57 void Database::put_class_attribute_property(string, DbData&)

Insert or update a list of attribute properties for the specified class. The attribute property names and their values are specified by the vector of DbDatum structures. Use the insert operator >> to insert the properties into the DbDatum structures. Here is an example of how to insert/update *min*, *max* properties for attribute *velocity* and *min*, *max* properties for attribute *acceleration* properties belonging to class *StepperMotor* into the database using this method :

```
DbDatum velocity("velocity"), vel_min("min"), vel_max("max");
DbDatum acceleration("acceleration"), acc_min("min"), acc_max("max");
DbData db_data;
velocity << 2;
vel_min << 0.0;
vel_max << 1000000.0;
db_data.push_back(velocity);
db_data.push_back(vel_min);
db_data.push_back(vel_max);
acceleration << 2;
acc_min << 0.0;
acc_max << 8000000;
db_data.push_back(acceleration);
db_data.push_back(acc_min);
db_data.push_back(acc_max);
db->put_class_attribute_property("StepperMotor", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.58 void Database::delete_class_attribute_property(string, DbData&)

Delete a list of attribute properties for the specified class. The attribute names are specified by the vector of DbDatum structures. All properties belonging to the listed attributes are deleted. Here is an example of how to delete the *unit* property of the *velocity* attribute of the *StepperMotor* class from the database using this method :

```
DbData db_data;
db_data.push_back(DbDatum("velocity"));
db_data.push_back(DbDatum("unit"));
db->delete_class_attribute_property("StepperMotor", db_data);
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.59 vector<DbHistory> Database::get_class_attribute_property_history(string &devname, string &attname, string &propname)

Get the list of the last 10 modifications of the specifed class attribute property. Note that propname and attname can contain a wildcard character (eg: "prop*"). An example of usage of a similar function can be found in the documentation of the get_property_history() function.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device

6.10.60 void Database::get_alias(string dev_name, string &dev_alias)

Get the device alias name from its name. The device name is specified by *dev_name* and the device alias name is returned in *dev_alias*. If there is no alias defined for the device, a DevFailed exception is thrown.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device (DB_AliasNotDefined)

6.10.61 void Database::get_device_alias(string dev_alias, string &dev_name)

Get the device name from an alias. The device alias is specified by *dev_alias* and the device name is returned in *dev_name*. If there is no device with the given alias, a DevFailed exception is thrown.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_DeviceNotDefined)*

6.10.62 void Database::get_attribute_alias(string attr_alias, string &attr_name)

Get the full attribute name from an alias. The attribute alias is specified by *attr_alias* and the full attribute name is returned in *attr_name*. If there is no attribute with the given alias, a DevFailed exception is thrown.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.63 void Database::put_attribute_alias(string &att_name, string &alias_name)

Set an alias for an attribute name. The attribute alias is specified by alias_name and the attribute name is specifed by att_name. If the given alias already exists, a DevFailed exception is thrown.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)

6.10.64 void Database::delete_attribute_alias(string &alias_name)

Remove the alias associated to an attribute name.

Exceptions: ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)

6.10.65 DbDatum Database::get_device_alias_list(string &filter)

Get device alias list. The parameter *alias* is a string to filter the alias list returned. Wildcard (*) is supported. For instance, if the string alias passed as the method parameter is initialised with only the * character, all the defined device alias will be returned. The DbDatum returned by this method is initialised with an array of strings and must be extracted into a vector<string>. If there is no alias with the given filter, the returned array will have a 0 size.

```
DbData db_data;
string filter("*");
db_data = db->get_device_alias_list(filter);
vector<string> al_list;
db_data >> al_list;
cout << al_list.size() << " device alias defined in db" << endl;
for (int i=0;i < al_list.size();i++)
    cout << "alias = " << al_list[i] << endl;
```

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.66 DbDatum Database::get_attribute_alias_list(string &filter)

Get attribute alias list. The parameter *alias* is a string to filter the alias list returned. Wildcard (*) is supported. For instance, if the string alias passed as the method parameter is initialised with only the * character, all the defined attribute alias will be returned. The DbDatum returned by this method is initialised with an array of strings and must be extracted into a vector<string>. If there is no alias with the given filter, the returned array will have a 0 size.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.67 void Database::put_device_alias(string &dev_name,string &alias_name)

Create a device alias. Alias name has to be uniq within a Tango control system and you will receive an exception if the alias is already defined.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.10.68 void Database::delete_device_alias(string &alias_name)

Delete a device alias.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11 Tango::DbDevice

A database object for a device which can be used to query or modify properties, import and export information for a device. This class provides an easy to use interface for device objects in the database. It uses the methods of the Database class therefore the reader is referred to these for the exact calling syntax and examples. The following methods are defined for the DbDevice class :

6.11.1 DbDevice::DbDevice(string &)

A constructor for a DbDevice object for a device in the TANGO database specified by the TANGO_HOST environment variable.

6.11.2 DbDevice::DbDevice(string &, Database *)

A constructor for a DbDevice object for the device in the specified database. This method reuses the Database supplied by the programmer.

6.11.3 DbDevImportInfo DbDevice::import_device()

Query the database for the import info of this device. Returns a DbDevImportInfo structure.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.4 void DbDevice::export_device(DbDevExportInfo&)

Update the export info for this device in the database.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.5 void DbDevice::add_device(DbDevInfo&)

Add/Update this device to the database.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.6 void DbDevice::delete_device()

Delete this device from the database.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.7 void DbDevice::get_property(DbData&)

Query the database for the list of properties of this device. See Database::get_device_property() for an example of how to specify and retrieve the properties.
Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.8 void DbDevice::put_property(DbData&)

Update the list of properties for this device in the database. See Database::put_device_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.9 void DbDevice::delete_property(DbData&)

Delete the list of specified properties for this device in the database. See Database::delete_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.10 void DbDevice::get_attribute_property(DbData&)

Query the database for the list of attribute properties of this device. See Database::get_device_attribute_property() for an example of how to specify and retrieve the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.11 void DbDevice::put_attribute_property(DbData&)

Update the list of attribute properties for this device in the database. See Database::put_device_attribute_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.11.12 void DbDevice::delete_attribute_property(DbData&)

Delete all properties for the list of specified attributes for this device in the database. See Database::delete_device_attribute_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.12 Tango::DbClass

A database object for a class which can be used to query or modify class properties.

6.12.1 DbClass::DbClass(string)

A constructor for a DbClass object for a class in the TANGO database specified by the TANGO_HOST environment variable.

6.12.2 DbClass::DbClass(string, Database *)

A constructor for a DbClass object for the class in the specified database. This method reuses the Database supplied by the programmer.

6.12.3 void DbClass::get_property(DbData&)

Query the database for the list of properties of this class. See Database::get_class_property() for an example of how to specify and retrieve the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.12.4 void DbClass::put_property(DbData&)

Update the list of properties for this class in the database. See Database::put_class_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.12.5 void DbClass::delete_property(DbData&)

Delete the list of specified properties for this class in the database. See Database::delete_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.12.6 void DbClass::get_attribute_property(DbData&)

Query the database for the list of attribute properties of this class. See Database::get_class_attribute_property() for an example of how to specify and retrieve the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.12.7 void DbClass::put_attribute_property(DbData&)

Update the list of attribute properties for this class in the database. See Database::put_class_attribute_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.12.8 void DbClass::delete_attribute_property(DbData&)

Delete all properties for the list of specified attributes for this class in the database. See Database::delete_class_attribute_property() for an example of how to specify the properties.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.13 Tango::DbServer

A database object for a device server which can be used to query or modify server database information.

6.13.1 DbServer::DbServer(string)

A constructor for a DbServer object for a server in the TANGO database specified by the TANGO_HOST environment variable.

6.13.2 DbServer::DbServer(string, Database *)

A constructor for a DbServer object for the server in the specified database. This method reuses the Database supplied by the programmer.

6.13.3 void DbServer::add_server(DbDevInfos &)

Add a group of devices to the database. The device names, server names and classes are specified in the vector of DbDevInfo structures.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.13.4 void DbServer::delete_server()

Delete the device server and its associated devices from the database.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.13.5 void DbServer::export_server(DbDevExportInfos &)

Export a group of device to the database. The device names, IOR, class, server name, pid etc. are specified in the vector of DbDevExportInfo structures.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.13.6 void DbServer::unexport_server()

Mark all the devices exported by the server as un exported.

Exceptions: *ConnectionFailed, CommunicationFailed, DevFailed from device (DB_SQLError)*

6.14 Tango::DbDatum

A single database value which has a name, type, address and value and methods for inserting and extracting C++ native types. This is the fundamental type for specifying database properties. Every property has a name and has one or more values associated with it. The values can be inserted and extracted using the operators << and >> respectively. A status flag indicates if there is data in the DbDatum object or not. An additional flag allows the user to activate exceptions.

6.14.1 Operators

The insert and extract operators are specified for the following C++ types :

1. boolean
2. unsigned char
3. short
4. unsigned short
5. DevLong
6. DevULong
7. DevLong64
8. DevULong64
9. float
10. double
11. string
12. char* (insert only)
13. const char *
14. vector<string>
15. vector<short>
16. vector<unsigned short>
17. vector<DevLong>
18. vector<DevULong>
19. vector<DevLong64>

20. vector<DevULong64>
21. vector<float>
22. vector<double>

Here is an example of creating, inserting and extracting some DbDatum types :

```
DbDatum my_short("my_short"), my_long("my_long"), my_string("my_string");
DbDatum my_float_vector("my_float_vector"), my_double_vector("my_double_vector");
string a_string;
short a_short;
DevLong a_long;
vector<float> a_float_vector;
vector<double> a_double_vector;
my_short << 100; // insert a short
my_short >> a_short; // extract a short
my_long << 1000; // insert a DevLong
my_long >> a_long; // extract a long
my_string << string("estas lista a bailar el tango ?"); // insert a string
my_string >> a_string; // extract a string
my_float_vector << a_float_vector // insert a vector of floats
my_float_vector >> a_float_vector; // extract a vector of floats
my_double_vector << a_double_vector; // insert a vector of doubles
my_double_vector >> a_double_vector; // extract a vector of doubles
```

Exception: WrongData if requested

6.14.2 bool DbDatum::is_empty()

is_empty() is a boolean method which trues true or false depending on whether the DbDatum object contains data or not. It can be used to test whether a property is defined in the database or not e.g.

```
sl_props.push_back(parity_prop);
dbase->get_device_property(device_name, sl_props);
if (! parity_prop.is_empty())
{
    parity_prop >> parity;
}
else
{
    cout << device_name << " has no parity defined in database !" << endl;
}
```

Exception: WrongData if requested

6.14.3 void DbDatum::exceptions(bitset<DbDatum::numFlags>)

Is a method which allows the user to switch on/off exception throwing for trying to extract data from an empty DbDatum object. The default is to not throw exception. The following flags are supported :

1. **isempty_flag** - throw a WrongData exception (reason = API_EmptyDbDatum) if user tries to extract data from an empty DbDatum object
2. **wrongtype_flag** - throw a WrongData exception (reason = API_IncompatibleArgumentType) if user tries to extract data with a type different than the type used for insertion

6.14.4 bitset<DbDatum::numFlags> exceptions()

Returns the whole exception flags.

6.14.5 void DbDatum::reset_exceptions(DbDatum::except_flags fl)

Resets one exception flag

6.14.6 void DbDatum::set_exceptions(DbDatum::except_flags fl)

Sets one exception flag

The following is an example of how to use these exceptions related methods

```
DbDatum da;

bitset<DbDatum::numFlags> bs = da.exceptions();
cout << "bs = " << bs << endl;

da.set_exceptions(DbDatum::wrongtype_flag);
bs = da.exceptions();

cout << "bs = " << bs << endl;
```

6.15 Tango::DbData

A vector of Tango::DbDatum structures. DbData is used to send or return one or more database properties or information. It is the standard input and output type for all methods which query and/or update properties in the database.

6.16 Exception

All the exception thrown by this API are Tango::DevFailed exception. This exception is a variable length array of Tango::DevError type. The Tango::DevError type is a four fields structure. These fields are :

1. A string describing the error type. This string replaces an error code and allows a more easy management of include files. This field is called **reason**
2. A string describing in plain text the reason of the error. This field is called **desc**
3. A string giving the name of the method which thrown the exception. This field is named **origin**

4. The error severity. This is an enumeration with three values which are WARN, ERR or PANIC. Its name is **severity**

This is a variable length array in order to transmit to the client what is the primary error reason. The sequence element 0 describes the primary error. An exception class hierarchy has been implemented within the API to ease API programmers task. All the exception classes inherits from the Tango::DevFailed class. Except for the *NamedDevFaildeList* exception class, they don't add any new fields to the exception, they just allow easy "catching". Exception classes thrown only by the API layer are :

- ConnectionFailed
- CommunicationFailed
- WrongNameSyntax
- NonDbDevice
- WrongData
- NonSupportedFeature
- AsynCall
- AsynReplyNotArrived
- EventSystemFailed
- NamedDevFailedList
- DeviceUnlocked

On top of these classes, exception thrown by the device (Tango::DevFailed exception) are directly passed to the client.

6.16.1 The ConnectionFailed exception

This exception is thrown when a problem occurs during the connection establishment between the application and the device. The API is stateless. This means that DeviceProxy constructors filter most of the exception except for cases described in the following table.

Method name	device type	error type	Level	reason
DeviceProxy constructor	with database	TANGO_HOST not set	0	API_TangoHostNotSet
		Device not defined in db or Alias not defined in db	0	DB_DeviceNotDefined
			1	API_CommandFailed
			2	API_DeviceNotDefined
	with database specified in dev name	Database server not running	0	API_CorbaException
			1	API_CantConnectToDatabase
	without database	Server running but device not defined in server	0	API_CorbaException
			1	API_DeviceNotExported
AttributeProxy constructor	with database	TANGO_HOST not set	0	API_TangoHostNotSet
		Device not defined in db	0	DB_DeviceNotDefined
			1	API_CommandFailed
			2	API_DeviceNotDefined
		Alias not defined in db	0	DB_SQLError
			1	API_CommandFailed
			2	API_AliasNotDefined
	with database specified in dev name	Database server not running	0	API_CorbaException
			1	API_CantConnectToDatabase

DeviceProxy or AttributeProxy method call (except cmd_inout read_attribute)	without database	Server not running	0	API_CorbaException
			1	API_ServerNotRunning
	with database	Server not running	0	API_DeviceNotExported
		Dead server	0	API_CorbaException
			1	API_CantConnectToDevice
		Dead database server when reconnection needed	0	API_CorbaException
			1	API_CantConnectToDatabase
DeviceProxy cmd_inout and read_attribute or AttributeProxy read and write	without database	Server not running	0	API_CorbaException
			1	API_ServerNotRunning
			2	API_CommandFailed
	with database	Server not running	0	API_DeviceNotExported
			1	API_CommandFailed
		Dead server	0	API_CorbaException
			1	API_CantConnectToDevice
			2	API_CommandFailed or API_AttributeFailed
		Dead database server when reconnection needed	0	API_CorbaException
			1	API_CantConnectToDatabase
			2	API_CommandFailed

The desc DevError structure field allows a user to get more precise information. These informations are :

DB_DeviceNotDefined The name of the device not defined in the database

API_CommandFailed The device and command name

API_CantConnectToDevice The device name

API_CorbaException The name of the CORBA exception, its reason, its locality, its completed flag and its minor code

API_CantConnectToDatabase The database server host and its port number

API_DeviceNotExported The device name

6.16.2 The CommunicationFailed exception

This exception is thrown when a communication problem is detected during the communication between the client application and the device server. It is a two levels Tango::DevError structure. In case of time-out, the DevError structures fields are:

Level	Reason	Desc	Severity
0	API_CorbaException	CORBA exception fields translated into a string	ERR
1	API_DeviceTimedOut	String with time-out value and device name	ERR

For all other communication errors, the DevError structures fields are:

Level	Reason	Desc	Severity
0	API_CorbaException	CORBA exception fields translated into a string	ERR
1	API_CommunicationFailed	String with device, method, command/attribute name	ERR

6.16.3 The WrongNameSyntax exception

This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_UnsupportedProtocol This error occurs when trying to build a DeviceProxy or an AttributeProxy instance for a device with an unsupported protocol. Refer to the appendix on device naming syntax to get the list of supported database modifier

API_UnsupportedDBaseModifier This error occurs when trying to build a DeviceProxy or an AttributeProxy instance for a device/attribute with a database modifier unsupported. Refer to the appendix on device naming syntax to get the list of supported database modifier

API_WrongDeviceNameSyntax This error occurs for all the other error in device name syntax. It is thrown by the DeviceProxy class constructor.

API_WrongAttributeNameSyntax This error occurs for all the other error in attribute name syntax. It is thrown by the AttributeProxy class constructor.

API_WrongWildcardUsage This error occurs if there is a bad usage of the wildcard character

6.16.4 The NonDbDevice exception

This exception has only one level of Tango::DevError structure. The reason field is set to API_NonDatabaseDevice. This exception is thrown by the API when using the DeviceProxy or AttributeProxy class database access for non-database device.

6.16.5 The WrongData exception

This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_EmptyDbDatum This error occurs when trying to extract data from an empty DbDatum object

API_IncompatibleArgumentType This error occurs when trying to extract data with a type different than the type used to send the data

API_EmptyDeviceAttribute This error occurs when trying to extract data from an empty DeviceAttribute object

API_IncompatibleAttrArgumentType This error occurs when trying to extract attribute data with a type different than the type used to send the data

API_EmptyDeviceData This error occurs when trying to extract data from an empty DeviceData object

API_IncompatibleCmdArgumentType This error occurs when trying to extract command data with a type different than the type used to send the data

6.16.6 The NonSupportedFeature exception

This exception is thrown by the API layer when a request to a feature implemented in Tango device interface release n is requested for a device implementing Tango device interface n-x. There is one possible value for the reason field which is API_UnsupportedFeature.

6.16.7 The AsynCall exception

This exception is thrown by the API layer when a the asynchronous model id badly used. This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_BadAsynPollId This error occurs when using an asynchronous request identifier which is not valid any more.

API_BadAsyn This error occurs when trying to fire callback when no callback has been previously registered

API_BadAsynReqType This error occurs when trying to get result of an asynchronous request with an asynchronous request identifier returned by a non-coherent asynchronous request (For instance, using the asynchronous request identifier returned by a *command_inout_asynch()* method with a *read_attribute_reply()* attribute).

6.16.8 The AsynReplyNotArrived exception

This exception is thrown by the API layer when:

- a request to get asynchronous reply is made and the reply is not yet arrived
- a blocking wait with timeout for asynchronous reply is made and the timeout expired.

There is one possible value for the reason field which is API_AsynReplyNotArrived.

6.16.9 The EventSystemFailed exception

This exception is thrown by the API layer when subscribing or unsubscribing from an event failed. This exception has only one level of Tango::DevError structure. The possible value for the reason field are :

API_NotificationServiceFailed This error occurs when the *subscribe_event()* method failed trying to access the CORBA notification service

API_EventNotFound This error occurs when you are using an incorrect event_id in the *unsubscribe_event()* method

API_InvalidArgs This error occurs when NULL pointers are passed to the subscribe or unsubscribe event methods

API_MethodArgument This error occurs when trying to subscribe to an event which has already been subsribed to

API_DSFailedRegisteringEvent This error means that the device server to which the device belongs to failed when it tries to register the event. Most likely, it means that there is no event property defined

API_EventNotFound Occurs when using a wrong event identifier in the *unsubscribe_event* method

6.16.10 The NamedDevFailedList exception

This exception is only thrown by the *DeviceProxy::write_attributes()* method. In this case, it is necessary to have a new class of exception to transfer the error stack for several attribute(s) which failed during the writing. Therefore, this exception class contains for each attributes which failed :

- The name of the attribute
- Its index in the vector passed as argumen tof the write_attributes() method
- The error stack as described in 6.16

6.16.10.1 long NamedDevFailedList::get_faulty_attr_nb()

Returns the number of attributes which failed during the write_attribute call.

6.16.10.2 vector<NamedDevFailed> NamedDevErrorList::err_list

Public data member of the NamedDevFailedList. There is one element in this vector for each attribute which failed during its writing.

6.16.10.3 string NamedDevFailed::name

Public data member of the NamedDevFailed class. It contains the name of the attribute which failed.

6.16.10.4 long NamedDevFailed::idx_in_call

Public data member of the NamedDevFailed class. It contains the index in the write_attributes method parameter vector of the attribute which failed.

6.16.10.5 DevErrorList NamedDevFailed::err_stack

Public data member of the NamedDevFailed class. This is the error stack.

The following piece of code is an example of how to use this class exception

```
catch (Tango::NamedDevFailed &e)
{
   long nb_faulty = e.get_faulty_attr_nb();
   for (long i = 0;i < nb_faulty;i++)
   {
       cout << "Attribute " << e.err_list[i].name << " failed!" << endl;
       for (long j = 0;j < e.err_list[i].err_stack.length();j++)
       {
           cout << "Reason [" << j << "] = " << e.err_list[i].err_stack[j].reason;
           cout << "Desc [" << j << "] = " << e.err_list[i].err_stack[j].desc;
       }
   }
}
```

This exception inherits from Tango::DevFailed. It is possible to catch it with a "catch DevFailed" catch block. In this case, like any other DevFailed exception, there is only one error stack. This stack is initialised with the name of all the attributes which failed in its "reason" field.

6.16.11 The DeviceUnlocked exception

This exception is thrown by the API layer when a device locked by the process has been unlocked by an admin client. This exception has two levels of Tango::DevError structure. There is only possible value for the reason field which is

API_DeviceUnlocked The device has been unlocked by another client (administration client)

The first level is the message reported by the Tango kernel from the server side. The second layer is added by the client API layer with informations on which API call generates the exception and device name.

6.17 Reconnection and exception

The Tango API automatically manages re-connection between client and server in case of communication error during a network access between a client and a server. The transparency reconnection mode allows a user to be (or not be) informed that automatic reconnection took place. If the transparency reconnection mode is not set, when a communication error occurs, an exception is returned to the caller and the connection is internally marked as bad. On the next try to contact the device, the API will try to re-build the network connection. If the transparency reconnection mode is set, the API will try to re-build the network connection has soon as the communication error occurs and the caller is not informed. Several cases are possible. They are summarized in the following table:

Case	Server state	call nb	exception (transparency false)	exception (transparency true)
Server killed and re-started	Server killed before call n	n	CommunicationFailed	ConnectionFailed
	down	n+1	ConnectionFailed(2 levels)	idem
	down	n + 2	idem	idem
	Running	n + x	No exception	No exception
Server died and re-started	Server died before call n	n	CommunicationFailed	ConnectionFailed
	died	n + 1	ConnectionFailed (3 levels)	idem
	died	n + 2	idem	idem
	Running	n + x	No exception	No exception
Server killed and re-started	Server killed and re-started before call n	n	CommunicationFailed	No exception
	Running	n+x	No exception	No exception
Server died and re-started	Server died and re-started before call n	n	CommunicationFailed	No exception
	Running	n + x	No exception	No exception

Please note that the timeout case is managed differently because it will not enter the re-connection system. The transparency reconnection mode is set by default to true for Tango version 5.5!

Chapter 7

TangoATK Programmer's Guide

This chapter is only a brief Tango ATK (Application ToolKit) programmer's guide. You can find a reference guide with a full description of TangoATK classes and methods in the ATK JavaDoc [17].

A tutorial document [22] is also provided and includes the detailed description of the ATK architecture and the ATK components. In the ATK Tutorial [22] you can find some code examples and also Flash Demos which explain how to start using Tango ATK.

7.1 Introduction

This document describes how to develop applications using the Tango Application Toolkit, TangoATK for short. It will start with a brief description of the main concepts behind the toolkit, and then continue with more practical, real-life examples to explain key parts.

7.1.1 Assumptions

The author assumes that the reader has a good knowledge of the Java programming language, and a thorough understanding of object-oriented programming. Also, it is expected that the reader is fluent in all aspects regarding Tango devices, attributes, and commands.

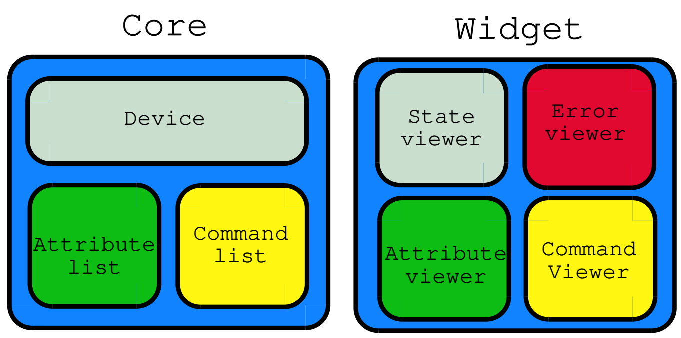
7.2 The key concepts of TangoATK

TangoATK was developed with these goals in mind

- TangoATK should help minimize development time
- TangoATK should help minimize bugs in applications
- TangoATK should support applications that contain attributes and commands from several different devices.
- TangoATK should help avoid code duplication.

Since most Tango-applications were foreseen to be displayed on some sort of graphic terminal, TangoATK needed to provide support for some sort of graphic building blocks. To enable this, and since the toolkit was to be written in Java, we looked to Swing to figure out how to do this.

Swing is developed using a variant over a design-pattern the Model-View-Controller (MVC) pattern called *model-delegate*, where the view and the controller of the MVC-pattern are merged into one object.



This pattern made the choice of labor division quite easy: all non-graphic parts of TangoATK reside in the packages beneath `fr.esrf.tangoatk.core`, and anything remotely graphic are located beneath `fr.esrf.tangoatk.widget`. More on the content and organization of this will follow.

The communication between the non-graphic and graphic objects are done by having the graphic object registering itself as a listener to the non-graphic object, and the non-graphic object emmiting events telling the listeners that its state has changed.

7.2.1 Minimize development time

For TangoATK to help minimize the development time of graphic applications, the toolkit has been developed along two lines of thought

- Things that are needed in most applications are included, eg `Splash`, a splash window which gives a graphical way for the application to show the progress of a long operation. The splash window is moslty used in the startup phase of the application.
- Building blocks provided by TangoATK should be easy to use and follow certain patterns, eg every graphic widget has a `setModel` method which is used to connect the widget with its non-graphic model.

In addition to this, TangoATK provides a framework for error handling, something that is often a time consuming task.

7.2.2 Minimize bugs in applications

Together with the Tango API, TangoATK takes care of most of the hard things related to programming with Tango. Using TangoATK the developer can focus on developing her application, not on understanding Tango.

7.2.3 Attributes and commands from different devices

To be able to create applications with attributes and commands from different devices, it was decided that the central objects of TangoATK were not to be the device, but rather the *attributes and the commands*. This will certainly feel a bit awkward at first, but trust me, the design holds.

For this design to be feasible, a structure was needed to keep track of the commands and attributes, so the *command-list and the attribute-list* was introduced. These two objects can hold commands and attributes from any number of devices.

7.2.4 Avoid code duplication

When writing applications for a control-system without a framework much code is duplicated. Anything from simple widgets for showing numeric values to error handling has to be implemented each time. TangoATK supplies a number of frequently used widgets along with a framework for connecting these widgets with their non-graphic counterparts. Because of this, the developer only needs to write the *glue* - the code which connects these objects in a manner that suits the specified application.

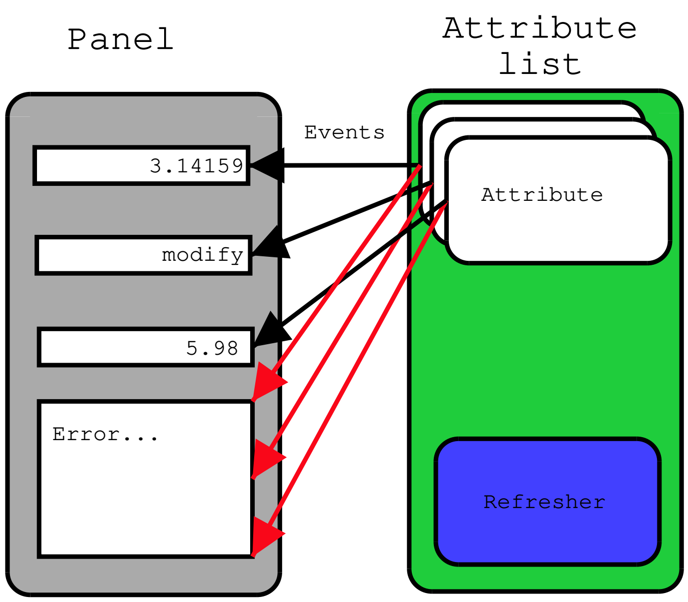
7.3 The real getting started

Generally there are two kinds of end-user applications: Applications that only know how to treat one device, and applications that treat many devices.

7.3.1 Single device applications

Single device applications are quite easy to write, even with a gui. The following steps are required

1. Instantiate an AttributeList and fill it with the attributes you want.
2. Instantiate a CommandList and fill it with the commands you want.
3. Connect the whole *AttributeList* with a *list viewer* and / or each *individual attribute* with an *attribute viewer*.
4. Connect the whole *CommandList* to a *command list viewer* and / or connect each *individual command* in the command list with a *command viewer*.



The following program (FirstApplication) shows an implementation of the list mentioned above. It should be rather self-explanatory with the comments.

```
package examples;

import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;
import javax.swing.JMenu;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.BorderLayout;

import fr.esrf.tangoatk.core.AttributeList;
import fr.esrf.tangoatk.core.ConnectionException;
```

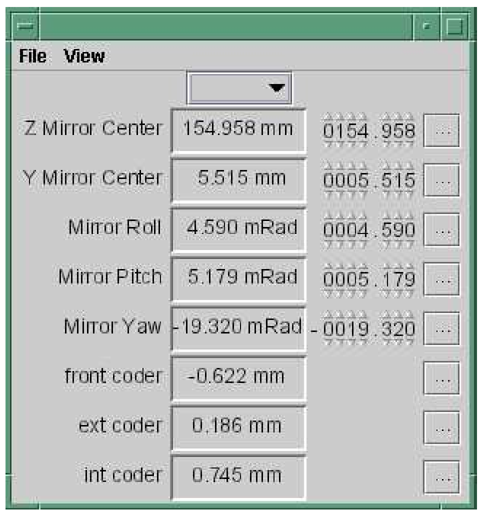
```
import fr.esrf.tangoatk.core.CommandList;
import fr.esrf.tangoatk.widget.util.ErrorHistory;
import fr.esrf.tangoatk.widget.util.ATKGraphicsUtils;
import fr.esrf.tangoatk.widget.attribute.ScalarListViewer;
import fr.esrf.tangoatk.widget.command.CommandComboViewer;

public class FirstApplication extends JFrame
{
JMenuBar menu;                    // So that our application looks
                                  // halfway decent.
AttributeList attributes;         // The list that will contain our
                                  // attributes
CommandList commands;             // The list that will contain our
                                  // commands
ErrorHistory errorHistory;        // A window that displays errors
ScalarListViewer sListViewer;     // A viewer which knows how to
                                  // display a list of scalar attributes.
                                  // If you want to display other types
                                  // than scalars, you'll have to wait
                                  // for the next example.
CommandComboViewer commandViewer; // A viewer which knows how to display
                                  // a combobox of commands and execute
                                  // them.
String device;                    // The name of our device.

public FirstApplication()
{
   // The swing stuff to create the menu bar and its pulldown menus
   menu = new JMenuBar();
   JMenu fileMenu = new JMenu();
   fileMenu.setText("File");
   JMenu viewMenu = new JMenu();
   viewMenu.setText("View");
   JMenuItem quitItem = new JMenuItem();
   quitItem.setText("Quit");
   quitItem.addActionListener(new
      java.awt.event.ActionListener()
      {
       public void
       actionPerformed(ActionEvent evt)
       {quitItemActionPerformed(evt);}
      });
   fileMenu.add(quitItem);
   JMenuItem errorHistItem = new JMenuItem();
   errorHistItem.setText("Error History");
   errorHistItem.addActionListener(new
           java.awt.event.ActionListener()
           {
            public void
            actionPerformed(ActionEvent evt)
            {errHistItemActionPerformed(evt);}
           });
   viewMenu.add(errorHistItem);
```

```
      menu.add(fileMenu);
      menu.add(viewMenu);
      //
      // Here we create ATK objects to handle attributes, commands and errors.
      //
      attributes = new AttributeList();
      commands = new CommandList();
      errorHistory = new ErrorHistory();
      device = "id14/eh3_mirror/1";
      sListViewer = new ScalarListViewer();
      commandViewer = new CommandComboViewer();
   //
   // A feature of the command and attribute list is that if you
   // supply an errorlistener to these lists, they'll add that
   // errorlistener to all subsequently created attributes or
   // commands. So it is important to do this _before_ you
   // start adding attributes or commands.
   //

      attributes.addErrorListener(errorHistory);
      commands.addErrorListener(errorHistory);

   //
   // Sometimes we're out of luck and the device or the attributes
   // are not available. In that case a ConnectionException is thrown.
   // This is why we add the attributes in a try/catch
   //

      try
      {

   //
   // Another feature of the attribute and command list is that they
   // can add wildcard names, currently only `*' is supported.
   // When using a wildcard, the lists will add all commands or
   // attributes available on the device.
   //
      attributes.add(device + "/*");
      }
      catch (ConnectionException ce)
      {
         System.out.println("Error fetching " +
                            "attributes from " +
                            device + " " + ce);
      }

   //
   // See the comments for attributelist
   //

      try
      {
         commands.add(device + "/*");
      }
```

```
      catch (ConnectionException ce)
      {
         System.out.println("Error fetching " +
                            "commands from " +
                            device + " " + ce);
      }

   //
   // Here we tell the scalarViewer what it's to show. The
   // ScalarListViewer loops through the attribute-list and picks out
   // the ones which are scalars and show them.
   //
      sListViewer.setModel(attributes);

   //
   // This is where the CommandComboViewer is told what it's to
   // show. It knows how to show and execute most commands.
   //

      commandViewer.setModel(commands);

   //
   // add the menubar to the frame
   //

      setJMenuBar(menu);

   //
   // Make the layout nice.
   //

      getContentPane().setLayout(new BorderLayout());
      getContentPane().add(commandViewer, BorderLayout.NORTH);
      getContentPane().add(sListViewer, BorderLayout.SOUTH);

   //
   // A third feature of the attributelist is that it knows how
   // to refresh its attributes.
   //

      attributes.startRefresher();

   //
   // JFrame stuff to make the thing show.
   //

      pack();
      ATKGraphicsUtils.centerFrameOnScreen(this); //ATK utility to center window
      setVisible(true);
      }

      public static void main(String [] args)
      {
         new FirstApplication();
```

```
    }
    public void quitItemActionPerformed(ActionEvent evt)
    {
       System.exit(0);
    }
    public void errHistItemActionPerformed(ActionEvent evt)
    {
       errorHistory.setVisible(true);
    }
}
```

The program should look something like this (depending on your platform and your device)



7.3.2 Multi device applications

Multi device applications are quite similar to the single device applications, the only difference is that it does not suffice to add the attributes by wildcard, you need to add them explicitly, like this:

```
try
{
    // a StringScalar attribute from the device one
   attributes.add("jlp/test/1/att_cinq");
   // a NumberSpectrum attribute from the device one
   attributes.add("jlp/test/1/att_spectrum");
   // a NumberImage attribute from the device two
   attributes.add("sr/d-ipc/id25-1n/Image");
}
catch (ConnectionException ce)
{
   System.out.println("Error fetching " +
       "attributes" + ce);
}
```

The same goes for commands.

7.3.3 More on displaying attributes

So far, we've only considered scalar attributes, and not only that, we've also cheated quite a bit since we just passed the attribute list to the `fr.esrf.tangoatk.widget.attribute.ScalarListViewer` and let it do all the magic. The attribute list viewers are only available for scalar attributes (NumberScalarListViewer and ScalarListViewer). If you have one or several spectrum or image attributes you must connect each spectrum or image attribute to it's corresponding attribute viewer individually. So let's take a look at how you can connect individual attributes (and not a whole attribute list) to an individual attribute viewer (and not to an attribute list viewer).

7.3.3.1 Connecting an attribute to a viewer

Generally it is done in the following way:

1. You retrieve the attribute from the attribute list
2. You instantiate the viewer
3. Your call the `setModel` method on the viewer with the attribute as argument.
4. You add your viewer to some panel

The following example (SecondApplication), is a Multi-device application. Since this application uses individual attribute viewers and not an attribute list viewer, it shows an implementation of the list mentioned above.

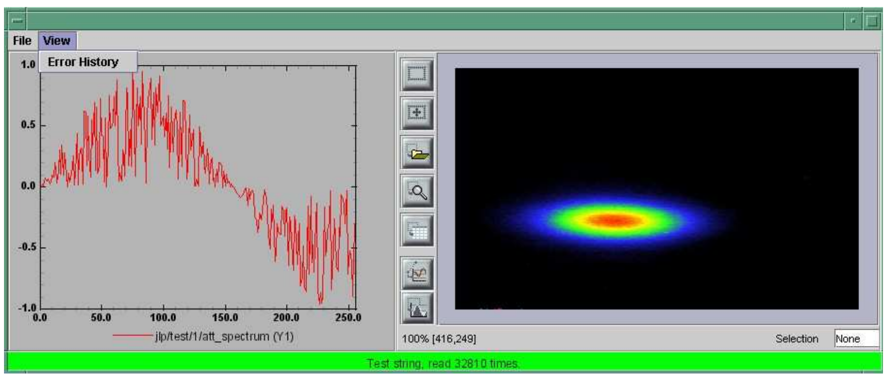
```
package examples;

import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;
import javax.swing.JMenu;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.BorderLayout;
import java.awt.Color;

import fr.esrf.tangoatk.core.AttributeList;
import fr.esrf.tangoatk.core.ConnectionException;

import fr.esrf.tangoatk.core.IStringScalar;
import fr.esrf.tangoatk.core.INumberSpectrum;
import fr.esrf.tangoatk.core.INumberImage;
import fr.esrf.tangoatk.widget.util.ErrorHistory;
import fr.esrf.tangoatk.widget.util.Gradient;
import fr.esrf.tangoatk.widget.util.ATKGraphicsUtils;
import fr.esrf.tangoatk.widget.attribute.NumberImageViewer;
import fr.esrf.tangoatk.widget.attribute.NumberSpectrumViewer;
import fr.esrf.tangoatk.widget.attribute.SimpleScalarViewer;
public class SecondApplication extends JFrame
{
     JMenuBar            menu;
     AttributeList       attributes;   // The list that will contain our attributes
     ErrorHistory        errorHistory; // A window that displays errors
```

```
IStringScalar        ssAtt;
INumberSpectrum      nsAtt;
INumberImage         niAtt;
public SecondApplication()
{
   // Swing stuff to create the menu bar and its pulldown menus
   menu = new JMenuBar();
   JMenu fileMenu = new JMenu();
   fileMenu.setText("File");
   JMenu viewMenu = new JMenu();
   viewMenu.setText("View");
   JMenuItem quitItem = new JMenuItem();
   quitItem.setText("Quit");
   quitItem.addActionListener(new java.awt.event.ActionListener()
                              {
                               public void actionPerformed(ActionEvent evt)
                               {quitItemActionPerformed(evt);}
                              });
   fileMenu.add(quitItem);
   JMenuItem errorHistItem = new JMenuItem();
   errorHistItem.setText("Error History");
   errorHistItem.addActionListener(new java.awt.event.ActionListener()
           {
            public void actionPerformed(ActionEvent evt)
            {errHistItemActionPerformed(evt);}
           });
   viewMenu.add(errorHistItem);
   menu.add(fileMenu);
   menu.add(viewMenu);
 //
 // Here we create TangoATK objects to view attributes and errors.
 //
   attributes = new AttributeList();
   errorHistory = new ErrorHistory();
 //
 // We create a SimpleScalarViewer, a NumberSpectrumViewer and
 // a NumberImageViewer, since we already knew that we were
 // playing with a scalar attribute, a number spectrum attribute
 // and a number image attribute this time.
 //
 SimpleScalarViewer     ssViewer = new SimpleScalarViewer();
   NumberSpectrumViewer   nSpectViewer = new NumberSpectrumViewer();
   NumberImageViewer      nImageViewer = new NumberImageViewer();
   attributes.addErrorListener(errorHistory);
//
// The attribute (and command) list has the feature of returning the last
// attribute that was added to it. Just remember that it is returned as an
// IEntity object, so you need to cast it into a more specific object, like
// IStringScalar, which is the interface which defines a string scalar
//
  try
   {
      ssAtt = (IStringScalar) attributes.add("jlp/test/1/att_cinq");
      nsAtt = (INumberSpectrum) attributes.add("jlp/test/1/att_spectrum");
```

```
         niAtt = (INumberImage) attributes.add("sr/d-ipc/id25-1n/Image");
      }
      catch (ConnectionException ce)
      {
         System.out.println("Error fetching one of the attributes  "+" " + ce);
         System.out.println("Application Aborted.");
         System.exit(0);
      }
      //
      // Pay close attention to the following three lines!! This is how it's done!
      // This is how it's always done! The setModel method of any viewer takes care
     // of connecting the viewer to the attribute (model) it's in charge of displaying.
     // This is the way to tell each viewer what (which attribute) it has to show.
     // Note that we use a viewer adapted to each type of attribute
     //
      ssViewer.setModel(ssAtt);
      nSpectViewer.setModel(nsAtt);
      nImageViewer.setModel(niAtt);
   //
      nSpectViewer.setPreferredSize(new java.awt.Dimension(400, 300));
      nImageViewer.setPreferredSize(new java.awt.Dimension(500, 300));
      Gradient  g = new Gradient();
      g.buidColorGradient();
      g.setColorAt(0,Color.black);
      nImageViewer.setGradient(g);
      nImageViewer.setBestFit(true);
      //
      // Add the viewers into the frame to show them
      //
      getContentPane().setLayout(new BorderLayout());
      getContentPane().add(ssViewer, BorderLayout.SOUTH);
      getContentPane().add(nSpectViewer, BorderLayout.CENTER);
      getContentPane().add(nImageViewer, BorderLayout.EAST);
      //
      // To have the attributes values refreshed we should start the
      // attribute list's refresher.
      //
      attributes.startRefresher();
      //
      // add the menubar to the frame
      //
      setJMenuBar(menu);
      //
      // JFrame stuff to make the thing show.
      //
      pack();
      ATKGraphicsUtils.centerFrameOnScreen(this); //ATK utility to center window
      setVisible(true);
   }
   public static void main(String [] args)
   {
      new SecondApplication();
   }
   public void quitItemActionPerformed(ActionEvent evt)
```

```
        {
           System.exit(0);
        }
        public void errHistItemActionPerformed(ActionEvent evt)
        {
           errorHistory.setVisible(true);
        }
    }
```

This program (SeondApplication) should look something like this (depending on your platform and your device attributes)



7.3.3.2 Synoptic viewer

TangoATK provides a generic class to view and to animate the synoptics. The name of this class is fr.esrf.tangoatk.widget.jdraw.SynopticFileViewer. This class is based on a "home-made" graphical layer called jdraw. The jdraw package is also included inside TangoATK distribution.

SynopticFileViewer is a sub-class of the class TangoSynopticHandler. All the work for connection to tango devices and run time animation is done inside the TangoSynopticHandler.

The recipe for using the TangoATK synoptic viewer is the following

1. You use Jdraw graphical editor to draw your synoptic
2. During drawing phase don't forget to associate parts of the drawing to tango attributes or commands. Use the "name" in the property window to do this
3. During drawing phase you can also aasociate a class (frequently a "specific panel" class) which will be displayed when the user clicks on some part of the drawing. Use the "extension" tab in the property window to do this.
4. Test the run-time behaviour of your synoptic. Use "Tango Synoptic view" command in the "views" pulldown menu to do this.
5. Save the drawing file.
6. There is a simple synoptic application (SynopticAppli) which is provided ready to use. If this generic application is enough for you, you can forget about the step 7.
7. You can now develop a specific TangoATK based application which instantiates the SynopticFile-Viewer. To load the synoptic file in the SynopticFileViewer you have the choice : either you load it by giving the absolute path name of the synoptic file or you load the synoptic file using Java input streams. The second solution is used when the synoptic file is included inside the application jarfile.

The SynopticFilerViewer will browse the objects in the synoptic file at run time. It discovers if some parts of the drawing is associated with an attribute or a command. In this case it will automatically connect to the corresponding attribute or command. Once the connection is successfull SynopticFileViewer will animate the synoptic according to the default behaviour described below :

- For *tango state attributes* : the colour of the drawing object reflects the value of the state. A mouse click on the drawing object associated with the tango state attribute will instantiate and display the class specified during the drawing phase. If no class is specified the atkpanel generic device panel is displayed.
- For *tango attributes* : the current value of the attribute is displayed through the drawing object
- For *tango commands* : the mouse click on the drawing object associated with the command will launch the device command.
- If the tooltip property is set to "name" when the mouse enters *any tango object* (attribute or command), inside the synoptic drawing the name of the tango object is displayed in a tooltip.

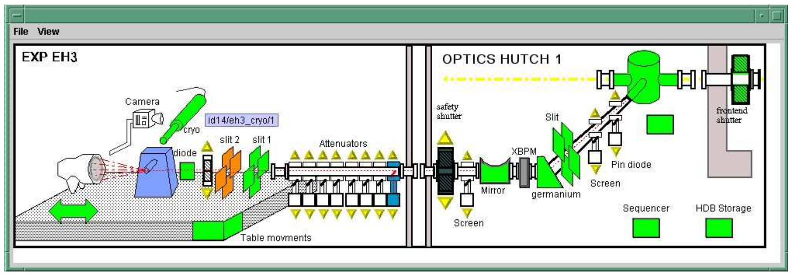
The following example (ThirdApplication), is a Synoptic application. We assume that the synoptic has already been drawn using Jdraw graphical editor.

```
package examples;
import java.io.*;
import java.util.*;
import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.BorderLayout;
import fr.esrf.tangoatk.widget.util.ErrorHistory;
import fr.esrf.tangoatk.widget.util.ATKGraphicsUtils;
import fr.esrf.tangoatk.widget.jdraw.SynopticFileViewer;
import fr.esrf.tangoatk.widget.jdraw.TangoSynopticHandler;
public class ThirdApplication extends JFrame
{
     JMenuBar              menu;
     ErrorHistory          errorHistory;  // A window that displays errors
     SynopticFileViewer    sfv;           // TangoATK generic synoptic viewer


     public ThirdApplication()
     {
        // Swing stuff to create the menu bar and its pulldown menus
        menu = new JMenuBar();
        JMenu fileMenu = new JMenu();
        fileMenu.setText("File");
        JMenu viewMenu = new JMenu();
        viewMenu.setText("View");
        JMenuItem quitItem = new JMenuItem();
        quitItem.setText("Quit");
        quitItem.addActionListener(new java.awt.event.ActionListener()
                                              {
```

```
                               public void actionPerformed(ActionEvent evt)
                               {quitItemActionPerformed(evt);}
                              });
fileMenu.add(quitItem);
JMenuItem errorHistItem = new JMenuItem();
errorHistItem.setText("Error History");
errorHistItem.addActionListener(new java.awt.event.ActionListener()
        {
         public void actionPerformed(ActionEvent evt)
         {errHistItemActionPerformed(evt);}
        });
viewMenu.add(errorHistItem);
menu.add(fileMenu);
menu.add(viewMenu);
//
// Here we create TangoATK synoptic viewer and error window.
//
errorHistory = new ErrorHistory();
sfv = new SynopticFileViewer();
try
{
    sfv.setErrorWindow(errorHistory);
}
catch (Exception setErrwExcept)
{
    System.out.println("Cannot set Error History Window");
}
//
// Here we define the name of the synoptic file to show and the tooltip mode to use
//
try
{
  sfv.setJdrawFileName("/users/poncet/ATK_OLD/jdraw_files/id14.jdw");
  sfv.setToolTipMode (TangoSynopticHandler.TOOL_TIP_NAME);
}
catch (FileNotFoundException  fnfEx)
{
   javax.swing.JOptionPane.showMessageDialog(
      null, "Cannot find the synoptic file : id14.jdw.\n"
           + "Check the file name you entered;"
           + " Application will abort ...\n"
           + fnfEx,
           "No such file",
           javax.swing.JOptionPane.ERROR_MESSAGE);
   System.exit(-1);
}
catch (IllegalArgumentException  illEx)
{
   javax.swing.JOptionPane.showMessageDialog(
      null, "Cannot parse the synoptic file : id14.jdw.\n"
           + "Check if the file is a Jdraw file."
           + " Application will abort ...\n"
           + illEx,
           "Cannot parse the file",
```

```
                    javax.swing.JOptionPane.ERROR_MESSAGE);
            System.exit(-1);
         }
         catch (MissingResourceException  mrEx)
         {
            javax.swing.JOptionPane.showMessageDialog(
               null, "Cannot parse the synoptic file : id14.jdw.\n"
                    + " Application will abort ...\n"
                    + mrEx,
                    "Cannot parse the file",
                    javax.swing.JOptionPane.ERROR_MESSAGE);
            System.exit(-1);
         }
         //
         // Add the viewers into the frame to show them
         //
         getContentPane().setLayout(new BorderLayout());
         getContentPane().add(sfv, BorderLayout.CENTER);
         //
         // add the menubar to the frame
         //
         setJMenuBar(menu);
         //
         // JFrame stuff to make the thing show.
         //
         pack();
         ATKGraphicsUtils.centerFrameOnScreen(this); //TangoATK utility to center window
         setVisible(true);
      }
      public static void main(String [] args)
      {
         new ThirdApplication();
      }
      public void quitItemActionPerformed(ActionEvent evt)
      {
         System.exit(0);
      }
      public void errHistItemActionPerformed(ActionEvent evt)
      {
         errorHistory.setVisible(true);
      }
   }
```

The synoptic application (ThirdApplication) should look something like this (depending on your synoptic drawing file)



7.3.4 A short note on the relationship between models and viewers

As seen in the examples above, the connection between a model and its viewer is generally done by calling `setModel(model)` on the viewer, it is never explained what happens behind the scenes when this is done.

7.3.4.1 Listeners

Most of the viewers implement some sort of *listener* interface, eg INumberScalarListener. An object implementing such a listener interface has the capability of receiving and treating *events* from a model which emits events.

```
// this is the setModel of a SimpleScalarViewer
  public void setModel(INumberScalar scalar) {
    clearModel();
    if (scalar != null) {
      format = scalar.getProperty("format").getPresentation();
      numberModel = scalar;

   // this is where the viewer connects itself to the
   // model. After this the viewer will (hopefully) receive
   // events through its numberScalarChange() method
   numberModel.addNumberScalarListener(this);


        numberModel.getProperty("format").addPresentationListener(this);
      numberModel.getProperty("unit").addPresentationListener(this);
    }
  }


// Each time the model of this viewer (the numberscalar attribute) decides it is time, it
// calls the numberScalarChange method of all its registered listeners
// with a NumberScalarEvent object which contains the
// the new value of the numberscalar attribute.
//

  public void numberScalarChange(NumberScalarEvent evt) {
    String val;
    val = getDisplayString(evt);
    if (unitVisible) {
      setText(val + " " + numberModel.getUnit());
    } else {
```

```
        setText(val);
      }
    }
```

All listeners in TangoATK implement the `IErrorListener` interface which specifies the `errorChange(ErrorEvent e)` method. This means that all listeners are forced to handle errors in some way or another.

7.4 The key objects of TangoATK

As seen from the examples above, the key objects of TangoATK are the `CommandList` and the `AttributeList`. These two classes inherit from the abstract class `AEntityList` which implements all of the common functionality between the two lists. These lists use the functionality of the `CommandFactory`, the `AttributeFactory`, which both derive from `AEntityFactory`, and the `DeviceFactory`.

In addition to these factories and lists there is one (for the time being) other important functionality lurking around, the refreshers.

7.4.1 The Refreshers

The refreshers, represented in TangoATK by the `Refresher` object, is simply a subclass of `java.lang.Thread` which will sleep for a given amount of time and then call a method refresh on whatever kind of `IRefreshee` it has been given as parameter, as shown below

```
// This is an example from DeviceFactory.
// We create a new Refresher with the name "device"
// We add ourself to it, and start the thread

Refresher refresher = new Refresher("device");
refresher.addRefreshee(this).start();
```

Both the `AttributeList` and the `DeviceFactory` implement the `IRefreshee` interface which specify only one method, `refresh()`, and can thus be refreshed by the `Refresher`. Even if the new release of TangoATK is based on the Tango Events, the refresher mecanisme will not be removed. As a matter of fact, the method refresh() implemented in ATTRIBUTELIST skips all attributes (members of the list) for which the subscribe to the tango event has succeeded and calls the old refresh() method for the others (for which subscribe to tango events has failed).

In a first stage this will allow the TangoATK applications to mix the use of the old tango device servers (which do not implement tango events) and the new ones in the same code. In other words, TangoATK subscribes for tango events if possible otherwise TangoATK will refresh the attributes through the old refresher mecanisme.

Another reason for keeping the refresher is that the subscribe event can fail even for the attributes of the new Tango device servers. As soon as the specified attribute is not polled the Tango events cannot be generated for that attribute. Therefore the event subscription will fail. In this case the attribute will be refreshed thanks to the ATK attribute list refresher.

The `AttributePolledList` class allows the application programmer to force explicitly the use of the refresher method for all attributes added in an AttributePolledList even if the corresponding device servers implement tango events. Some viewers (fr.esrf.tangoatk.widget.attribute.Trend) need an AttributePolledList in order to force the refresh of the attribute without using tango events.

7.4.1.1 What happens on a refresh

When `refresh` is called on the `AttributeList` and the `DeviceFactory`, they loop through their objects, `IAttributes` and `IDevices`, respectively, and ask them to refresh themselves if they are not event driven.

When ATTRIBUTEFACTORY, creates an `IAttribute`, TangoATK tries to subscribe for Tango Change event for that attribute. If the subscription succeeds then the attribute is marked as event driven. If the subscription for Tango Change event fails, TangoATK tries to subscribe for Tango Periodic event. If the subscription succeeds then the attribute is marked as event driven. If the subscription fails then the attribute is marked as to be " without events".

In the REFRESH() method of the ATTRIBUTELIST during the loop through the objects if the object is marked event driven then the object is simply skipped. But if the object (attribute) is not marked as event driven, the REFRESH() method of the ATTRIBUTELIST, asks the object to refresh itself by calling the "REFRESH()" method of that object (attribute or device). The REFRESH() method of an attribute will in turn call the "readAttribute" on the Tango device.

The result of this is that the `IAttributes` fire off events to their registered listeners containing snapshots of their state. The events are fired either because the IATTRIBUTE has received a Tango Change event, respectively a Tango Periodic event (event driven objects), or because the REFRESH() method of the object has issued a readAttribute on the Tango device.

7.4.2 The DeviceFactory

The device factory is responsible for two things

1. Creating new devices (Tango device proxies) when needed
2. Refreshing the state and status of these devices

Regarding the first point, new devices are created when they are asked for and only if they have not already been created. If a programmer asks for the same device twice, she is returned a reference to the same device-object.

The `DeviceFactory` contains a Refresher as described above, which makes sure that the all Devices in the DeviceFactory updates their state and status and fire events to its listeners.

7.4.3 The AttributeFactory and the CommandFactory

These factories are responsible for taking a name of an attribute or command and returning an object representing the attribute or command. It is also responsible for making sure that the appropriate `IDevice` is already available. Normally the programmer does not want to use these factory classes directly. They are used by TangoATK classes indirectly when the application programmer calls the AttributeList's (or CommandList's) ADD() method.

7.4.4 The AttributeList and the CommandList

These lists are containers for attributes and commands. They delegate the construction-work to the factories mentioned above, and generally do not do much more, apart from containing refreshers, and thus being able to make the objects they contain refresh their listeners.

7.4.5 The Attributes

The attributes come in several flavors. Tango supports the following types:

- Short
- Long
- Double

- String
- Unsigned Char
- Boolean
- Unsigned Short
- Float
- Unsigned Long

According to Tango specifications, all these types can be of the following formats:

- Scalar, a single value
- Spectrum, a single array
- Image, a two dimensional array

For the sake of simplicity, TangoATK has combined all the numeric types into one, presenting all of them as doubles. So the TangoATK classes which handle the numeric attributes are : NumberScalar, NumberSpectrum and NumberImage (Number can be short, long, double, float, ...).

7.4.5.1 The hierarchy

The numeric attribute hierarchy is expressed in the following interfaces:

INumberScalar extends **INumber**

INumberSpectrum extends **INumber**

INumberImage extends **INumber**

and **INumber** in turn extends **IAttribute**

Each of these types emit their proper events and have their proper listeners. Please consult the javadoc for further information.

7.4.6 The Commands

The commands in Tango are rather ugly beasts. There exists the following kinds of commands

- Those which take input
- Those which do not take input
- Those which do output
- Those which do not do output

Now, for both input and output we have the following types:

- Double
- Float
- Unsigned Long
- Long
- Unsigned Short

- Short
- String

These types can appear in scalar or array formats. In addition to this, there are also four other types of parameters:

1. Boolean
2. Unsigned Char Array
3. The StringLongArray
4. The StringDoubleArray

The last two types mentioned above are two-dimensional arrays containing a string array in the first dimension and a long or double array in the second dimension, respectively.

As for the attributes, all numeric types have been converted into doubles, but there has been made little or no effort to create an hierarchy of types for the commands.

7.4.6.1 Events and listeners

The commands publish results to their `IResultListeners`, by the means of a `ResultEvent`. The `IResultListener` extends `IErrorListener`, any viewer of command-results should also know how to handle errors. So a viewer of command-results implements IResultListener interface and registers itself as a resultListener for the command it has to show the results.

Chapter 8

Writing a TANGO device server

8.1 The device server framework

This chapter will present the TANGO device server framework. It will introduce what is the device server pattern and then it will describe a complete device server framework. A definition of classes used by the device server framework is given in this chapter. This manual is not intended to give the complete and detailed description of classes data member or methods, refer to [8] to get this full description. But first, the naming convention used in this project is detailed.

The aim of the class definition given in this chapter is only to help the reader to understand how a TANGO device server works. For a detailed description of these classes (and their methods), refer to chapter 8.4 or to [8].

8.1.1 Naming convention and programming language

TANGO fully supports two different programming languages which are **C++** and **Java**. When the Java code differs from the C++ code, examples in both languages will be given. For C++, its standard library has been used. Details about this library can be found in [9].

Every software project needs a naming convention. The naming convention adopted for the TDSOM is very simple and only defines two guidelines which are:

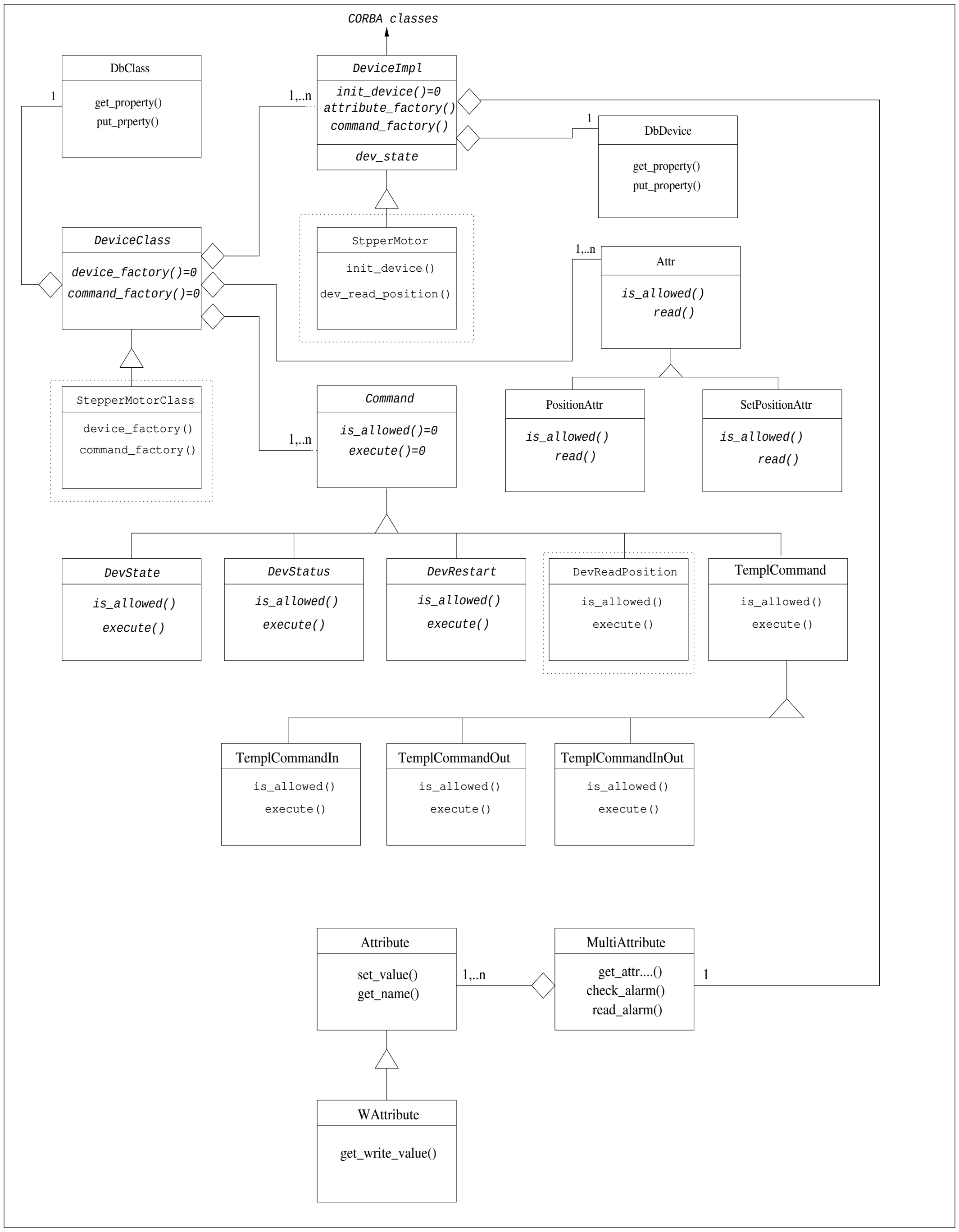
- Class names start with uppercase and use capitalization for compound words (For instance MyClassName).
- Method names are in lowercase and use underscores for compound words (For instance my_method_name).

These conventions will be use hereafter for both C++ and Java.

8.1.2 The device pattern

Device server are written using the Device pattern. The aim of this pattern is to provide the control programmer with a framework in which s/he can develop new control objects. The device pattern uses other design patterns like the Singleton and Command patterns. These patterns are fully described in [10]. The device pattern class diagram for stepper motor device is drawn in figure 8.1 . In this figure, only classes surrounded with a dash line square are device specific. All the other classes are part of the TDSOM core and are developed by the Tango system team. Different kind of classes are used by the device pattern.

- Three of them are root classes and it is only necessary to inherit from them. These classes are the **DeviceImpl**, **DeviceClass** and **Command** classes.
- Classes necessary to implement commands. The TDSOM supports two ways to create command : Using inheritance or using the template command model. It is possible to mix model within the same device pattern



(a) Device pattern class diagram

Figure 8.1: Device pattern class diagram

1. Using **inheritance**. This model of creating command heavily used the polymorphism offered by each modern object oriented programming language. In this schema, each command supported by a device via the command_inout or command_inout_async operation is implemented by a separate class. The Command class is the root class for each of these classes. It is an abstract class. A *execute* method must be defined in each sub-class. A *is_allowed* method may also be re-defined in each class if the default one does not fulfill all the needs[1]. In our stepper motor device server example, the DevReadPosition command follows this model.
2. Using the **template command** model. Using this model, it is not necessary to write one class for each command. You create one instance of classes already defined in the TDSOM for each command. The link between command name and method which need to be executed is done through pointers to method for C++ and through method names for Java. To support different kind of command, four classes are part of the TDSOM. These classes are :
   (a) The **TemplCommand** class for command without input or output parameter
   (b) The **TemplCommandIn** class for command with input parameter but without output parameter
   (c) The **TemplCommandOu**t class for command with output parameter but without input parameter
   (d) The **TemplCommandInOut** class for all the remaining commands

- Classes necessary to implement TANGO device attributes. All these classes are part of the TANGO core classes. These classes are the **MultiAttribute**, **Attribute**, **WAttribute**, **Attr**, **SpectrumAttr** and **ImageAttr** classes. The last three are used to create user attribute. Each attribute supported by a device is implemented by a separate class. The Attr class is the root class for each of these classes. According to the attribute data format, the user class implementing the attribute must inherit from the Attr, SpectrumAttr or ImageAtttr class. SpectrumAttr class inherits from Attr class and Image Attr class inherits from the SpectrumAttr class. The Attr base class defined three methods called *is_allowed*, *read* and *write*. These methods may be redefined in sub-classes in order to implement the attribute specific behaviour.
- The other are device specific. For stepper motor device, they are named StepperMotor, StepperMotorClass and DevReadPosition.

8.1.2.1 The DeviceImpl class

8.1.2.1.1 Description This class is the device root class and is the link between the Device pattern and CORBA. It inherits from CORBA classes and implements all the methods needed to execute CORBA operations and attributes. For instance, its method *command_inout* is executed when a client requests a command_inout operation. The method *name* of the DeviceImpl class is executed when a client requests the name CORBA attribute. This class also encapsulates some key device data like its name, its state, its status, its black box.... This class is an abstract class and cannot be instantiated as is.

8.1.2.1.2 Contents The contents of this class can be summarize as :

- Different constructors and one destructor
- Methods to access instance data members outside the class or its derivate classes. These methods are necessary because data members are declared as protected.
- Methods triggered by CORBA attribute request
- Methods triggered by CORBA operation request
- The *init_device()* method. This method makes the class abstract. It should be implemented by a sub-class. It is used by the inherited classes constructors.

[1] The default is_allowed method behavior is to always allows the command

- Methods triggered by the automatically added State and Status commands. These methods are declared virtual and therefore can be redefined in sub-classes. These two commands are automatically added to the list of commands defined for a class of devices. They are discussed in chapter 8.1.5
- A method called *always_executed_hook()* always executed for each command before the device state is tested for command execution. This method gives the programmer a hook where he(she) can program some mandatory action which must be done before any command execution. An example of the such action is an hardware access to the device to read its real hardware state.
- A method called *read_attr_hardware()* triggered by the read_attributes CORBA operation. This method is called once for each read_attributes call. This method is virtual and may be redefined in sub-classes.
- Methods for signal management (C++ specific)
- Data members like the device name, the device status, the device state
- Some private methods and data members

8.1.2.2 The DbDevice class

Each DeviceImpl instance is an aggregate with one instance of the DbDevice class. This DbDevice class can be used to query or modify device properties. It provides an easy to use interface for device objects in the database. The description of this class can be found in the Tango java or C++ API documentation.

8.1.2.3 The Command class

8.1.2.3.1 Description of the inheritance model Within the TDSOM, each command supported by a device and implemented using the inheritance model is implemented by a separate class. The Command class is the root class for each of these classes. It is an abstract class. It stores the command name, the command argument types and description and mainly defines two methods which are the *execute* and *is_allowed* methods. The *execute* method should be implemented in each sub-class. A default *is_allowed* method exists for command always allowed. A command also stores a parameter which is the command display type. It is also used to select if the command must be displayed according to the application mode (every day operation or expert mode).

8.1.2.3.2 Description of the template model Using this method, it is not necessary to create a separate class for each device command. In this method, each command is represented by an instance of one of the template command classes. They are four template command classes. All these classes inherits from the Command class. These four classes are :

1. The **TemplCommand** class. One object of this class must be created for each command without input nor output parameters
2. The **TemplCommandIn** class. One object of this class must be created for each command without output parameter but with input parameter
3. The **TemplCommandOut** class. One object of this class must be created for each command without input parameter but with output parameter
4. The **TemplCommandInOut** class. One object of this class must be created for each command with input and output parameters

These four classes redefine the *execute* and *is_allowed* method of the Command class. These classes provides constructors which allow the user to :

- specify which method must be executed by these classes *execute* method

- optionally specify which method must be executed by these classes *is_allowed* method.

The method specification is done via pointer to method with C++ and simply with method name for java.

Remember that it is possible to mix command implementation method within the same device pattern.

8.1.2.3.3 Contents The content of this class can be summarizes as :

- Class constructors and destructor
- Declaration of the *execute* method
- Declaration of the *is_allowed* method
- Methods to read/set class data members
- Methods to extract data from the object used to transfer data on the network
- Methods to insert data into the object used to transfer data on the network
- Class data members like command name, command input data type, command input data description...

8.1.2.4 The DeviceClass class

8.1.2.4.1 Description This class implements all what is specific for a controlled object class. For instance, every device of the same class supports the same list of commands and therefore, this list of available commands is stored in this DeviceClass. The structure returned by the info operation contains a documentation URL[2]. This documentation URL is the same for every device of the same class. Therefore, the documentation URL is a data member of this class. There should have only one instance of this class per device pattern implementation. The device list is also stored in this class. It is an abstract class because the two methods *device_factory()* and *command_factory()* are declared as pure virtual. The rule of the *device_factory()* method is to create all the devices belonging to the device class. The rule of the *command_factory()* method is to create one instance of all the classes needed to support device commands. This class also stored the *attribute_factory* method. The rule of this method is to store in a vector of strings, the name of all the device attributes. This method has a default implementation which is an empty body for device without attribute.

8.1.2.4.2 Contents The contents of this class can be summarize as :

- The *command_handler* method
- Methods to access data members.
- Signal related method (C++ specific)
- Class constructor. It is protected to implements the Singleton pattern
- Class data members like the class command list, the device list...

8.1.2.5 The DbClass class

Each DeviceClass instance is an aggregate with one instance of the DbClass class. This DbClass class can be used to query or modify class properties. It provides an easy to use interface for device objects in the database. The description of this class can be found in the Tango java or C++ API documentation.

[2] URL stands for **U**niform **R**esource **L**ocator

8.1.2.6 The MultiAttribute class

8.1.2.6.1 Description This class is a container for all the TANGO attributes defined for the device. There is one instance of this class for each device. This class is mainly an aggregate of Attribute object(s). It has been developed to ease TANGO attribute management.

8.1.2.6.2 Contents The class contents could be summarizes as :

- Miscellaneous methods to retrieve one attribute object in the aggregate
- Method to retrieve a list of attribute with an alarm level defined
- Get attribute number method
- Miscellaneous methods to check if an attribute value is outside the authorized limits
- Method to add messages for all attribute with an alarm set
- Data members with the attribute list

8.1.2.7 The Attribute class

8.1.2.7.1 Description There is one object of this class for each device attribute. This class is used to store all the attribute properties, the attribute value and all the alarm related data. Like commands, this class also stores th attribute display type. It is foreseen to be used by future Tango graphical application toolkit to select if the attribute must be displayed according to the application mode (every day operation or expert mode).

8.1.2.7.2 Contents

- Miscellaneous method to get boolean attribute information
- Methods to access some data members
- Methods to get/set attribute properties
- Method to check if the attribute is in alarm condition
- Methods related to attribute data
- Friend function to print attribute properties
- Data members (properties value and attribute data)

8.1.2.8 The WAttribute class

8.1.2.8.1 Description This class inherits from the Attribute class. There is one instance of this class for each writable device attribute. On top of all the data already managed by the Attribute class, this class stores the attribute set value.

8.1.2.8.2 Contents Within this class, you will mainly find methods related to attribute set value storage and some data members.

8.1.2.9 The Attr class

Within the TDSOM, each attribute supported by a device is implemented by a separate class. The Attr class is the root class for each of these classes. It is used in conjonction with the Attribute and Wattribute classes to implement Tango attribute behaviour. It defines three methods which are the *is_allowed, read* and *write* methods. A default *is_allowed* method exists for attribute always allowed. Default *read* and *write* empty methods are defined. For readable attribute, it is necessary to overwrite the *read* method. For writable attribute, it is necessary to overwrite the *write* method and for read and write attribute, both methods must be overwritten.

8.1.2.10 The SpectrumAttr class

This class inherits from the Attr class. It is the base class for user spectrum attribute. It is used in conjonction with the Attribute and WAttribute class to implement Tango spectrum attribute behaviour. From the Attr class, it inherits the Attr *is_allowed*, *read* and *write* methods.

8.1.2.11 The ImageAttr class

This class inherits from the SpectrumAttr class. It is the base class for user image attribute. It is used in conjonction with the Attribute and WAttribute class to implement Tango image attribute behaviour. From the Attr class, it inherits the Attr *is_allowed*, *read* and *write* methods.

8.1.2.12 The StepperMotor class

8.1.2.12.1 Description This class inherits from the DeviceImpl class and is the class implementing the controlled object behavior. Each command will trigger a method in this class written by the device server programmer and specific to the object to be controlled. This class also stores all the device specific data.

8.1.2.12.2 Definition

```
class StepperMotor: public Tango::DeviceImpl
{
public :
   StepperMotor(Tango::DeviceClass *,string &);
   StepperMotor(Tango::DeviceClass *,const char *);
   StepperMotor(Tango::DeviceClass *,const char *,const char *);
   ~StepperMotor() {};

   DevLong dev_read_position(DevLong);
   DevLong dev_read_direction(DevLong);
   bool direct_cmd_allowed(const CORBA::Any &);

   virtual Tango::DevState dev_state();
   virtual Tango::ConstDevString dev_status();

   virtual void always_executed_hook();

   virtual void read_attr_hardware(vector<long> &attr_list);

   void read_position(Tango::Attribute &);
   bool is_Position_allowed(Tango::AttReqType req);
   void write_SetPosition(Tango::WAttribute &);
   void read_Direction(Tango::Attribute &);
```

```
24
25   virtual void init_device();
26   virtual void delete_device();
27
28   void get_device_properties();
29
30 protected :
31   long axis[AGSM_MAX_MOTORS];
32   DevLong position[AGSM_MAX_MOTORS];
33   DevLong direction[AGSM_MAX_MOTORS];
34   long state[AGSM_MAX_MOTORS];
35
36   Tango::DevLong *attr_Position_read;
37   Tango::DevLong *attr_Direction_read;
38   Tango::DevLong attr_SetPosition_write;
39
40   Tango::DevLong min;
41   Tango::DevLong max;
42
43   Tango::DevLong *ptr;
44 };
45
46 } /* End of StepperMotor namespace */
```

Line 1 : The StepperMotor class inherits from the DeviceImpl class

Line 4-7 : Class constructors and destructor

Line 9 : Method triggered by the DevReadPosition command

Line 10-11 : Methods triggered by the DevReadDirection command

Line 13 : Redefinition of the *dev_state* method of the DeviceImpl class. This method will be triggered by the State command

Line 14 : Redefinition of the *dev_statu*s method of the DeviceImpl class. This method will be triggered by the Status command

Line 16 : Redefinition of the *always_executed_hook* method.

Line 25 : Definition of the *init_device* method (declared as pure virtual by the DeviceImpl class)

Line 26 : Definition of the *delete_device* method

Line 30-44 : Device data

8.1.2.13 The StepperMotorClass class

8.1.2.13.1 Description This class inherits from the DeviceClass class. Like the DeviceClass class, there should be only one instance of the StepperMotorClass. This is ensured because this class is written following the Singleton pattern as defined in [10]. All controlled object class data which should be defined only once per class must be stored in this object.

8.1.2.13.2 Definition

```
1 class StepperMotorClass : public DeviceClass
2 {
3 public:
4 static StepperMotorClass *init(const char *);
5 static StepperMotorClass *instance();
```

```
6 ~StepperMotorClass() {_instance = NULL;}
7
8 protected:
9 StepperMotorClass(string &);
10 static StepperMotorClass *_instance;
11 void command_factory();
12
13 private:
14 void device_factory(Tango_DevVarStringArray *);
15 };
```

Line 1 : This class is a sub-class of the DeviceClass class
Line 4-5 and 9-10: Methods and data member necessary for the Singleton pattern
Line 6 : Class destructor
Line 11 : Definition of the *command_factory* method declared as pure virtual in the DeviceClass call
Line 13-14 : Definition of the *device_factory* method declared as pure virtual in the DeviceClass class

8.1.2.14 The DevReadPosition class

8.1.2.14.1 Description This is the class for the DevReadPosition command. This class implements the *execute* and *is_allowed* methods defined by the Command class. This class is necessary because this command is implemented using the inheritance model.

8.1.2.14.2 Definition

```
1 class DevReadPositionCmd : public Command
2 {
3 public:
4     DevReadPositionCmd(const char *,Tango_CmdArgType, Tango_CmdArgType, const char *, const char*);
5     ~DevReadPositionCmd() {};
6
7     virtual bool is_allowed (DeviceImpl *, const CORBA::Any &);
8     virtual CORBA::Any *execute (DeviceImpl *, const CORBA::Any &);
9 };
```

Line 1 : The class is a sub class of the Command class
Line 4-5 : Class constructor and destructor
Line 7-8 : Definition of the *is_allowed* and *execute* method declared as pure virtual in the Command class.

8.1.2.15 The PositionAttr class

8.1.2.15.1 Description This is the class for the Position attribute. This attribute is a scalar attribute and therefore inherits from the Attr base class. This class implements the *read* and *is_allowed* methods defined by the Attr class.

8.1.2.15.2 Definition

```
1 class PositionAttr: public Tango::Attr
2 {
3 public:
4 PositionAttr():Attr("Position",Tango::DEV_LONG,Tango::READ);
5 ~PositionAttr() {};
6
7 virtual void read(Tango::DeviceImpl *dev,Tango::Attribute &att)
8 {(static_cast<StepperMotor *>(dev))->read_Position(att);}
9 virtual bool is_allowed(Tango::DeviceImpl *dev,Tango::AttReqType ty)
10 {return (static_cast<StepperMotor *>(dev))->is_Position_allowed(ty);}
11 };
```

Line 1 : The class is a sub class of the Attr class
Line 4-5 : Class constructor and destructor
Line 7 : Re-definition of the *read* method defined in the Attr class. This is simply a "forward" to the *read_Position* method of the StepperMotor class
Line 9 : Re-definition of the *is_allowed* method defined in the Attr class. This is also a "forward" to the *is_Position_allowed* method of the StepperMotor class

8.1.3 Startup of a device pattern

To start the device pattern implementation for stepper motor device, four methods of the StepperMotorClass class must be executed. These methods are :

1. The creation of the StepperMethodClass singleton via its *init*() method
2. The *command_factory*() method of the StepperMotorClass class
3. The *attribute_factory*() method of the StepperMotorClass class. This method has a default empty body for device class without attributes.
4. The *device_factory*() method of the StepperMotorClass class

This startup procedure is described in figure 8.2 . The creation of the StepperMotorClass will automatically create an instance of the DeviceClass class. The constructor of the DeviceClass class will create the Status, State and Init command objects and store them in its command list.

The *command_factory*() method will simply create all the user defined commands and add them in the command list.

The *attribute_factory*() method will simply build a list of device attribute names.

The *device_factory*() method will create each StepperMotor object and store them in the StepperMotorClass instance device list. The list of devices to be created and their names is passed to the *device_factory* method in its input argument. StepperMotor is a sub-class of DeviceImpl class. Therefore, when a StepperMotor object is created, a DeviceImpl object is also created. The DeviceImpl constructor builds all the device attribute object(s) from the attribute list built by the *attribute_factory()* method.

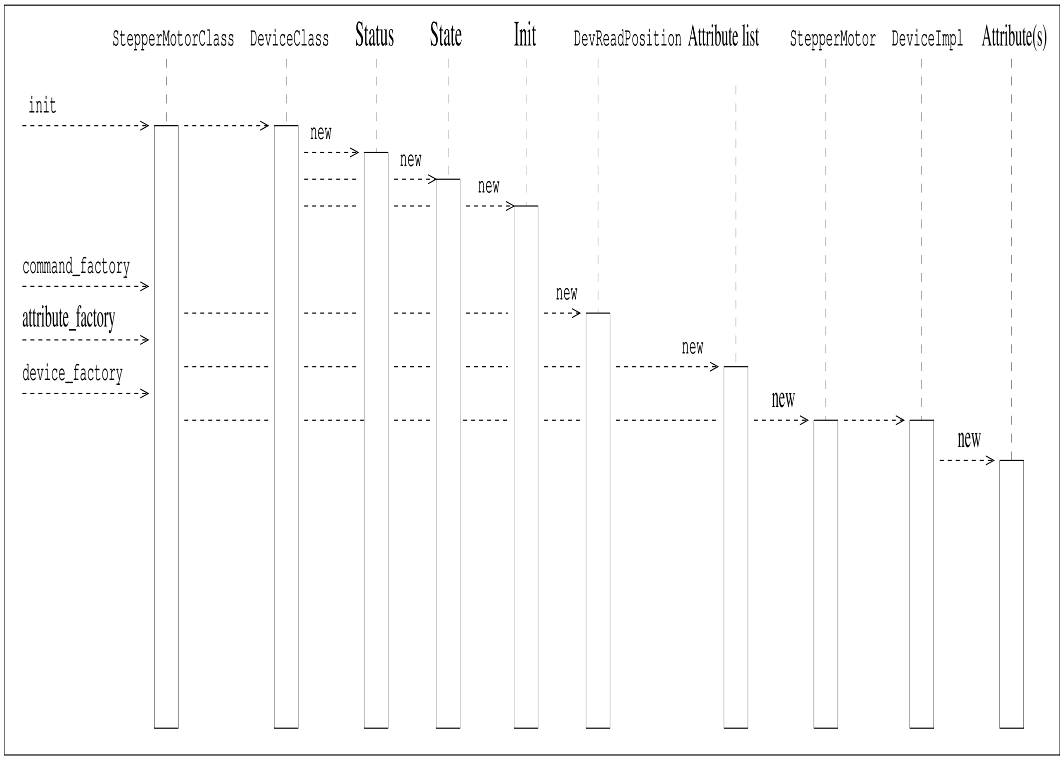


Figure 8.2: Device pattern startup sequence

8.1.4 Command execution sequence

The figure 8.3

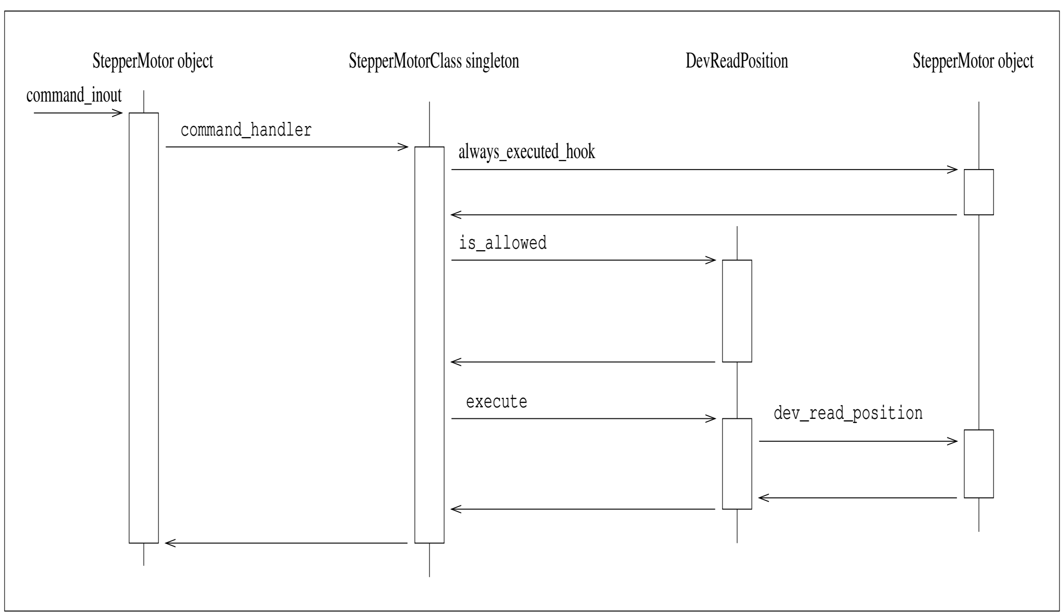


Figure 8.3: Command execution timing

described how the method implementing a command is executed when a command_inout CORBA operation is requested by a client. The *command_inout* method of the StepperMotor object (inherited from the DeviceImpl class) is triggered by an instance of a class generated by the CORBA IDL compiler. This method calls the *command_handler*() method of the StepperMotorClass object (inherited from the DeviceClass class). The *command_handler* method searches in its command list for the wanted command (using its name). If the command is found, the *always_executed_hook* method of the StepperMotor object is called. Then, the *is_allowed* method of the wanted command is executed. If the *is_allowed* method returns correctly, the *execute* method is executed. The *execute* method extracts the incoming data from the CORBA object use to transmit data over the network and calls the user written method which implements the command.

8.1.5 The automatically added commands

In order to increase the common behavior of every kind of devices in a TANGO control system, three commands are automatically added to each class of devices. These commands are :

- State
- Status
- Init

The default behavior of the method called by the State command depends on the device state. If the device state is ON or ALARM, the method will :

- read the attribute(s) with an alarm level defined
- check if the read value is above/below the alarm level and eventually change the device state to ALARM.
- returns the device state.

For all the other device state, the method simply returns the device state stored in the DeviceImpl class. Nevertheless, the method used to return this state (called *dev_state*) is defined as virtual and can be redefined in DeviceImpl sub-class. The difference between the default State command and the state CORBA attribute is the ability of the State command to signal an error to the caller by throwing an exception.

The default behavior of the method called by the Status command depends on the device state. If the device state is ON or ALARM, the method returns the device status stored in the DeviceImpl class plus additional message(s) for all the attributes which are in alarm condition. For all the other device state, the method simply returns the device status as it is stored in the DeviceImpl class. Nevertheless, the method used to return this status (called *dev_status*) is defined as virtual and can be redefined in DeviceImpl sub-class. The difference between the default Status command and the status CORBA attribute is the ability of the Status command to signal an error to the caller by throwing an exception.

The Init command is used to re-initialize a device without changing its network connection. This command calls the device *delete_device* method and the device *init_device* method. The rule of the *delete_device* method is to free memory allocated in the *init_device* method in order to avoid memory leak.

8.1.6 Reading/Writing attributes

8.1.6.1 Reading attributes

A Tango client is able to read Tango attribute(s) with the CORBA read_attributes call. Inside the device server, this call will trigger several methods of the device class (StepperMotor in our example) :

1. The *always_executed_hook()* method.
2. A method call *read_attr_hardware()*. This method is called one time per read_attributes CORBA call. The aim of this method is to read the device hardware and to store the result in a device class data member.

3. For each attribute to be read

   (a) A method called *is_<att name>_allowed()*. The rule of this method is to allow (or disallow) the next method to be executed. It is usefull for device with some attributes which can be read only in some precise conditions. It has one parameter which is the request type (read or write)

   (b) A method called *read_<att name>()*. The aim of this method is to extract the real attribute value from the hardware read-out and to store the attribute value into the attribute object. It has one parameter which is a reference to the Attribute object to be read.

The figure 8.4 is a drawing of these method calls sequencing. For attribute always readable, a default *is_allowed* method is provided. This method always returns true.

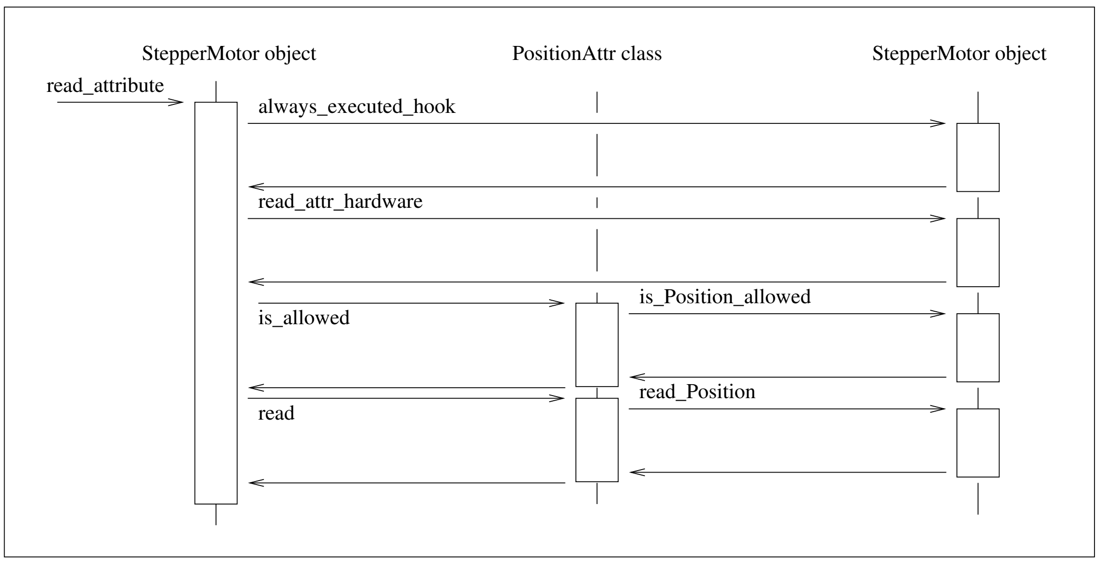


Figure 8.4: Read attribute sequencing

8.1.6.2 Writing attributes

A Tango client is able to write Tango attribute(s) with the CORBA write_attributes call. Inside a device server, this call will trigger several methods of the device class (StepperMotor in our example)

1. The *always_executed_hook()* method.

2. For each attribute to be written

   (a) A method called *is_<att name>_allowed()*. The rule of this method is to allow (or disallow) the next method to be executed. It is usefull for device with some attributes which can be written only in some precise conditions. It has one parameter which is the request type (read or write)

   (b) A method called *write_<att name>()*. It has one parameter which is a reference to the WAttribute object to be written. The aim of this method is to get the data to be written from the WAttribute object and to write this value into the corresponding hardware.

The figure 8.5 is a drawing of these method calls sequencing. For attribute always writeable, a default is_allowed method is provided. This method always allways returns true.

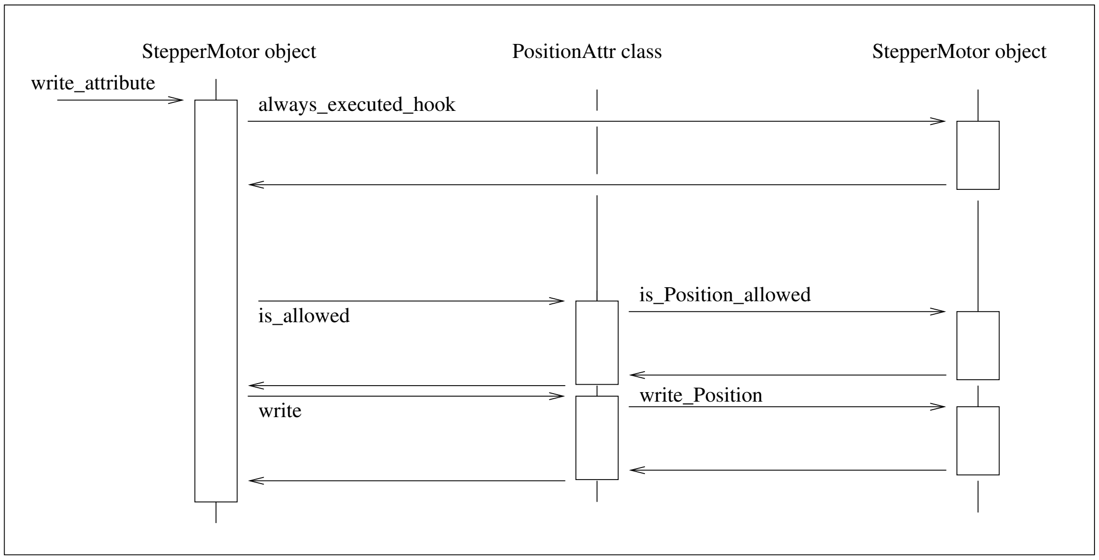


Figure 8.5: Write attribute sequencing

8.1.7 The device server framework

8.1.7.1 Vocabulary

A device server pattern implementation is embedded in a process called a **device server**. Several instances of the same device server process can be used in a TANGO control system. To identify instances, a device server process is started with an **instance name** which is different for each instance. The device server name is the couple device server executable name/device server instance name. For instance, a device server started with the following command

Perkin id11

starts a device server process with an instance name id11, an executable name Perkin and a device server name Perkin/id11.

8.1.7.2 The DServer class

In order to simplify device server process administration, a device of the DServer class is automatically added to each device server process. Thus, every device server process supports the same set of administration commands. The implementation of this DServer class follows the device pattern and therefore, its device behaves like any other devices. The device name is

dserver/device server executable name/device server instance name

For instance, for the device server process described in chapter 8.1.7.1, the dserver device name is dserver/perkin/id11. This name is returned by the adm_name CORBA attribute available for every device. On top of the three automatically added commands, this device supports the following commands :

- DevRestart
- RestartServer
- QueryClass
- QueryDevice
- Kill

- SetTraceLevel (Java server only)
- GetTraceLevel (Java server only)
- SetTraceOutput (Java server only)
- GetTraceOutput (Java server only)
- AddLoggingTarget (C++ server only)
- RemoveLoggingTarget (C++ server only)
- GetLoggingTarget (C++ server only)
- GetLoggingLevel (C++ server only)
- SetLoggingLevel (C++ server only)
- StopLogging (C++ server only)
- StartLogging (C++ server only)
- PolledDevice
- DevPollStatus
- AddObjPolling
- RemObjPolling
- UpdObjPollingPeriod
- StartPolling
- StopPolling
- EventSubscriptionChange
- ZmqEventSubscriptionChange
- LockDevice
- UnLockDevice
- ReLockDevices
- DevLockStatus

These commands will be fully described later in this document.

Several controlled object classes can be embedded within the same device server process and it is the rule of this device to create all these device server patterns and to call their command and device factories as described in 8.1.3. The name and number of all the classes to be created is known to this device after the execution of a method called *class_factory*. With C++, it is the user responsibility to write this method. Using Java, this method is already written and automatically retrieves which classes must be created and creates them.

8.1.7.3 The Tango::Util class

8.1.7.3.1 Description This class merges a complete set of utilities in the same class. It is implemented as a singleton and there is only one instance of this class per device server process. It is mandatory to create this instance in order to run a device server. The description of all the methods implemented in this class can be found in [8].

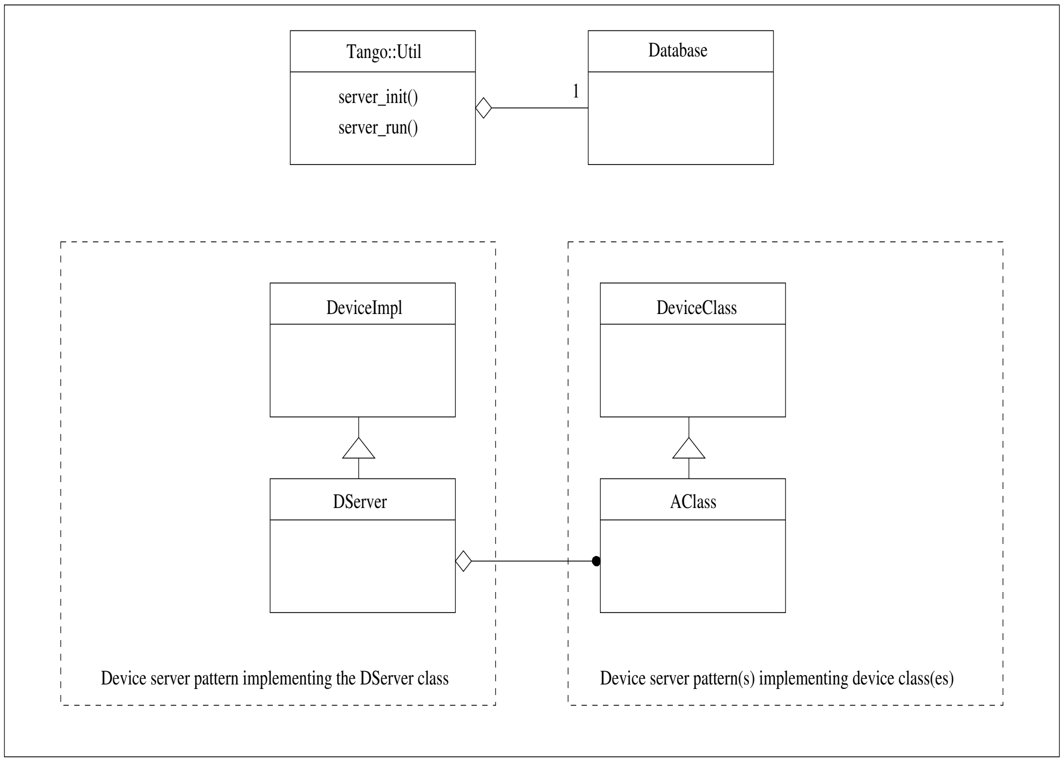


Figure 8.6: A complete device server

8.1.7.3.2 Contents Within this class, you can find :

- Static method to create/retrieve the singleton object
- Miscellaneous utility methods like getting the server output trace level, getting the CORBA ORB pointer, retrieving device server instance name, getting the server PID and more. Please, refer to [8] to get a complete list of all these utility methods.
- Method to create the device pattern implementing the DServer class (*server_init()*)
- Method to start the server (*server_run()*)
- TANGO database related methods

8.1.7.4 A complete device server

Within a complete device server, at least two implementations of the device server pattern are created (one for the dserver object and the other for the class of devices to control). On top of that, one instance of the Tango::Util class must also be created. A drawing of a complete device server is in figure 8.6

8.1.7.5 Device server startup sequence

The device server startup sequence is the following :

1. Create an instance of the Tango::Util class. This will initialize the CORBA Object Request Broker
2. Called the *server_init* method of the Tango::Util instance The call to this method will :
   (a) Create the DServerClass object of the device pattern implementing the DServer class. This will create the dserver object which during its construction will :

i. Called the *class_factory* method of the DServer object. This method must create all the xxxClass instance for all the device pattern implementation embedded in the device server process.

ii. Call the *command_factory* and *device_factory* of all the classes previously created. The list of devices passed to each call to the *device_factory* method is retrieved from the TANGO database.

3. Wait for incoming request with the *server_run()* method of the Tango::Util class.

8.2 Exchanging data between client and server

Exchanging data between clients and server means most of the time passing data between processes running on different computer using the network. Tango limits the type of data exchanged between client and server and defines a way to exchange these data. This chapter details these features. Memory allocation and error reporting are also discussed.

All the rules described in this chapter are valid only for data exchanged between client and server. For device server internal data, classical C++ or Java types can be use.

8.2.1 Command / Attribute data types

Commands have a fixed calling syntax - consisting of one input argument and one output argument. Arguments type must be chosen out of a fixed set of 24 data types. Attributes support a sub-set of these data types. These are the data type with the (1) note. The following table details type name, code and the corresponding CORBA IDL types.

The type name used in the type name column of this table is the C++ name. In the IDL file, all the Tango definition are grouped in a IDL module named Tango. The IDL module maps to C++ namespace. Therefore, all the data type are parts of a namespace called Tango. For Java, the IDL module maps to Java package and name are not changed related to the IDL file.

Type name	IDL type
Tango::DevBoolean (1)	boolean
Tango::DevShort (1)	short
Tango::DevLong (1)	long
Tango::DevLong64 (1)	long long
Tango::DevFloat (1)	float
Tango::DevDouble (1)	double
Tango::DevUShort (1)	unsigned short
Tango::DevULong (1)	unsigned long
Tango::DevULong64 (1)	unsigned long long
Tango::DevString (1)	string
Tango::DevVarCharArray	sequence of unsigned char
Tango::DevVarShortArray	sequence of short
Tango::DevVarLongArray	sequence of long
Tango::DevVarLong64Array	sequence of long long
Tango::DevVarFloatArray	sequence of float
Tango::DevVarDoubleArray	sequence of double
Tango::DevVarUShortArray	sequence of unsigned short
Tango::DevVarULongArray	sequence of unsigned long
Tango::DevVarULong64Array	sequence of unsigned long long
Tango::DevVarStringArray	sequence of string
Tango::DevVarLongStringArray	structure with a sequence of long and a sequence of string

Tango::DevVarDoubleStringArray	structure with a sequence of double and a sequence of string
Tango::DevState (1)	enumeration
Tango::DevEncoded (1)	structure with a string and a sequence of char

The CORBA Interface Definition Language uses a type called **sequence** for variable length array. This sequence type is mapped differently according to the language used (C++ or Java). The Tango::DevUxxx types are used for unsigned types. The Tango::DevVarxxxxArray must be used when the data to be transferred are variable length array. The Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray are structures with two fields which are variable length array of Tango long (32 bits) and variable length array of strings for the Tango::DevVarLongStringArray and variable length array of double and variable length array of string for the Tango::DevVarDoubleStringArray. The Tango::State type is used by the State command to return the device state.

8.2.1.1 Using command data types with C++

Unfortunately, the mapping between IDL and C++ was defined before the C++ class library had been standardized. This explains why the standard C++ string class or vector classes are not used in the IDL to C++ mapping.

TANGO commands argument types can be grouped on five groups depending on the IDL data type used. These groups are :

1. Data type using basic types (Tango::DevBoolean, Tango::DevShort, Tango::DevLong, Tango::DevFloat, Tango::DevDouble, Tango::DevUshort and Tango::DevULong)
2. Data type using strings (Tango::DevString type)
3. Data types using sequences (Tango::DevVarxxxArray types except Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray)
4. Data types using structures (Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray types)
5. Data type using enumeration (Tango::DevState type)

In the following sub chapters, only summaries of the IDL to C++ mapping are given. For a full description of the C++ mapping, please refer to [2]

8.2.1.1.1 Basic types For these types, the mapping between IDL and C++ is obvious and defined in the following table.

Tango type name	IDL type	C++	typedef
Tango::DevBoolean	boolean	CORBA::Boolean	unsigned char
Tango::DevShort	short	CORBA::Short	short
Tango::DevLong	long	CORBA::Long	int
Tango::DevLong64	long long	CORBA::LongLong	long long or long (64 bits chip)
Tango::DevFloat	float	CORBA::Float	float
Tango::DevDouble	double	CORBA::Double	double
Tango::DevUShort	unsigned short	CORBA::UShort	unsigned short
Tango::DevULong	unsigned long	CORBA::ULong	unsigned long
Tango::DevULong64	unsigned long long	CORBA:ULongLong	unsigned long long or unsigned long (64 bits chip)

The types defined in the column named C++ should be used for a better portability. All these types are defined in the CORBA namespace and therefore their qualified names is CORBA::xxx.

8.2.1.1.2 Strings Strings are mapped to **char ***. The use of *new* and *delete* for dynamic allocation of strings is not portable. Instead, you must use helper functions defined by CORBA (in the CORBA namespace). These functions are :

```
char *CORBA::string_alloc(unsigned long len);
char *CORBA::string_dup(const char *);
void CORBA::string_free(char *);
```

These functions handle dynamic memory for strings. The *string_alloc* function allocates one more byte than requested by the len parameter (for the trailing 0). The function *string_dup* combines the allocation and copy. Both *string_alloc* and *string_dup* return a null pointer if allocation fails. The *string_free* function must be used to free memory allocated with *string_alloc* and *string_dup*. Calling *string_free* for a null pointer is safe and does nothing. The following code fragment is an example of the Tango::DevString type usage

```
1 Tango::DevString str = CORBA::string_alloc(5);
2 strcpy(str,"TANGO");
3
4 Tango::DevString str1 = CORBA::string_dup("Do you want to danse TANGO?");
5
6 CORBA::string_free(str);
7 CORBA::string_free(str1);
```

Line 1-2 : TANGO is a five letters string. The CORBA::string_alloc function parameter is 5 but the function allocates 6 bytes

Line 4 : Example of the CORBA::string_dup function

Line 6-7 : Memory deallocation

8.2.1.1.3 Sequences IDL sequences are mapped to C++ classes that behave like vectors with a variable number of elements. Each IDL sequence type results in a separate C++ class. Within each class representing a IDL sequence types, you find the following method (only the main methods are related here) :

1. Four constructors.
   (a) A default constructor which creates an empty sequence.
   (b) The maximum constructor which creates a sequence with memory allocated for at least the number of elements passed as argument. This does not limit the number of element in the sequence but only the way how memory is allocated to store element
   (c) A sophisticated constructor where it is possible to assign the memory used by the sequence with a preallocated buffer.

(d) A copy constructor which does a deep copy

2. An assignment operator which does a deep copy
3. A *length* accessor which simply returns the current number of elements in the sequence
4. A *length* modifier which changes the length of the sequence (which is different than the number of elements in the sequence)
5. Overloading of the [] operator. The subscript operator [] provides access to the sequence element. For a sequence containing elements of type T, the [] operator is overloaded twice to return value of type T & and const T &. Insertion into a sequence using the [] operator for the const T & make a deep copy. Sequence are numbered between 0 and *length*() -1.

Note that using the maximum constructor will not prevent you from setting the length of the sequence with a call to the length modifier. The following code fragment is an example of how to use a Tango::DevVarLongArray type

```
1 Tango::DevVarLongArray *mylongseq_ptr;
2 mylongseq_ptr = new Tango::DevVarLongArray();
3 mylongseq_ptr->length(4);
4
5 (*mylongseq_ptr)[0] = 1;
6 (*mylongseq_ptr)[1] = 2;
7 (*mylongseq_ptr)[2] = 3;
8 (*mylongseq_ptr)[3] = 4;
9
10 // (*mylongseq_ptr)[4] = 5;
11
12 CORBA::Long nb_elt = mylongseq_ptr->length();
13
14 mylongseq_ptr->length(5);
15 (*mylongseq_ptr)[4] = 5;
16
17 for (int i = 0;i < mylongseq_ptr->length();i++)
18 cout << "Sequence elt " << i + 1 << " = " << (*mylongseq_ptr)[i] << endl;
```

Line 1 : Declare a pointer to Tango::DevVarLongArray type which is a sequence of long

Line 2 : Create an empty sequence

Line 3 : Change the length of the sequence to 4

Line 5 - 8 : Initialize sequence elements

Line 10 ; Oups !!! The length of the sequence is 4. The behavior of this line is undefined and may be a core can be dumped at run time

Line 12 : Get the number of element actually stored in the sequence

Line 14-15 : Grow the sequence to five elements and initialize element number 5

Line 17-18 : Print sequence element

Another example for the Tango::DevVarStringArray type is given

```
1 Tango::DevVarStringArray mystrseq(4);
2 mystrseq.length(4);
3
4 mystrseq[0] = CORBA::string_dup("Rock and Roll");
5 mystrseq[1] = CORBA::string_dup("Bossa Nova");
6 mystrseq[2] = CORBA::string_dup(“Waltz”);
7 mystrseq[3] = CORBA::string_dup("Tango");
8
9 CORBA::Long nb_elt = mystrseq.length();
10
11 for (int i = 0;i < mystrseq.length();i++)
12 cout << "Sequence elt " << i + 1 << " = " << mystrseq[i] << endl;
```

Line 1 : Create a sequence using the maximum constructor
Line 2 : Set the sequence length to 4. This is mandatory even if you used the maximum constructor.
Line 4-7 : Populate the sequence
Line 9 : Get how many strings are stored into the sequence
Line 11-12 : Print sequence elements.

8.2.1.1.4 Structures Only three TANGO types are defined as structures. These types are the Tango::DevVarLongStringArray, the Tango::DevVarDoubleStringArray and the Tango::DevEncoded data type. IDL structures map to C++ structures with corresponding members. For the Tango::DevVarLongStringArray, the two members are named *svalue* for the sequence of strings and *lvalue* for the sequence of longs. For the Tango::DevVarDoubleStringArray, the two structure members are called *svalue* for the sequence of strings and *dvalue* for the sequence of double. For the Tango::DevEncoded, the two structure members are called *encoded_format* for a string describing the data coding and *encoded_data* for the data themselves. The encoded_data field type is a Tango::DevVarCharArray. An example of the usage of the Tango::DevVarLongStringArray type is detailed below.

```
1 Tango::DevVarLongStringArray my_vl;
2
3 myvl.svalue.length(2);
4 myvl.svalue[0] = CORBA_string_dup("Samba");
5 myvl.svalue[1] = CORBA_string_dup("Rumba");
6
7 myvl.lvalue.length(1);
8 myvl.lvalue[0] = 10;
```

Line 1 : Declaration of the structure
Line 3-5 : Initialization of two strings in the sequence of string member
Line 7-8 : Initialization of one long in the sequence of long member

8.2.1.1.5 Enumeration Only one TANGO type is an enumeration. This is the Tango::DevState type used to transfer device state between client and server. IDL enumerated types map to C++ enumerations (amazing no!) with a trailing dummy enumerator to force enumeration to be a 32 bit type. The first enumerator will have the value 0, the next one will have the value 1 and so on.

```
Tango::DevState state;

state = Tango::ON;
state = Tango::FAULT;
```

8.2.1.2 Using command data types with Java

All the rules described in this chapter are valid only for data exchanged between client and server. For device server internal data, classical Java types can be use.

TANGO commands argument types can be grouped on four groups depending on the IDL data type used. These groups are :

1. Data type using basic types (DevBoolean, DevShort, DevLong, DevFloat, DevDouble, DevUShort, DevULong and DevString)
2. Data types using sequences (DevVarxxxArray types except DevVarLongStringArray and DevVarDoubleStringArray)
3. Data types using structures (DevVarLongStringArray and DevVarDoubleStringArray types)
4. Data type using enumeration (DevState type)

In the following sub chapters, only summaries of the IDL to Java mapping are given. For a full description of the Java mapping, please refer to [12].

8.2.1.2.1 Basic types For these types, the mapping between IDL and Java is obvious and defined in the following table.

Tango type name	IDL type	Java type
DevBoolean	boolean	boolean
DevShort	short	short
DevLong	long	int
DevLong64	long long	long
DevFloat	float	float
DevDouble	double	double
DevString	string	String
DevUShort	unsigned short	short
DevULong	unsigned long	int
DevULong64	unsigned long long	long

The Java int is a 32 bits type[3] and therefore, the DevLong type maps to Java int. Java does not support unsigned types, this is why the DevUShort type maps to short and the DevULong type maps to int. In the contrary of C++, Java does not support a preprocessor and therefore, declaring a data from the DevLong type (or any other type in the previous table) will result in compiler errors. Instead, the Java types must be used.

IDL string maps directly to java.lang.String class.

[3] The Java long type is a 64 bits data type

8.2.1.2.2 Sequences IDL sequences map to Java array. The following tables details the mapping used for Tango sequence types.

Tango type name	IDL type	Java type
DevVarCharArray	sequence of byte	byte[]
DevVarShortArray	sequence of short	short[]
DevVarLongArray	sequence of long	int[]
DevVarLong64Array	sequence of long long	long[]
DevVarFloatArray	sequence of float	float[]
DevVarDoubleArray	sequence of double	double[]
DevVarUShortArray	sequence of unsigned short	short[]
DevVarULongArray	sequence of unsigned long	int[]
DevVarULong64Array	sequence of unsigned long long	long[]
DevVarStringArray	sequence of string	String[]

8.2.1.2.3 Structures IDL structures map to a final Java class with the same name. This class provides instance variables for all IDL structure fields. It also provides a default constructor and a constructor from all structures fields values. The class name, the field name and types are summaries in the following table

Tango type name	Java class name	field name	field Java type
DevVarLongStringArray	DevVarLongStringArray	lvalue svalue	int[] String[]
DevVarDoubleStringArray	DevVarDoubleStringArray	dvalue svalue	double[] String[]
DevEncoded	DevEncoded	encoded_format encoded_data	String char[]

8.2.1.2.4 Enumeration Enumeration does not exist in Java. An IDL enumeration is mapped to a final class with the same name as the enum type. This class has the following members :

1. A *value* method which returns the value as an integer.
2. A pair of static data members per label.
   (a) The first one is an integer with a name equals to the label name prepended with an underscore ("_") like _ON for instance.
   (b) The second one is a reference to an object of the class representing the enumeration with its value set to the label value.
3. An integer conversion method called *from_int* which returns a reference to an object of the class representing the enumeration
4. A private constructor

The following code fragment is an example of Tango command data types usage

---

```
1 short l = 2;
2
3 String[] str_array = new String[2];
4 str_array[0] = new String("Be Bop");
5 str_array[1] = new String("Break dance");
6
7 System.out.println("Elt nb in DevVarStringArray data " + str_array.length);
8 for (int i = 0;i < str_array.length;i++)
9 System.out.println("Element value = " + str_array[i]);
10
11 DevVarLongStringArray ls = new DevVarLongStringArray();
12 ls.lvalue = new int[1];
13 ls.lvalue[0] = 1;
14 ls.svalue = new String[2];
15 ls.svalue[0] = new String("Smurf");
16 ls.svalue[1] = new String("Pogo");
17
18 DevState st = DevState.FAULT;
19 switch (st.value())
20 {
21 case DevState._ON :
22 System.out.println("The state is ON");
23 st = DevState.FAULT;
24 break;
25
26 case DevState._FAULT :
27 System.out.println("The state is FAULT");
28 st = DevState.ON;
29 break;
30 }
```

Line 1 : Use of a DevShort type (pretty simple no)
Line 3-5 : Use of a DevVarStringArray data type with 2 elements
Line 7-9 : Print DevVarStringArray data element number and value
Line 11-16 : Use of a DevVarLongStringArray data type
Line 18 : Initialization of a DevState data with the FAULT state
Line 19 : Test on the DevState data value
Line 21 : Use the integer value associated to each enumeration label to test DevState data
Line 23 : Update DevState data value

8.2.2 Passing data between client and server

In order to have one definition of the CORBA operation used to send a command to a device whatever the command data type is, TANGO uses CORBA IDL **any** object. The IDL type *any* provides a universal type that can hold a value of arbitrary IDL types. Type *any* therefore allows you to send and receive values whose types are not fixed at compile time.

Type *any* is often compared to a void * in C. Like a pointer to void, an *any* value can denote a datum of any type. However, there is an important difference; whereas a void * denotes a completely untyped value that can be interpreted only with advance knowledge of its type, values of type *any* maintain type safety.

For example, if a sender places a string value into an *any*, the receiver cannot extract the string as a value of the wrong type. Attempt to read the contents of an *any* as the wrong type cause a run-time error.

Internally, a value of type *any* consists of a pair of values. One member of the pair is the actual value contained inside the *any* and the other member of the pair is the type code. The type code is a description of the value's type. The type description is used to enforce type safety when the receiver extracts the value. Extraction of the value succeeds only if the receiver extracts the value as a type that matches the information in the type code.

Within TANGO, the command input and output parameters are objects of the IDL *any* type. Only insertion/extraction of all types defined as command data types is possible into/from these *any* objects.

8.2.2.1 C++ mapping for IDL any type

The IDL any maps to the C++ class **CORBA::Any**. This class contains a large number of methods with mainly methods to insert/extract data into/from the any. It provides a default constructor which builds an any which contains no value and a type code that indicates "no value". Such an any must be used for command which does not need input or output parameter. The operator **<<=** is overloaded many times to insert data into an any object. The operator **>>=** is overloaded many times to extract data from an any object.

8.2.2.1.1 Inserting/Extracting TANGO basic types The insertion or extraction of TANGO basic types is straight forward using the <<= or >>= operators. Nevertheless, the Tango::DevBoolean type is mapped to a unsigned char and other IDL types are also mapped to char C++ type (The unsigned is not taken into account in the C++ overloading algorithm). Therefore, it is not possible to use operator overloading for these IDL types which map to C++ char. For the Tango::DevBoolean type, you must use the *CORBA::Any::from_boolean* or *CORBA::Any::to_boolean* intermediate objects defined in the CORBA::Any class.

8.2.2.1.2 Inserting/Extracting TANGO strings The <<= operator is overloaded for const char * and always makes a deep copy. This deep copy is done using the CORBA::*string_dup* function. The extraction of strings uses the >>= overloaded operator. The main point is that the Any object retains ownership of the string, so the returned pointer points at memory inside the Any. This means that you must not deallocate the extracted string and you must treat the extracted string as read-only.

8.2.2.1.3 Inserting/Extracting TANGO sequences Insertion and extraction of sequences also uses the overloaded <<= and >>= operators. The insertion operator is overloaded twice: once for insertion by reference and once for insertion by pointer. If you insert a value by reference, the insertion makes a deep copy. If you insert a value by pointer, the Any assumes the ownership of the pointed-to memory.

Extraction is always by pointer. As with strings, you must treat the extracted pointer as read-only and must not deallocate it because the pointer points at memory internal to the Any.

8.2.2.1.4 Inserting/Extracting TANGO structures This is identical to inserting/extracting sequences.

8.2.2.1.5 Inserting/Extracting TANGO enumeration This is identical to inserting/extracting basic types

---

```
CORBA::Any a;
Tango::DevLong l1,l2;
l1 = 2;
a <<= l1;
a >>= l2;

```

```
7 CORBA::Any b;
8 Tango::DevBoolean b1,b2;
9 b1 = true;
10 b <<= CORBA::Any::from_boolean(b1);
11 b >>= CORBA::Any::to_boolean(b2);
12
13 CORBA::Any s;
14 Tango::DevString str1,str2;
15 str1 = "I like dancing TANGO";
16 s <<= str1;
17 s >>= str2;
18
19 // CORBA::string_free(str2);
20 // a <<= CORBA::string_dup("Oups");
21
22 CORBA::Any seq;
23 Tango::DevVarFloatArray fl_arr1;
24 fl_arr1.length(2);
25 fl_arr1[0] = 1.0;
26 fl_arr1[1] = 2.0;
27 seq <<= fl_arr1;
28 const Tango::DevVarFloatArray *fl_arr_ptr;
29 seq >>= fl_arr_ptr;
30
31 // delete fl_arr_ptr;
```

Line 1-5 : Insertion and extraction of Tango::DevLong type

Line 7-11 Insertion and extraction of Tango::DevBoolean type using the CORBA::Any::from_boolean and CORBA::Any::to_boolean intermediate structure

Line 13-17 : Insertion and extraction of Tango::DevString type

Line 19 : Wrong ! You should not deallocate a string extracted from an any

Line 20 : Wrong ! Memory leak because the <<= operator will do the copy.

Line 22-29 : Insertion and extraction of Tango::DevVarxxxArray types. This is an insertion by reference and the use of the <<= operator makes a deep copy of the sequence. Therefore, after line 27, it is possible to deallocate the sequence

Line 31: Wrong.! You should not deallocate a sequence extracted from an any

8.2.2.2 The insert and extract methods of the Command class

In order to simplify the insertion/extraction into/from Any objects, small helper methods have been written in the Command class. The signatures of these methods are :

```
void extract(const CORBA::Any &,<Tango type> &);
CORBA::Any *insert(<Tango type>);
```

An *extract* method has been written for all Tango types. These method extract the data from the Any object passed as parameter and throw an exception if the Any data type is incompatible with the awaiting type. An *insert* method have been written for all Tango types. These method create an Any object, insert the data into the Any and return a pointer to the created Any. For Tango types mapped to sequences or structures, two *insert* methods have been written: one for the insertion from pointer and the other for the insertion from reference. For Tango strings, two *insert* methods have been written: one for insertion from a classical Tango::DevString type and the other from a const Tango::DevString type. The first one deallocate the memory after the insert into the Any object. The second one only inserts the string into the Any object.

The previous example can be rewritten using the insert/extract helper methods (We suppose that we can use the Command class insert/extract methods)

```
1 Tango::DevLong l1,l2;
2 l1 = 2;
3 CORBA::Any *a_ptr = insert(l1);
4 extract(*a_ptr,l2);
5
6 Tango::DevBoolean b1,b2;
7 b1 = true;
8 CORBA::Any *b_ptr = insert(b1);
9 extract(*b_ptr,b2);
10
11 Tango::DevString str1,str2;
12 str1 = "I like dancing TANGO";
13 CORBA::Any *s_ptr = insert(str1);
14 extract(*s_ptr,str2);
15
16 Tango::DevVarFloatArray fl_arr1;
17 fl_arr1.length(2);
18 fl_arr1[0] = 1.0;
19 fl_arr1[1] = 2.0;
20 insert(fl_arr1);
21 CORBA::Any *seq_ptr = insert(fl_arr1);
22 Tango::DevVarFloatArray *fl_arr_ptr;
23 extract(*seq_ptr,fl_arr_ptr);
```

Line 1-4 : Insertion and extraction of Tango::DevLong type

Line 6-9 : Insertion and extraction of Tango::DevBoolean type

Line 11-14 : Insertion and extraction of Tango::DevString type

Line 16-23 : Insertion and extraction of Tango::DevVarxxxArray types. This is an insertion by reference which makes a deep copy of the sequence. Therefore, after line 20, it is possible to deallocate the sequence

8.2.2.3 Java mapping for IDL any type

The IDL any maps to the Java class **org.omg.CORBA.Any** . This class has all the necessary methods to insert and extract instances of IDL native types (short, int, float, string..). The method name to insert native IDL types is *insert_<type name>* (*insert_short*(), *insert_float*(), *insert_string*()). They all take a reference to the element to be inserted as argument. The method name to extract basic types is *extract_<type name>* (*extract_short*(), *extract_float*() or *extract_string*()). These extract methods do not need argument and return a reference to the extracted data. If the extraction operations have a mismatched type, the CORBA BAD_OPERATION exception is raised. An "any" object is constructed with the *create_any*() method of the CORBA "orb" object. This orb object represents the Object Request Broker. Within a Tango device server, you can retrieve it with a method of the TangoUtil class described in [8].

8.2.2.3.1 Inserting/Extracting TANGO basic types and strings The insertion or extraction of TANGO basic types and strings is straight forward using the insert or extract methods provided by the org.omg.CORBA.Any class.

8.2.2.3.2 Inserting/Extracting TANGO sequences, structures or enumeration The IDL to Java compiler generates Helper classes for all types defined in the IDL file. The generated classes name is the name of the type followed by the suffix **Helper** (DevVarCharArrayHelper, DevLongHelper). Classes are generated even for types which directly map to native Java types. Several static methods needed to manipulate the type are supplied in these classes. These include "Any" insert and extract operations for the type. For a data type <typename>, the insert and extract method are :

- public static void insert(org.omg.CORBA.Any a, <typename> t) {..}
- public static <typename> extract(Any a) {...}

Such classes exists for all the TANGO data types. The following code fragment is an example of the insertion/extraction in/from Any object with Java

```
1 Any a = TangoUtil.instance().get_orb().create_any();
2 int l1 = 1;
3 a.insert_long(l1);
4 int l2 = a.extract_long();
5
6 DevLongHelper.insert(a,l1);
7 int l3 = DevLongHelper.extract(a);
8
9 Any s = TangoUtil.instance().get_orb().create_any();
10 String str = new String("I like dancing TANGO");
11 s.insert_string(str);
12 String str_ex = s.extract_string();
13
14 DevStringHelper.insert(s,str);
15 String str_help = DevStringHelper.extract(s);
16
17 Any arr = TangoUtil.instance().get_orb().create_any();
18 int[] array = new int[2];
19 array[0] = 1;
20 array[1] = 2;
21 DevVarLongArrayHelper.insert(arr,array);
22 int[] array_ext = DevVarLongArrayhelper.extract(arr);
```

Line 1 : Create an instance of the Any class.

Line 3 : Insert a DevLong data into the Any object. The method name is insert_long because this is a method to insert an IDL long type into the object even if the IDL long type maps to an int in Java.

Line 4 : Extract a DevLong type from the Any

Line 6-7 : Insert or Extract DevLong data type to/from the Any object using the Helper class.

Line 9-12 : Create an Any object and a DevString data. Insert and Extract this string into/from the Any using the method provided by the any object

Line 14-15 : Insert or Extract string into/from the Any using methods provided by the Helper class

Line 17-22 : The same thing for data of the DevVarLongArray type. Note that DevVarLongArray is not a basic IDL type and the Any class does not provide method to insert/extract data of this type into/from the Any. The use of the methods provided by the Helper class is mandatory in this case.

8.2.2.4 The insert and extract methods of the Command class for Java

In order to simplify the insertion/extraction into/from Any objects, small helper methods have been written in the Command class. The signatures of these methods are :

```
<java type> extract_<Tango type_name>(Any);
Any insert(<Tango type>);
```

An *extract* method has been written for all Tango types. These method extract the data from the Any object passed as parameter and throw an exception if the Any data type is incompatible with the awaiting type. All these *extract* methods take the same input parameter and only differ in their return type which is not taken into account for method overloading. Therefore, the name of the method depends on the type of the data to be extracted. The following is some example of these method names and signatures :

- *int extract_DevLong(Any) throws DevFailed* for the DevLong type
- *int[] extract_DevVarULongArray(Any) throws DevFailed* for DevVarULongArray type
- *String[] extract_DevVarStringArray(Any) throws DevFailed* for DevVarStringArray

An *insert* method have been written for all Tango types. These method create an Any object, insert the data into the Any and return a pointer to the created Any. The previous example can be rewritten using the insert/extract helper methods (We suppose that we can use the Command class insert/extract methods)

```
1 int l1 = 1;
2 Any a = insert(l1);
3 int l2 = extract_DevLong(a);
4
5 String str = new String("I like dancing TANGO");
6 Any s = insert(str);
7 String str_ex = extract_DevString(s);
8
9 int[] array = new int[2];
10 array[0] = 1;
11 array[1] = 2;
12 Any arr = insert(array);
13 int[] array_ext = extract_DevVarLongArray(arr);
```

Line 1-3 : Insertion/Extraction of DevLong type
Line 5-7 : Insertion/Extraction of DevString type
Line 9-13 : Insertion/Extraction of DevVarLongArray type

8.2.3 C++ memory management

The rule described here are valid for variable length command data types like Tango::DevString or all the Tango:: DevVarxxxxArray types.

The method executing the command must allocate the memory used to pass data back to the client or use static memory (like buffer declares as object data member. If necessary, the ORB will deallocate this memory after the data have been sent to the caller. Fortunately, for incoming data, the method have no memory management responsibilities. The details about memory management given in this chapter assume that the insert/extract methods of the Tango::Command class are used and only the method in the device object is discussed.

8.2.3.1 For string

Example of a method receiving a Tango::DevString and returning a Tango::DevString is detailed just below

```
1 Tango::DevString MyDev::dev_string(Tango::DevString argin)
2 {
3 Tango::DevString argout;
4
5 cout << "the received string is " << argin << endl;
6
7 string str("Am I a good Tango dancer ?");
8 argout = new char[str.size() + 1];
9 strcpy(argout,str.c_str());
10
11 return argout;
12 }
```

Note that there is no need to deallocate the memory used by the incoming string. Memory for the outgoing string is allocated at line 8, then it is initialized at the following line. The memory allocated at line 8 will be automatically freed by the usage of the *Command::insert()* method. Using this schema, memory is allocated/freed each time the command is executed. For constant string length, a statically allocated buffer can be used.

```
1 Tango::ConstDevString MyDev::dev_string(Tango::DevString argin)
2 {
3 Tango::ConstDevString argout;
4
5 cout << "the received string is " << argin << endl;
6
7 argout = "Hello world";
8 return argout;
9 }
```

A Tango::ConstDevString data type is used. It is not a new data Tango data type. It has been introduced only to allows *Command::insert()* method overloading. The argout pointer is initialized at line 7 with memory statically allocated. In this case, no memory will be freed by the *Command::insert()* method. There is also no memory copy in the contrary of the previous example. A buffer defined as object data member can also be used to set the argout pointer.

8.2.3.2 For array/sequence

Example of a method returning a Tango::DevVarLongArray is detailed just below

```
1 Tango::DevVarLongArray *MyDev::dev_array()
2 {
3 Tango::DevVarLongArray *argout = new Tango::DevVarLongArray();
4
5 long output_array_length = ...;
6 argout->length(output_array_length);
7 for (int i = 0;i < output_array_length;i++)
8 (*argout)[i] = i;
9
10 return argout;
11 }
```

In this case, memory is allocated at line 3 and 6. Then, the sequence is populated. The sequence is created and returned using pointer. The *Command::insert()* method will insert the sequence into the CORBA::Any object using this pointer. Therefore, the CORBA::Any object will take ownership of the allocated memory. It will free it when it will be destroyed by the CORBA ORB after the data have been sent away. It is also possible to use a statically allocated memory and to avoid copying in the sequence used to returned the data. This is explained in the following example assuming a buffer of long data is declared as device data member and named buffer.

```
1 Tango::DevVarLongArray *MyDev::dev_array()
2 {
3 Tango::DevVarLongArray *argout;
4
5 long output_array_length = ...;
6 argout = create_DevVarLongArray(buffer,output_array_length);
7 return argout;
8 }
```

At line 3 only a pointer to a DevVarLongArray is defined. This pointer is set at line 6 using the *create_DevVarLongArray()* method. This method will create a sequence using this buffer without memory allocation and with minimum copying. The *Command::insert()* method used here is the same than the one used in the previous example. The sequence is created in a way that the destruction of the CORBA::Any object in which the sequence will be inserted will not destroy the buffer. The following create_xxx methods are defined in the DeviceImpl class :

Method name	data type
create_DevVarCharArray()	unsigned char
create_DevVarShortArray()	short
create_DevVarLongArray()	DevLong
create_DevVarLong64Array()	DevLong64
create_DevVarFloatArray()	float
create_DevVarDoubleArray()	double
create_DevVarUShortArray()	unsigned short
create_DevVarULongArray()	DevULong
create_DevVarULong64Array()	DevULong64

8.2.3.3 For string array/sequence

Example of a method returning a Tango::DevVarStringArray is detailed just below

```
1 Tango::DevVarStringArray *MyDev::dev_str_array()
2 {
3 Tango::DevVarStringArray *argout = new Tango::DevVarStringArray();
4
5 argout->length(3);
6 (*argout)[0] = CORBA::string_dup("Rumba");
7 (*argout)[1] = CORBA::string_dup("Waltz");
8 string str("Jerck");
9 (*argout)[2] = CORBA::string_dup(str.c_str());
10 return argout;
11 }
```

Memory is allocated at line 3 and 5. Then, the sequence is populated at lines 6,7 and 9. The usage of the *CORBA::string_dup* function also allocates memory. The sequence is created and returned using pointer. The *Command::insert()* method will insert the sequence into the CORBA::Any object using this pointer. Therefore, the CORBA::Any object will take ownership of the allocated memory. It will free it when it will be destroyed by the CORBA ORB after the data have been sent away. For portability reason, the ORB uses the *CORBA::string_free* function to free the memory allocated for each string. This is why the corresponding *CORBA::string_du*p or *CORBA::string_alloc* function must be used to reserve this memory.It is also possible to use a statically allocated memory and to avoid copying in the sequence used to returned the data. This is explained in the following example assuming a buffer of pointer to char is declared as device data member and named int_buffer.

```
1 Tango::DevVarStringArray *DocDs::dev_str_array()
2 {
3 int_buffer[0] = "first";
4 int_buffer[1] = "second";
5
6 Tango::DevVarStringArray *argout;
7 argout = create_DevVarStringArray(int_buffer,2);
8 return argout;
9 }
```

The intermediate buffer is initialized with statically allocated memory at lines 3 and 4. The returned sequence is created at line 7 with the *create_DevVarStringArray()* method. Like for classical array, the sequence is created in a way that the destruction of the CORBA::Any object in which the sequence will be inserted will not destroy the buffer.

8.2.3.4 For Tango composed types

Tango supports only two composed types which are Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray. These types are translated to C++ structure with two sequences. It is not possible to use memory statically allocated for these types. Each structure element must be initialized as described in the previous subchapters using the dynamically allocated memory case.

8.2.4 Reporting errors

Tango uses the C++ and Java try/catch plus exception mechanism to report errors. Two kind of errors can be transmitted between client and server :

1. CORBA system error. These exceptions are raised by the ORB and indicates major failures (A communication failure, An invalid object reference...)
2. CORBA user exception. These kind of exceptions are defined in the IDL file. This allows an exception to contain an arbitrary amount of error information of arbitrary type.

TANGO defines one user exception called **DevFailed**. This exception is a variable length array of **DevError** type (a sequence of DevError). The DevError type is a four fields structure. These fields are :

1. A string describing the type of the error. This string replaces an error code and allows a more easy management of include files.
2. The error severity. It is an enumeration with the three values which are WARN, ERR or PANIC.
3. A string describing in plain text the reason of the error
4. A string describing the origin of the error

The Tango::DevFailed type is a sequence of DevError structures in order to transmit to the client what is the primary error reason when several classes are used within a command. The sequence element 0 must be the DevError structure describing the primary error. A method called *print_exception*() defined in the Tango::Except class prints the content of exception (CORBA system exception or Tango::DevFailed exception). Some static methods of the Tango::Except class called *throw_exception*() can be used to throw Tango::DevFailed exception. Some other static methods called *re_throw_exception()* may also be used when the user want to add a new element in the exception sequence and re-throw the exception. With Java, these functions are static methods of the Except class. Details on these methods can be found in [8].

8.2.4.1 Example of throwing exception using C++

This example is a piece of code from the *command_handler*() method of the DeviceImpl class. An exception is thrown to the client to indicate that the requested command is not defined in the command list.

```
TangoSys_OMemStream o;

o << "Command " << command << " not found" << ends;
Tango::Except::throw_exception((const char *)"API_CommandNotFound",
o.str(),
(const char *)"DeviceClass::command_handler");


try
{
.....
}
```

```
13 catch (Tango::DevFailed &e)
14 {
15 TangoSys_OMemStream o;
16
17 o << "Command " << command << " not found" << ends;
18 Tango::Except::re_throw_exception(e,
19 (const char *)"API_CommandNotFound",
20 o.str(),
21 (const char *)"DeviceClass::command_handler");
22 }
```

Line 1 : Build a memory stream. Use the TangoSys_MemStream because memory streams are not managed the same way between Windows and Unix

Line 3 : Build the reason string in the memory stream

Line 4-5 : Throw the exception to client using one of the *throw_exception* static method of the Except class. This throw_exception method used here allows the definition of the error type string, the reason string and the origin string of the DevError structure. The remaining DevError field (the error severity) will be set to its default value. Note that the first and third parameters are casted to a *const char* *. Standard C++ defines that such a string is already a *const char* * but the GNU C++ compiler (release 2.95) does not use this type inside its function overloading but rather uses a *char* * which leads to calling the wrong function.

Line 13-22 : Re-throw an already catched tango::DevFailed exception with one more element in the exception sequence.

8.2.4.2 Example of throwing exception using Java

This example is a fragment of code from the *command_handler*() method of the DeviceImpl class. An exception is thrown to the client to indicate that the requested command is not defined in the command list.

```
1 StringBuffer o = new StringBuffer("Command ");
2 o.append(command);
3 o.append(" not found");
4
5 Except.throw_exception("API_CommandNotFound",
6 o.toString(),
7 "DeviceClass.command_handler");
```

Line 1-3 : Build a string with a message describing the error. The StringBuffer class is used instead of the String class because the StringBuffer class allows dynamic resizing of the string.

Line 5-7 : Throw the exception to client using the static *throw_exception* method of the Except class. The *throw_exception* method used here allows the definition of the reason string, the description string and the origin string of the DevError structure. The remaining DevError field (the error severity) will be set to its default value. Like C++, some static *re_throw_exception()* methods also exist to re-throw DevFailed exception with one more sequence element.

Note that the CORBA system exception inherits from the java.lang.RuntimeException. Exception derivate from this class do not need to be catched or re-thrown. This is the case for the BAD_OPERATION exception thrown when a mismatched type is used to extract data from an Any object. CORBA user exception (like the DevFailed exception) inherits from the java.Exception class and needs to be catched or re-thrown.

8.3 The Tango Logging Service

A first introduction about this logging service has been done in chapter 3.5

The TANGO Logging Service (TLS) gives the user the control over how much information is actually generated and to where it goes. In practice, the TLS allows to select both the logging level and targets of any device within the control system.

8.3.1 Logging Targets

The TLS implementation allows each device logging requests to print simultaneously to multiple destinations. In the TANGO terminology, an output destination is called a **logging target**. Currently, targets exist for console, file and log consumer device.

CONSOLE: logs are printed to the console (i.e. the standard output),

FILE: logs are stored in a XML file. A rolling mechanism is used to backup the log file when it reaches a certain size (see below),

DEVICE: logs are sent to a device implementing a well known TANGO interface (see section A.8 for a definition of the log consumer interface). One implementation of a log consumer associated to a graphical user interface is available within the Tango package. It is called the LogViewer.

The device's logging behavior can be control by adding and/or removing targets.

Note : When the size of a log file (for file logging target) reaches the so-called rolling-file-threshold (rft), it is backuped as "current_log_file_name" + "_1" and a new "current_log_file_name" is opened. Obviously, there is only one backup file at a time (i.e. any existing backup is destroyed before the current log file is backuped). The default threshold is 2Mb, the minimum is 500 Kb and the maximum is 20 Mb.

8.3.2 Logging Levels

Devices can be assigned a logging level. It acts as a filter to control the kind of information sent to the targets. Since, there are (usually) much more low level log statements than high level statements, the logging level also control the amount of information produced by the device. The TLS provides the following levels (semantic is just given to be indicative of what could be log at each level):

OFF: Nothing is logged

FATAL: A fatal error occurred. The process is about to abort

ERROR: An (unrecoverable) error occurred but the process is still alive

WARN: An error occurred but could be recovered locally

INFO: Provides information on important actions performed

DEBUG: Generates detailed information describing the internal behavior of a device

Levels are ordered the following way:

DEBUG < INFO < WARN < ERROR < FATAL < OFF

For a given device, a level is said to be enabled if it is greater or equal to the logging level assigned to this device. In other words, any logging request which level is lower than the device's logging level is ignored.

Note: The logging level can't be controlled at target level. The device's targets shared the same device logging level.

8.3.3 Sending TANGO Logging Messages

8.3.3.1 Logging macros in C++

The TLS provides the user with easy to use C++ macros with *printf* and *stream* like syntax. For each logging level, a macro is defined in both styles:

- LOG_{FATAL, ERROR, WARN, INFO or DEBUG}
- {FATAL, ERROR, WARN, INFO or DEBUG}_STREAM

These macros are supposed to be used within the device's main implementation class (i.e. the class that inherits (directly or indirectly) from the Tango::DeviceImpl class). In this context, they produce logging messages containing the device name. In other words, they automatically identify the log source. Section 8.3.3.2 gives a trick to log in the name of device outside its main implementation class. Printf like example:

```
LOG_DEBUG(("Msg#%d - Hello world", i++));
```

Stream like example:

```
DEBUG_STREAM << "Msg#" << i++ << "- Hello world" << endl;
```

These two logging requests are equivalent. Note the double parenthesis in the printf version.

8.3.3.2 C++ logging in the name of a device

A device implementation is sometimes spread over several classes. Since all these classes implement the same device, their logging requests should be associated with this device name. Unfortunately, the C++ logging macros can't be used because they are outside the device's main implementation class. The Tango::LogAdapter class is a workaround for this limitation.

Any method not member of the device's main implementation class, which send log messages associated to a device must be a member of a class inheriting from the Tango::LogAdapter class. Here is an example:

```
class MyDeviceActualImpl: public Tango::LogAdapter
{
public :
MyDeviceActualImpl(...,Tango::DeviceImpl *device,...)
:Tango::LogAdpater(device)
{
....
//
// The following log is associated to the device passed to the constructor
//
DEBUG_STREAM << "In MyDeviceActualImpl constructor" << endl;

....
}
};
```

8.3.3.3 Logging in Java

In order to send a log from a device implementation method (i.e. a method of a class inheriting from *TangoDs.DeviceImpl*), the developer makes use of the *org.apache.log4j.Logger* instance which reference is returned by the *DeviceImpl.get_logger* method. The *org.apache.log4j.Logger.{fatal,error,warn,info* and *debug}* methods provide the actual logging features. See for more information about the Logger class. Here is an example of Logging usage with Java:

```
public class myDevice extends DeviceImpl implements TangoConst
{
...

public void init_device()
{

// A Debug log

get_logger().debug("Initializing device " + get_name());

try
{
// Initialization code
String p = get_property("startup property");
if (p == null)
{
get_logger().warn("No startup property defined for " + get_name());
...
}
}
catch (Exception e)
{
// An error log

get_logger().error("unknown exception caught");
}
}
...
}
```

8.3.3.4 Logging in the name of a device with Java

Using Java, you can log in the name of a device from anywhere in your code as far as you get a reference to this device. Use the device *get_logger* public method to obtain its associated logger then proceed as describe in 8.3.3.3.

8.4 Writing a device server

Writing a device server can be made easier by adopting the correct approach. This chapter will describe how to write a device server. It is divided into the following parts : understanding the device, defining device commands, choosing device state and writing the necessary classes. All along this chapter, examples will be given using the stepper motor device server. Writing a device server for our stepper motor example device means writing :

- The *main* function
- The *class_factory* method (only for C++ device server)
- The *StepperMotorClass* class
- The *DevReadPositionCmd* and *DevReadDirectionCmd* classes
- The *PositionAttr*, *SetPositionAttr* and *DirectionAttr* classes
- The *StepperMotor* class.

All these functions and classes will be detailed. The stepper motor device server described in this chapter supports 2 commands and 3 attributes which are :

- Command DevReadPosition implemented using the inheritance model
- Command DevReadDirection implemented using the template command model
- Attribute Position (position of the first motor). This attribute is readable and is linked with a writable attribute (called SetPosition). When the value of this attribute is requested by the client, the value of the associated writable attribute is also returned.
- Attribute SetPosition (writable attribute linked with the Position attribute). This attribute has some properties with user defined default value.
- Attribute Direction (direction of the first motor)

As the reader will understand during the reading of the following sub-chapters, the command and attributes classes (*DevReadPositionCmd*, *DevReadDirectionCmd*, *PositionAttr*, *SetPositionAttr* and *DirectionAttr*) are very simple classes. A tool called **Pogo** has been developped to automatically generate/maintain these classes and to write part of the code needed in the remaining one. See xx to know more on this Pogo tool.

In order to also gives an example of how the database objects part of the Tango device pattern could be used, our device have two properties. These properties are of the Tango long data types and are named “Max” and “Min”.

8.4.1 Understanding the device

The first step before writing a device server is to develop an understanding of the hardware to be programmed. The Equipment Responsible should have description of the hardware and its operating modes (manuals, spec sheets etc.). The Equipment Responsible must also provide specifications of what the device server should do. The Device Server Programmer should demand an exact description of the registers, alarms, interlocks and any timing constraints which have to be kept. It is very important to have a good understanding of the device interfacing before starting designing a new class.

Once the Device Server Programmer has understood the hardware the next important step is to define what is a logical device i.e. what part of the hardware will be abstracted out and treated as a logical device. In doing so the following points of the TDSOM should be kept in mind

- Each device is known and accessed by its ascii name.
- The device is exported onto the network to be imported by applications.
- Each device belongs to a class.
- A list of commands exists per device.
- Applications use the device server api to execute commands on a device.

The above points have to be taken into account when designing the level of device abstraction. The definition of what is a device for a certain hardware is primarily the job of the Device Server Programmer and the Applications Programmer but can also involve the Equipment Responsible. The Device Server Programmer should make sure that the Applications Programmer agrees with her definition of what is a device.

Here are some guidelines to follow while defining the level of device abstraction -

- **efficiency**, make sure that not a too fine level of device abstraction has been chosen. If possible group as many attributes together to form a device. Discuss this with the Applications Programmer to find out what is efficient for her application.

- **hardware independency**, one of the main reasons for writing device servers is to provide the Applications Programmer with a *software* interface as opposed to a *hardware* interface. Hide the hardware structure of the device. For example if the user is only interested in a single channel of a multichannel device then define each channel to be a logical device. The user should not be aware of hardware addresses or cabling details. The user is very often a scientist who has a physics-oriented world view and not a hardware-oriented world view. Hardware independency also has the advantage that applications are immune to hardware changes to the device

- **object oriented world view**, another *raison d'etre* behind the device server model is to build up an object oriented view of the world. The device should resemble the user's view of the object as closely as possible. In the case of the ESRF's beam lines for example, the devices should resemble beam line scientist's view of the machine.

- **atomism**, each device can be considered like an atom - is a independent object. It should appear independent to the client even if behind the scenes it shares some hardware or software with other objects. This is often the case with multichannel devices where the user would like to see each channel as a device but it is obvious that the channels cannot be programmed completely independently. The logical device is there to hide or make transparent this fact. If it is impossible to send commands to one device without modifying another device then a single device should be made out the two devices.

- **tailored** *vs* **general**, one of the philosophies of the TDSOM is to provide tailored solutions. For example instead of writing one *serial line* class which treats the general case of a serial line device and leaving the device protocol to be implemented in the client the TDSOM advocates implementing a device class which handles the protocol of the device. This way the client only has to know the commands of the class and not the details of the protocol. Nothing prevents the device class from using a general purpose serial line class if it exists of course.

8.4.2 Defining device commands

Each device has a list of commands which can be executed by the application across the network or locally. These commands are the Application Programmer's network knobs and dials for interacting with the device.

The list of commands to be implemented depends on the capabilities of the hardware, the list of sensible functions which can be executed at a distance and of course the functionality required by the application. This implies a close collaboration between the Equipment Responsible, Device Server Programmer and the Application Programmer.

When drawing up the list of commands particular attention should be paid to the following points

- **performance**, no single command should monopolize the device server for a long time (a nominal value for long is one second). Commands should be implemented in such a way that it executes immediately returning with a response. At best try to keep command execution time down to less than the typical overhead of an rpc call i.e. som milliseconds. This of course is not always possible e.g. a serial line device could require 100 milliseconds of protocol exchange. The Device Server

Programmer should find the best trade-off between the users requirements and the devices capabilities. If a command implies a sequence of events which could last for a long time then implement the sequence of events in another thread - don't block the device server.

- **robustness**, should be provided which allow the client to recover from error conditions and or do a warm startup.

8.4.2.1 Standard commands

A minimum set of three commands exist for all devices. These commands are

- State which returns the state of a device
- Status which returns the status of the device as a formatted ascii string
- Init which re-initialize a device without changing its network connection

These commands have already been discussed in 8.1.5

8.4.3 Choosing device state

The device state is a number which reflects the availability of the device. To simplify the coding for generic application, a predefined set of states are supported by TANGO. This list has 14 members which are

State name
ON
OFF
CLOSE
OPEN
INSERT
EXTRACT
MOVING
STANDBY
FAULT
INIT
RUNNING
ALARM
DISABLE
UNKNOWN

The names used here have obvious meaning.

8.4.4 Device server utilities to ease coding/debugging

The device server framework supports one in C++ and two set of utilities to ease the process of coding and debugging device server code. These utilities are :

1. The device server verbose option
2. The device server output redirection system (Java specific)

Using these two facilities avoids the usage of the classical "#ifdef DEBUG" style which makes code less readable.

8.4.4.1 The device server verbose option

Each device server supports a verbose option called **-v**. Four verbose levels are defined from 1 to 4. Level 4 is the most talkative one. If you use the -v option without specifying level, level 4 will be assumed.

Since Tango release 3, a Tango Logging Service has been introduced (detailed in chapter 8.3). This -v option set-up the logging service. If it used, it will automatically add a *console* target to all devices embedded within the device server process. Level 1 and 2 will set the logging level to all devices embedded within the device server to INFO. Level 3 and 4 will set the logging level to all devices embedded within the device server to DEBUG. All messages sent by the API layer are associated to the administration device.

Java specific: A device server started with output level n will print all the messages of level between 1 and n. For instance, if you start a device server using -v3 option, only the output for level 1,2 and 3 will be displayed. Output for level 4 will not be printed. If you don't used the -v option, the output level is set to 0. By convention, level 3 and 4 are reserved for print message embedded into the Tango library. Level 1 and 2 are free for the user.

8.4.4.1.1 Choosing the output level using C++ In C++ device server, this feature is now implemented using the Tango Logging Service (TLS), see chapter 8.3 to get all details on this service.

8.4.4.1.2 Choosing the output level using Java With Java, four static objects inside the Util class have been defined. These objects are called *out1*, *out2*, *out3* and *out4*. These four objects support the *println* method exactly as the *out* object inside the System class does. The first object (*out1*) defines a message which should be printed only when output level 1 or more is requested. The second one (*out2*) defines a message which should be printed only when output level 2 or more is requested. The same philosophy is used for *out3* and *out4*. The usage of these *outx* objects is the same than the classical *out*.

```
1 Util.out3.println("What a nice dance");
2 Util.out3.println("What's its name ?");
3
4 System.out.println("Its name is TANGO");
```

Line 1-2 : The two questions are level 3 messages.

Line 4 : This print will be printed whatever the print level is.

If this piece of code is part of a device server started with a -v2 option, only the message defined line 4 will be displayed. If the device server is started with a -v3, -v4 or -v option, the two messages defined at lines 1 and 2 will also be displayed.

8.4.4.1.3 Changing the output level at run time (Java specific) It is possible to change the output level at run time. You do so using commands of the dserver device. These two commands are :

- SetTraceLevel. This command needs the new trace level as input parameter. Using this command supersedes the level requested at device server process command line
- GetTraceLevel. This command returns the actual trace level.

8.4.4.2 Device server output redirection (Java specific)

Two commands of the dserver device allow device server output redirection. Theses two commands are :

- SetTraceOutput. This command sets all the device server output used to print message to be redirected to a file. This command needs the complete file path as input parameter. The file is local to the computer where the device server process is running.

- GetTraceOutput. This command returns the name of the file used to redirect device server process output. If no SetTraceOutput command has been used prior to the execution of this command, it returns a special string (“Initial Output”) to indicates that the output is still the output defines at process startup.

8.4.4.3 Java usage example

These two previously described features can ease device server debugging. Suppose a device server process is started with the following command line (UNIX command line)

Java -DTANGO_HOST=xxx Perkin/Perkin id22 >/dev/null

This command line does not define any output level. Therefore the default output level is chosen (0) and no message are printed. Sending a SetTraceLevel command requesting level 4 and a SetTraceOutput command with a file name /tmp/server.out will make the device server sending all the output to the /tmp/server.out file **without stopping the process**. The inspection of the /tmp/server.out file will hopefully help to find the reason of the device server problem. When the output are not needed anymore, sending a SetTraceOutput command with the input parameter set to “Initial Output” followed by a SetTraceLevel command with a requested level of 0 will return the server to its original state.

8.4.4.4 C++ utilities to ease device server coding

Some utilities functions have been added in the C++ release to ease Tango device server development. These utilities allow the user to

- Init a C++ vector from a data of one of the Tango DevVarXXXArray data types
- Init a data of one of the Tango::DevVarxxxArray data type from a C++ vector
- Print a data of one of Tango::DevVarxxxArray data type

They mainly used the “<<” operator overloading features. The following code lines are an example of usage of these utilities.

```
1 vector<string> v1;
2 v1.push_back("one");
3 v1.push_back("two");
4 v1.push_back("three");
5
6 Tango::DevVarStringArray s;
7 s << v1;
8 cout << s << endl;
9
10 vector<string> v2;
11 v2 << s;
12
13 for (int i = 0;i < v2.size();i++)
14 cout << "vector element = " << v2[i] << endl;
```

Line 1-4 : Create and Init a C++ string vector
Line 7 : Init a Tango::DevVarStringArray data from the C++ vector
Line 8 : Print all the Tango::DevVarStringArray element in one line of code.
Line 11 : Init a second empty C++ string vector with the content of the Tango::DevVarStringArray

Line 13-14 : Print vector element

Warning: Note that due to a strange behavior of the Windows VC++ compiler compared to other compilers, to use these utilities with the Windows VC++ compiler, you must add the line “using namespace tango” at the beginning of your source file.

8.4.5 Avoiding name conflicts

8.4.5.1 Using C++

Namespace are used to avoid name conflicts. Each device pattern implementation is defined within its own namespace. The name of the namespace is the device pattern class name. In our example, the namespace name is *StepperMotor*.

8.4.5.2 Using Java

Package are used to avoid name conflicts. Each device pattern implementation is defined within its own package. The name of the package is the device pattern class name. In our example, the package name is *StepperMotor*.

8.4.6 The device server main function

A device server main function (or method) always follows the same framework. It exactly implements all the action described in chapter 8.1.7.5. Even if it could be always the same, it has not been included in the library because some linkers are perturbed by the presence of two main functions.

8.4.6.1 Using C++

```
#include <tango.h>

int main(int argc,char *argv[])
{

Tango::Util *tg;

try
{

tg = Tango::Util::init(argc,argv);

tg->server_init();

cout << "Ready to accept request" << endl;
tg->server_run();
}
catch (bad_alloc)
{
cout << "Can’t allocate memory!!!" << endl;
cout << "Exiting" << endl;
}
catch (CORBA::Exception &e)
{
Tango::Except::print_exception(e);
```

```
26
27 cout << "Received a CORBA::Exception" << endl;
28 cout << "Exiting" << endl;
29 }
30
31 tg->server_cleanup();
32
33 return(0);
34 }
```

Line 1 : Include the **tango.h** file. This file is a master include file. It includes several other files. The list of files included by tango.h can be found in [8]

Line 11 : Create the instance of the Tango::Util class (a singleton). Passing argc,argv to this method is mandatory because the device server command line is checked when the Tango::Util object is constructed.

Line 13 : Start all the device pattern creation and initialization with the *server_init()* method

Line 16 : Put the server in a endless waiting loop with the *server_run()* method. In normal case, the process should never returns from this line.

Line 18-22 : Catch all exceptions due to memory allocation error, display a message to the user and exit

Line 23 : Catch all standard TANGO exception which could occur during device pattern creation and initialization

Line 25 : Print exception parameters

Line 27-28 : Print an additional message

Line 31 : Cleanup the server before exiting by calling the *server_cleanup()* method.

8.4.6.2 Using Java

The *main* method can be defined in any class. There is no mandatory class where it should be defined. In our StepperMotor example, the *main* method has been implemented in the StepperMotor class because it is the most logical place.

```
package StepperMotor

import java.util.*;
import org.omg.CORBA.*;
import fr.esrf.Tango.*;
import fr.esrf.TangoDs.*;

public class StepperMotor extends DeviceImpl implements TangoConst
{
public static void main(String[] argv)
{
try
{

Util tg = Util.init(argv,"StepperMotor");

tg.server_init();

System.out.println("Ready to accept request");

```

```
21 tg.server_run();
22 }
23 catch (OutOfMemoryError ex)
24 {
25 System.err.println("Can't allocate memory !!!!");
26 System.err.println("Exiting");
27 }
28 catch (UserException ex)
29 {
30 Except.print_exception(ex);
31
32 System.err.println("Received a CORBA user exception");
33 System.err.println("Exiting");
34 }
35 catch (SystemException ex)
36 {
37 Except.print_exception(ex);
38
39 System.err.println("Received a CORBA system exception");
40 System.err.println("Exiting");
41 }
42
43 System.exit(-1);
44
45 }
46 }
```

---

line 1 : The StepperMotor class is part of the StepperMotor package

Line 3-6 : Import several packages. The reason of importing these package will be explained when the StepperMotor class will be detailed later in this chapter

Line 8 : Definition of the StepperMotor class (will be explained later)

Line 10 : Definition of the *main* method

Line 15 : Create the instance of the Util class (a singleton). Passing argv to this method is mandatory because the device server command line is checked when the Util object is constructed. The second argument of this *init* method is the device server executable name as defined in 8.1.7.1

Line 17 : Start all the device pattern creation and initialization

Line 21 : Put the server in a endless waiting loop. In normal case, the process should never returns from this line.

Line 23-27 : Catch all exceptions due to memory error and display a message to the user. It seems strange to deal with memory allocation error with Java.The Java garbage collection system reclaims memory only for object which have a reference count equal to zero. If, inside a program, objects are created and stay with an object reference count different than zero, they will never be destructed. If many of these objects are created, memory allocation errors can occurs. You may think that the author of this manual is paranoid but have a look at [13]

Line 28-34 : Catch CORBA user exception included the TANGO DevFailed exception which could occur during device pattern creation and initialization

Line 30 : Use the static *print_exception* method of the Except class to print all the data members of the exception object.

Line 35-41 : catch CORBA system exception.

Line 37 : Use the static *print_exception* method of the Except class to print all the data members of the exception object.

Line 43 : Exit the device server

8.4.7 The DServer::class_factory method (C++ specific)

As described in chapter 8.1.7.2, C++ device server needs a *class_factory*() method. This method creates all the device pattern implemented in the device server by calling their *init*() method. The following is an example of a *class_factory* method for a device server with one implementation of the device server pattern for stepper motor device.

```
1 #include <tango.h>
2 #include <steppermotorclass.h>
3
4 void Tango::DServer::class_factory()
5 {
6
7 add_class(StepperMotor::StepperMotorClass::init("StepperMotor"));
8
9 }
```

Line 1 : Include the Tango master include file
Line 2 : Include the steppermotorclass class definition file
Line 7 : Create the StepperMotorClass singleton by calling its *init* method and stores the returned pointer into the DServer object. Remember that all classes for the device pattern implementation for the stepper motor class is defined within a namespace called *StepperMotor*.

8.4.8 Writing the StepperMotorClass class

8.4.8.1 Using C++

8.4.8.1.1 The class definition file

```
#include <tango.h>

namespace StepperMotor
{

class StepperMotorClass : public Tango::DeviceClass
{
public:
static StepperMotorClass *init(const char *);
static StepperMotorClass *instance();
~StepperMotorClass() {_instance = NULL;}

protected:
StepperMotorClass(string &);
static StepperMotorClass *_instance;
void command_factory();
void attribute_factory(vector<Tango::Attr *> &);

public:
void device_factory(const Tango::DevVarStringArray *);
};

} /* End of StepperMotor namespace */
```

Line 1 : Include the Tango master include file

Line 3 : This class is defined within the *StepperMotor* namespace

Line 6 : Class StepperMotorClass inherits from Tango::DeviceClass

Line 9-10 : Definition of the *init* and *instance* methods. These methods are static and can be called even if the object is not already constructed.

Line 11: The destructor

Line 14 : The class constructor. It is protected and can't be called from outside the class. Only the *init* method allows a user to create an instance of this class. See [10] to get details about the singleton design pattern.

Line 15 : The instance pointer. It is static in order to set it to NULL during process initialization phase

Line 16 : Definition of the *command_factory* method

Line 17 : Definition of the *attribute_factory* method

Line 20 : Definition of the *device_factory* method

8.4.8.1.2 The singleton related methods

```
#include <tango.h>

#include <steppermotor.h>
#include <steppermotorclass.h>

namespace StepperMotor
{

StepperMotorClass *StepperMotorClass::_instance = NULL;

StepperMotorClass::StepperMotorClass(string &s):
Tango::DeviceClass(s)
{
INFO_STREAM << "Entering StepperMotorClass constructor" << endl;

INFO_STREAM << "Leaving StepperMotorClass constructor" << endl;
}


StepperMotorClass *StepperMotorClass::init(const char *name)
{
if (_instance == NULL)
{
try
{
string s(name);
_instance = new StepperMotorClass(s);
}
catch (bad_alloc)
{
throw;
}
}
return _instance;
```

```
35 }
36
37 StepperMotorClass *StepperMotorClass::instance()
38 {
39 if (_instance == NULL)
40 {
41 cerr << "Class is not initialised !!" << endl;
42 exit(-1);
43 }
44 return _instance;
45 }
```

Line 1-4 : include files: the Tango master include file (tango.h), the StepperMotorClass class definition file (steppermotorclass.h) and the StepperMotor class definition file (steppermotor.h)

Line 6 : Open the *StepperMotor* namespace.

Line 9 : Initialize the static _instance field of the StepperMotorClass class to NULL

Line 11-18 : The class constructor. It takes an input parameter which is the controlled device class name. This parameter is passed to the constructor of the DeviceClass class. Otherwise, the constructor does nothing except printing a message

Line 20-35 : The *init* method. This method needs an input parameter which is the controlled device class name (StepperMotor in this case). This method checks is the instance is already constructed by testing the _instance data member. If the instance is not constructed, it creates one. If the instance is already constructed, the method simply returns a pointer to it.

Line 37-45 : The *instance* method. This method is very similar to the *init* method except that if the instance is not already constructed. the method print a message and abort the process.

As you can understand, it is not possible to construct more than one instance of the StepperMotorClass (it is a singleton) and the *init* method must be called prior to any other method.

8.4.8.1.3 The command_factory method Within our example, the stepper motor device supports two commands which are called DevReadPosition and DevReadDirection. These two command takes a Tango::DevLong argument as input and output parameter. The first command is created using the inheritance model and the second command is created using the template command model.

```

void StepperMotorClass::command_factory()
{
command_list.push_back(new DevReadPositionCmd("DevReadPosition",
Tango::DEV_LONG,
Tango::DEV_LONG,
"Motor number (0-7)",
"Motor position"));

command_list.push_back(
new TemplCommandInOut<Tango::DevLong,Tango::DevLong>
((const char *)"DevReadDirection",
static_cast<Tango::Lg_CmdMethPtr_Lg>
(&StepperMotor::dev_read_direction),
static_cast<Tango::StateMethPtr>
(&StepperMotor::direct_cmd_allowed))
);
```

```
18 }
19
```

Line 4 : Creation of one instance of the DevReadPositionCmd class. The class is created with five arguments which are the command name, the command type code for its input and output parameters and two strings which are the command input and output parameters description. The pointer returned by the new C++ keyword is added to the vector of available command.

Line 10-14 : Creation of the object used for the DevReadDirection command. This command has one input and output parameter. Therefore the created object is an instance of the TemplCommandInOut class. This class is a C++ template class. The first template parameter is the command input parameter type, the second template parameter is the command output parameter type. The second TemplCommandInOut class constructor parameter (set at line 13) is a pointer to the method to be executed when the command is requested. A casting is necessary to store this pointer as a pointer to a method of the DeviceImpl class[4]. The third TemplCommandInOut class constructor parameter (set at line 15) is a pointer to the method to be executed to check if the command is allowed. This is necessary only if the default behavior (command always allowed) does not fulfill the needs. A casting is necessary to store this pointer as a pointer to a method of the DeviceImpl class. When a command is created using the template command method, the input and output parameters type are determined from the template C++ class parameters.

8.4.8.1.4 The device_factory method The *device_factory* method has one input parameter. It is a pointer to Tango::DevVarStringArray data which is the device name list for this class and the instance of the device server process. This list is fetch from the Tango database.

```
1 void StepperMotorClass::device_factory(const Tango::_DevVarStringArray *devlist_ptr)
2 {
3
4 for (long i = 0;i < devlist_ptr->length();i++)
5 {
6 DEBUG_STREAM << "Device name : " << (*devlist_ptr)[i] << endl;
7
8 device_list.push_back(new StepperMotor(this,
9 (*devlist_ptr)[i]));
10
11 if (Tango::Util::_UseDb == true)
12 export_device(device_list.back());
13 else
14 export_device(device_list.back(),(*devlist_ptr[i]));
15 }
16 }
```

Line 4 : A loop for each device

Line 8 : Create the device object using a StepperMotor class constructor which needs two arguments. These two arguments are a pointer to the StepperMotorClass instance and the device name. The pointer to the constructed object is then added to the device list vector

Line 11-14 : Export device to the outside world using the *export_device* method of the DeviceClass class.

[4]The StepperMotor class inherits from the DeviceImpl class and therefore is a DeviceImpl

8.4.8.1.5 The attribute_factory method The rule of this method is to fulfill a vector of pointer to attributes. A reference to this vector is passed as argument to this method.

```
1 void StepperMotorClass::attribute_factory(vector<Tango::Attr *> &att_list)
2 {
3 att_list.push_back(new PositionAttr());
4
5 Tango::UserDefaultAttrProp def_prop;
6 def_prop.set_label("Set the motor position");
7 def_prop.set_format("scientific;setprecision(4)");
8 Tango::Attr *at = new SetPositionAttr();
9 at->set_default_properties(def_prop);
10 att_list.push_back(at);
11
12 att_list.push_back(new DirectcionAttr());
13 }
```

Line 3 : Create the PositionAttr class and store the pointer to this object into the attribute pointer vector.

Line 5-7 : Create a Tango::UserDefaultAttrProp instance and set the label and format properties default values in this object

Line 8 : Create the SetPositionAttr attribute.

Line 9 : Set attribute user default value with the *set_default_properties()* method of the Tango::Attr class.

Line 10 : Store the pointer to this object into the attribute pointer vector.

Line 12 : Create the DirectionAttr class and store the pointer to this object into the attribute pointer vector.

Please, note that in some rare case, it is necessary to add attribute to this list during the device server life cycle. This *attribute_factory()* method is called once during device server start-up. A method *add_attribute()* of the DeviceImpl class allows the user to add a new attribute to the attribute list outside of this *attribute_factory()* method. See [8] for more information on this method.

8.4.8.2 Using Java

8.4.8.2.1 The singleton related method

```
package StepperMotor;

import java.util.*;
import fr.esrf.Tango.*;
import fr.esrf.TangoDs.*;

public class StepperMotorClass extends DeviceClass implements TangoConst
{
private static StepperMotorClass _instance = null;


public static StepperMotorClass instance()
{
if (_instance == null)
```

```
15 {
16 System.err.println("StepperMotorClass is not initialised !!!");
17 System.err.println("Exiting");
18 System.exit(-1);
19 }
20 return _instance;
21 }
22
23
24 public static StepperMotorClass init(String class_name) throws DevFailed
25 {
26 if (_instance == null)
27 {
28 _instance = new StepperMotorClass(class_name);
29 }
30 return _instance;
31 }
32
33 protected StepperMotorClass(String name) throws DevFailed
34 {
35 super(name);
36
37 Util.out2.println("Entering StepperMotorClass constructor");
38
39 Util.out2.println("Leaving StepperMotorClass constructor");
40 }
41 }
```

Line 1 : This class is part of the StepperMotor package.

Line 3-5 : Import different packages. The first one (**java.lang.util**) is a classical Java package from the JDK. The second one (**fr.esrf.Tango**) is the package generated by the IDL compiler from the Tango IDL file. The last one (**fr.esrf.TangoDs**) is the name of the package with all the root classes of the device server framework.

Line 7 : The StepperMotorClass inherits from the DeviceClass and implements the **TangoConst** interface. The TangoConst interface does not defines any method but simply defines constant variables. The TangoConst interface is a member of the TangoDs package.

Line 9 : The instance pointer. It is static and private. It is initialized to NULL

Line 12-21 : The *instance* method. This method is very similar to the *init* method except that if the instance is not already constructed. the method print a message and abort the process.

Line 24-31: The *init* method. This method needs an input parameter which is the controlled device class name (StepperMotor in this case). This method checks is the instance is already constructed by testing the _instance data member. If the instance is not constructed, it creates one. If the instance is already constructed, the method simply returns a pointer to it.

Line 33-40 : The class constructor which is protected. It takes an input parameter which is the controlled device class name. This parameter is passed to the constructor of the DeviceClass class (line 35). Otherwise, the constructor does nothing except printing a message

As you can understand, it is not possible to construct more than one instance of the StepperMotorClass (it is a singleton) and the *init* method must be called prior to any other method.

8.4.8.2.2 The command_factory method Within our example, the stepper motor device supports two commands which are called DevReadPosition and DevReadDirection. These two command takes a Tango_DevLong argument as input and output parameter. The first command is created using the inheritance model and the second command is created using the template command model.

```
1 public void command_factory()
2 {
3 String str = new String("DevReadPosition");
4 command_list.addElement(new DevReadPositionCmd(str,
5 Tango_DEV_LONG,Tango_DEV_LONG,
6 "Motor number (0-7)",
7 "Motor position"));
8
9 str = new String("DevReadDirection");
10 command_list.addElement(new TemplCommandInOut(str,
11 "dev_read_direction",
12 "direct_cmd_allowed"));
13 }
```

Line 4: Creation of one instance of the DevReadPositionCmd class. The class is created with five arguments which are the command name, the command type code for its input and output parameters and the parameters description (input and output). The Tango_DEV_LONG constant is defined in the TangoConst interface. The reference returned by the *new* Java keyword is added to the vector of available command via the *addElement* method of the Java Vector class.

Line 10-12 : Creation of the object used for the DevReadDirection command. This command has one input and output parameter. Therefore the created object is an instance of the TemplCommandInOut class. The second TemplCommandInOut class constructor parameter (set at line 11) is the method name to be executed when the command is requested. The third TemplCommandInOut class constructor parameter (set at line 12) is the method name to be executed to check if the command is allowed. This is necessary only if the default behavior (command always allowed) does not fulfill the needs. When a command is created using the template command method, the input and output parameter types are determined from the given method declaration.

8.4.8.2.3 The device_factory method The *device_factory* method has one input parameter. It is a pointer to a DevVarStringArray[5] data which is the device name list for this class and the instance of the device server process. This list is fetch from the Tango database.

```
public void device_factory(String[] devlist) throws DevFailed
{
for (int i = 0;i < devlist.length;i++)
{
Util.out4.println("Device name : " + devlist[i]);

device_list.addElement(new StepperMotor(this,
devlist[i],
"A Tango motor",
DevState.ON,
```

[5] DevVarStringArray maps to Java String[]

```
11 "The motor is ON"));
12
13 if (Util.instance()._UseDb == true)
14 export_device(((DeviceImpl)(device_list.lastElement())));
15 else
16 export_device(((DeviceImpl)(device_list.lastElement())),
17 devlist[i]);
18 }
19 }
```

Line 3 : A loop for each device

Line 7 : Create the device object using a StepperMotor class constructor which needs five arguments. These five arguments are a reference to the StepperMotorClass instance, the device name, the device description, the device original state and the device original status. The reference to the constructed object is then added to the device list vector with the *addElement* method of the java.util.Vector class.

Line 13-17 : Export device to the outside world using the *export_device* method of the DeviceClass class. The *lastElement* method of the java.util.Vector class returns a reference to an object of the java Object class. It must be casted before being passed to the *export_device* method

8.4.8.2.4 The attribute_factory method The rule of this method is to fulfill a vector of references to attribute. A reference to this vector is passed to this method. The Tango core classes will use this vector to build all the attributes related objects (An instance of the MultiAttribute class and one Attribute or WAttribute object for each attribute defined in this vector).

```
1 public void attribute_factory(Vector att) throws DevFailed
2 {
3 att.addElement(new Attr("Position",
4 Tango_DEV_LONG,
5 AttrWriteType.READ_WITH_WRITE,
6 "SetPosition"));
7
8 UserDefaultAttrProp def_prop = new UserDefaultAttrProp();
9 def_prop.set_label("set the motor position");
10 def_prop.set_format("scientific;setprecision(4)");
11 Attr at = new Attr("SetPosition",
12 Tango_DEV_LONG,
13 AttrWriteType.WRITE));
14 at.set_default_properties(def_prop);
15 att.addElement(at);
16
17 att.addElement(new Attr("Direction",
18 Tango_DEV_LONG));
19 }
```

Line 3-6 : Build a one dimension attribute of TANGO_DEV_LONG type with an associate writable attribute. Store a reference to this attribute in the vector. In this example, the attribute display type is not defined in the Attr class constructor. Therefore, it will be initialized with its default value (OPERATOR). Several Attr class constructors are defined with or without the attribute display type. See [8] for a complete constructor list.

Line 8-10 : Create a UserDefaultAttrProp instance and set the label and format properties default values in this object

Line 11-13 : Build a one dimension writable attribute.

Line 14 : Set attribute user default value with the *set_default_properties()* method of the Tango::Attr class.

Line 15 : Store the reference to this attribute object into the attribute vector.

Line 17-18 : Build a one dimension attribute. Store the reference to this attribute object into the attribute vector.

Please, note that in some rare case, it is necessary to add attribute to this list during the device server life cycle. This *attribute_factory()* method is called once during device server start-up. A method *add_attribute()* of the DeviceImpl class allows the user to add a new attribute to the attribute list outside of this *attribute_factory()* method. See [8] for more information on this method.

8.4.9 The DevReadPositionCmd class

8.4.9.1 Using C++

8.4.9.1.1 The class definition file

```
1 #include <tango.h>
2
3 namespace StepperMotor
4 {
5
6 class DevReadPositionCmd : public Tango::Command
7 {
8 public:
9 DevReadPositionCmd(const char *,Tango::CmdArgType,
10 Tango::CmdArgType,
11 const char *,const char *);
12 ~DevReadPositionCmd() {};
13
14 virtual bool is_allowed (Tango::DeviceImpl *, const CORBA::Any &);
15 virtual CORBA::Any *execute (Tango::DeviceImpl *, const CORBA::Any &);
16 };
17
18 } /* End of StepperMotor namespace */
```

Line 1 : Include the tango master include file

Line 3 : Open the *StepperMotor* namespace.

Line 6 : The DevReadPositionCmd class inherits from the Tango::Command class

Line 9 : The constructor

Line 12 : The destructor

Line 14 : The definition of the *is_allowed* method. This method is not necessary if the default behavior implemented by the default *is_allowed* method fulfill the requirements. The default behavior is to always allows the command execution (always return true).

Line 15: The definition of the *execute* method

8.4.9.1.2 The class constructor The class constructor does nothing. It simply invoke the Command constructor by passing it its five arguments which are:

1. The command name
2. The command input type code
3. The command output type code
4. The command input parameter description
5. The command output parameter description

With this 5 parameters command class constructor, the command display level is not specified. Therefore it is set to its default value (OPERATOR). If the command does not have input or output parameter, it is not possible to use the Command class constructor defined with five parameters. In this case, the command constructor execute the Command class constructor with three elements (class name, input type, output type) and set the input or output parameter description fields with the *set_in_type_desc* or *set_out_type_desc* Command class methods. To set the command display level, it is possible to use a 6 parameters constructor or it is also possible to set it in the constructor code with the *set_disp_level* method. Many Command class constructors are defined. See [8]for a complete list.

8.4.9.1.3 The is_allowed method In our example, the DevReadPosition command is allowed only if the device is in the ON state. This method receives two argument which are a pointer to the device object on which the command must be executed and a reference to the command input Any object. This method returns a boolean which must be set to true if the command is allowed. If this boolean is set to false, the DeviceClass *command_handler* method will automatically send an exception to the caller.

```
1 bool DevReadPositionCmd::is_allowed(Tango::DeviceImpl *device,
2 const CORBA::Any &in_any)
3 {
4 if (device->get_state() == Tango::ON)
5 return true;
6 else
7 return false;
8 }
```

Line 4 : Call the *get_state* method of the DeviceImpl class which simply returns the device state
Line 5 : Authorize command if the device state is ON
Line 7 : Refuse command execution in all other cases.

8.4.9.1.4 The execute method This method receives two arguments which are a pointer to the device object on which the command must be executed and a reference to the command input Any object. This method returns a pointer to an any object which must be initialized with the data to be returned to the caller.

```
CORBA::Any *DevReadPositionCmd::execute(
Tango::DeviceImpl *device,
const CORBA::Any &in_any)
{
INFO_STREAM << "DevReadPositionCmd::execute(): arrived" << endl;
Tango::DevLong motor;
```

```
7
8 extract(in_any,motor);
9 return insert(
10 (static_cast<StepperMotor *>(device))->dev_read_position(motor));
11 }
```

Line 8 : Extract incoming data from the input any object using a Command class *extract* helper method. If the type of the data in the Any object is not a Tango::DevLong, the *extract* method will throw an exception to the client.

Line 9 : Call the stepper motor object method which execute the DevReadPosition command and insert the returned value into an allocated Any object. The Any object allocation is done by the *insert* method which return a pointer to this Any.

8.4.9.2 Using Java

8.4.9.2.1 The class constructor The class constructor does nothing. It simply invoke the Command constructor by passing it its five arguments which are:

1. The command name
2. The command input type code
3. The command output type code
4. The command input parameter description
5. The command output parameter description

With this 5 parameters command class constructor, the command display level is not specified. Therefore it is set to its default value (OPERATOR). If the command does not have input or output parameter, it is not possible to use the Command class constructor defined with five parameters. In this case, the command constructor execute the Command class constructor with three elements (class name, input type, output type) and set the input or output parameter description fields with the *set_in_type_desc* or *set_out_type_desc* Command class methods. To set the command display level, it is possible to use a 6 parameters constructor or it is also possible to set it in the constructor code with the *set_disp_level* method. Many Command class constructors are defined. See [8]for a complete list.

8.4.9.2.2 The is_allowed method In our example, the DevReadPosition command is allowed only if the device is in the ON state. This method receives two argument which are a reference to the device object on which the command must be executed and a reference to the command input Any object. This method returns a boolean which must be set to true if the command is allowed. If this boolean is set to false, the DeviceClass *command_handler* method will automatically send an exception to the caller.

```
package StepperMotor;

import org.omg.CORBA.*;
import fr.esrf.Tango.*;
import fr.ersf.TangoDs.*;

public class DevReadPositionCmd extends Command implements TangoConst
{
public boolean is_allowed(DeviceImpl dev, Any data_in)
{
```

```
11 if (dev.get_state() == DevState.ON)
12 return(true);
13 else
14 return(false);
15 }
16
17 }
```

Line 1 : This class is part of the StepperMotor package

Line 3-5 : Import different packages. The first one (**org.omg.CORBA**) is a package which contains all the CORBA related classes. The second one (**fr.esrf.Tango**) is the package generated by the IDL compiler from the Tango IDL file. The last one (**fr.ersf.TangoDs**) is the name of the package with all the root classes of the device server pattern.

Line 7 : The DevReadPositionCmd class inherits from the Command class and implements the **TangoConst** interface. The TangoConst interface does not defines any method but simply defines constant variables. The TangoConst interface is a member of the TangoDs package.

Line 11 : Call the *get_state* method of the DeviceImpl class which simply returns a reference to the device state

Line 12 : Authorise command if the device state is ON

Line 14 : Refuse command execution in all other cases.

8.4.9.2.3 The execute method This method receives two arguments which are a reference to the device object on which the command must be executed and a reference to the command input Any object. This method returns a reference to an any object which must be initialized with the data to be returned to the caller.

```
1 public Any execute(DeviceImpl device,Any in_any) throws DevFailed
2 {
3 Util.out2.println("DevReadPositionCmd.execute(): arrived");
4
5 int motor = extract_DevLong(in_any);
6
7 return insert(((StepperMotor)(device)).dev_read_position(motor));
8 }
```

Line 5 : Extract incoming data from the input any object

Line 7 : Call the stepper motor object method which execute the DevReadPosition command, insert its return value into an any and return.

8.4.10 The PositionAttr class

8.4.10.1 Using C++

8.4.10.1.1 The class definition file

```
1 #include <tango.h>
2 #include <steppermotor.h>
3
4 namespace StepperMotor
5 {
6
7
8 class PositionAttr: public Tango::Attr
9 {
10 public:
11 PositionAttr():Attr("Position",
12 Tango::DEV_LONG,
13 Tango::READ_WITH_WRITE,
14 "SetPosition") {};
15 ~PositionAttr() {};
16
17 virtual void read(Tango::DeviceImpl *dev,Tango::Attribute &att)
18 {(static_cast<StepperMotor *>(dev))->read_Position(att);}
19 virtual bool is_allowed(Tango::DeviceImpl *dev,Tango::AttReqType ty)
20 {return (static_cast<StepperMotor *>(dev))->is_Position_allowed(ty);}
21 };
22
23 } /* End of StepperMotor namespace */
24
25 #endif // _STEPPERMOTORCLASS_H
```

Line 1-2 : Include the tango master include file and the steppermotor class definition include file

Line 4 : Open the *StepperMotor* namespace.

Line 8 : The PosiitionAttr class inherits from the Tango::Attr class

Line 11-14 : The constructor with 4 arguments

Line 15 : The destructor

Line 17 : The definition of the *read* method. This method forwards the call to a StepperMotor class method called *read_Position()*

Line 19 : The definition of the *is_allowed* method. This method is not necessary if the default behaviour implemented by the default *is_allowed* method fulfills the requirements. The default behaviour is to always allows the attribute reading (always return true). This method forwards the call to a StepperMotor class method called *is_Position_allowed()*

8.4.10.1.2 The class constructor The class constructor does nothing. It simply invoke the Attr constructor by passing it its four arguments which are:

1. The attribute name
2. The attribute data type code
3. The attribute writable type code
4. The name of the associated write attribute

With this 4 parameters Attr class constructor, the attribute display level is not specified. Therefore it is set to its default value (OPERATOR). To set the attribute display level, it is possible to use in the constructor code the *set_disp_level* method. Many Attr class constructors are defined. See [8]for a complete list.

This Position attribute is a scalar attribute. For spectrum attribute, instead of inheriting from the Attr class, the class must inherits from the SpectrumAttr class. Many SpectrumAttr class constructors are defined. See [8]for a complete list.

For Image attribute, instead of inheriting from the Attr class, the class must inherits from the ImageAttr class. Many ImageAttr class constructors are defined. See [8]for a complete list.

8.4.10.1.3 The is_allowed method This method receives two argument which are a pointer to the device object to which the attribute belongs to and the type of request (read or write). In the PositionAttr class, this method simply "forwards" the request to a method of the StepperMotor class called *is_Position_allowed()* passing the request type to this method. This method returns a boolean which must be set to true if the attribute is allowed. If this boolean is set to false, the DeviceImpl read_attribute method will automatically send an exception to the caller.

8.4.10.1.4 The read method This method receives two arguments which are a pointer to the device object to which the attribute belongs to and a reference to the corresponding attribute object. This method "forwards" the request to a StepperMotor class called *read_Position()* passing it the reference on the attribute object.

8.4.11 The StepperMotor class

8.4.11.1 Using C++

8.4.11.1.1 The class definition file

```
#include <tango.h>

#define AGSM_MAX_MOTORS 8 // maximum number of motors per device

namespace StepperMotor
{

class StepperMotor: public Tango::DeviceImpl
{
public :
    StepperMotor(Tango::DeviceClass *,string &);
    StepperMotor(Tango::DeviceClass *,const char *);
    StepperMotor(Tango::DeviceClass *,const char *,const char *);
    ~StepperMotor() {};

    DevLong dev_read_position(DevLong);
    DevLong dev_read_direction(DevLong);
    bool direct_cmd_allowed(const CORBA::Any &);

    virtual Tango::DevState dev_state();
    virtual Tango::ConstDevString dev_status();

    virtual void always_executed_hook();

    virtual void read_attr_hardware(vector<long> &attr_list);

    void read_position(Tango::Attribute &);
    bool is_Position_allowed(Tango::AttReqType req);
```

```
29     void write_SetPosition(Tango::WAttribute &);
30     void read_Direction(Tango::Attribute &);
31
32     virtual void init_device();
33     virtual void delete_device();
34
35     void get_device_properties();
36
37 protected :
38     long axis[AGSM_MAX_MOTORS];
39     DevLong position[AGSM_MAX_MOTORS];
40     DevLong direction[AGSM_MAX_MOTORS];
41     long state[AGSM_MAX_MOTORS];
42
43     Tango::DevLong *attr_Position_read;
44     Tango::DevLong *attr_Direction_read;
45     Tango::DevLong attr_SetPosition_write;
46
47     Tango::DevLong min;
48     Tango::DevLong max;
49
50     Tango::DevLong *ptr;
51 };
52
53 } /* End of StepperMotor namespace */
```

Line 1 : Include the Tango master include file

Line 5 : Open the *StepperMotor* namespace.

Line 8 : The StepperMotor class inherits from the DeviceImpl class

Line 11-13 : Three different object constructors

Line 14 : The destructor which calls the *delete_device()* method

Line 16 : The method to be called for the execution of the DevReadPosition command. This method must be declared as virtual if it is needed to redefine it in a class inheriting from StepperMotor. See chapter 8.7.2 for more details about inheriting.

Line 17 : The method to be called for the execution of the DevReadDirection command

Line 18 : The method called to check if the execution of the DevReadDirection command is allowed. This method is necessary because the DevReadDirection command is created using the template command method and the default behavior is not acceptable

Line 20 : Redefinition of the *dev_state*. This method is used by the State command

Line 21 : Redefinition of the *dev_status*. This method is used by the Status command

Line 23 : Redefinition of the *always_executed_hook* method. This method is the place to code mandatory action which must be executed prior to any command.

Line 25-30 : Attribute related methods

Line 32 : Definition of the *init_device* method.

Line 33 : Definition of the *delete_device* method

Line 35 : Definition of the *get_device_properties* method

Line 38-50 : Data members.

Line 43-44 : Pointers to data for readable attributes Position and Direction

Line 45 : Data for the SetPosition attribute

Line 47-48 : Data members for the two device properties

8.4.11.1.2 The constructors Three constructors are defined here. It is not mandatory to defined three constructors. But at least one is mandatory. The three constructors take a pointer to the StepperMotorClass instance as first parameter[6]. The second parameter is the device name as a C++ string or as a classical pointer to char array. The third parameter necessary only for the third form of constructor is the device description string passed as a classical pointer to a char array.

```
#include <tango.h>
#include <steppermotor.h>

namespace StepperMotor
{

StepperMotor::StepperMotor(Tango::DeviceClass *cl,string &s)
:Tango::DeviceImpl(cl,s.c_str())
{
init_device();
}

StepperMotor::StepperMotor(Tango::DeviceClass *cl,const char *s)
:Tango::DeviceImpl(cl,s)
{
init_device();
}

StepperMotor::StepperMotor(Tango::DeviceClass *cl,const char *s,const char *d)
:Tango::DeviceImpl(cl,s,d)
{
init_device();
}

void StepperMotor::init_device()
{
cout << "StepperMotor::StepperMotor() create " << device_name << endl;

long i;

for (i=0; i< AGSM_MAX_MOTORS; i++)
{
axis[i] = 0;
position[i] = 0;
direction[i] = 0;
}

ptr = new Tango::DevLong[10];

get_device_properties();
}

void StepperMotor::delete_device()
{
```

[6] The StepperMotorClass inherits from the DeviceClass and therefore is a DeviceClass

```
45 delete [] ptr;
46 }
```

Line 1-2 : Include the Tango master include file (tango.h) and the StepperMotor class definition file (steppermotor.h)

Line 4 : Open the *StepperMotor* namespace

Line 7-11 : The first form of the class constructor. It execute the Tango::DeviceImpl class constructor with the two parameters. Note that the device name passed to this constructor as a C++ string is passed to the Tango::DeviceImpl constructor as a classical C string. Then the *init_device* method is executed.

Line 13-17 : The second form of the class constructor. It execute the Tango::DeviceImpl class constructor with its two parameters. Then the *init_device* method is executed.

Line 19-23: The third form of constructor. Again, it execute the Tango::DeviceImpl class constructor with its three parameters. Then the *init_device* method is executed.

Line 25-41 : The *init_device* method. All the device data initialization is done in this method. The device properties are also retrieved from database with a call to the *get_device_properties* method at line 40. The device data member called *ptr* is initialized with allocated memory at line 38. It is not needed to have this pointer, it has been added only for educational purpose.

Line 43-46 : The *delete_device* method. The rule of this method is to free memory allocated in the *init_device* method. In our case , only the device data member *ptr* is allocated in the *init_device* method. Therefore, its memory is freed at line 45. This method is called by the automatically added Init command before it calls the *init_device* method. It is also called by the device destructor.

8.4.11.1.3 The methods used for the DevReadDirection command The DevReadDirection command is created using the template command method. Therefore, there is no specific class needed for this command but only one object of the TemplCommandInOut class. This command needs two methods which are the *dev_read_direction* method and the *direct_cmd_allowed* method. The *direct_cmd_allowed* method defines here implements exactly the same behavior than the default one. This method has been used only for pedagogic issue. The *dev_read_direction* method will be executed by the *execute* method of the TemplCommandInOut class. The *direct_cmd_allowed* method will be executed by the *is_allowed* method of the TemplCommandInOut class.

```
DevLong StepperMotor::dev_read_direction(DevLong axis)
{
if (axis < 0 || axis > AGSM_MAX_MOTORS)
{
WARNING_STREAM << "Steppermotor::dev_read_direction(): axis out of range !";
WARNING_STREAM << endl;
TangoSys_OMemStream o;

o << "Axis number " << axis << " out of range" << ends;
throw_exception((const char *)"StepperMotor_OutOfRange",
o.str(),
(const char *)"StepperMotor::dev_read_direction");
}

return direction[axis];
}


bool StepperMotor::direct_cmd_allowed(const CORBA::Any &in_data)
```

```
20 {
21 INFO_STREAM << "In direct_cmd_allowed() method" << endl;
22
23 return true;
24 }
25
```

Line 1-16 : The *dev_read_direction* method

Line 5-12 : Throw exception to client if the received axis number is out of range

Line 7 : A TangoSys_OMemStream is used as stream. The TangoSys_OMemStream has been defined in improve portability across platform. For Unix like operating system, it is a ostrtream type. For operating system with a full implementation of the standard library, it is a ostringstream type.

Line 19-24 : The *direct_cmd_allowed* method. The command input data is passed to this method in case of it is needed to take the decision. This data is still packed into the CORBA Any object.

8.4.11.1.4 The methods used for the Position attribute To enable reading of attributes, the StepperMotor class must re-define two or three methods called *read_attr_hardware(), read_<Attribute_name>()* and if necessary a method called
is_<Attribute_name>_allowed(). The aim of the first one is to read the hardware. It will be called only once at the beginning of each read_attribute CORBA call. The second method aim is to build the exact data for the wanted attribute and to store this value into the Attribute object. Special care has been taken in order to minimize the number of data copy and allocation. The data passed to the Attribute object as attribute value is passed using pointers. It must be allocated by the method[7] and the Attribute object will not free this memory. Data members called attr_<Attribute_name>_read are foreseen for this usage. The *read_attr_hardware()* method receives a vector of long which are indexes into the main attributes vector of the attributes to be read. The *read_Position()* method receives a reference to the Attribute object. The third method (*is_Position_allowed()*) aim is to allow or dis-allow, the attribute reading. In some cases, some attributes can be read only if some conditions are met. If this method returns true, the *read_<Attribute_name>()* method will be called. Otherwise, an error will be generated for the attribute. This method receives one argument which is an emumeration describing the attribute request type (read or write). In our example, the reading of the Position attribute is allowed only if the device state is ON.

```
void StepperMotor::read_attr_hardware(vector<long> &attr_list)
{
INFO_STREAM << "In read_attr_hardware for " << attr_list.size();
INFO_STREAM << " attribute(s)" << endl;

for (long i = 0;i < attr_list.size();i++)
{
string attr_name;
attr_name = dev_attr->get_attr_by_ind(attr_list[i]).get_name();

if (attr_name == "Position")
{
attr_Position_read = &(position[0]);
}
else if (attr_name == "Direction")
{
attr_Direction_read = &(direction[0]);
```

[7] It can also be data declared as object data members or memory declared as static

```
18 }
19 }
20 }
21
22 void read_Position(Tango::Attribute &att)
23 {
24 att.set_value(attr_Position_read);
25 }
26
27 bool is_Position_allowed(Tango::AttReqType req)
28 {
29 if (req == Tango::WRITE_REQ)
30 return false;
31 else
32 {
33 if (get_state() == Tango::ON)
34 return true;
35 else
36 return false;
37 }
38 }
```

Line 6 : A loop on each attribute to be read

Line 9 : Get attribute name

Line 11 : Test on attribute name

Line 13 : Read hardware (pretty simple in our case)

Line 24 : Set attribute value in Attribute object using the *set_value()* method. This method will also initializes the attribute quality factor to Tango::ATTR_VALID if no alarm level are defined and will set the attribute returned date. It is also possible to use a method called *set_value_date_quality()* which allows the user to set the attribute quality factor as well as the attribute date.

Line 33 : Test on device state

8.4.11.1.5 The methods used for the SetPosition attribute To enable writing of attributes, the StepperMotor class must re-define one or two methods called *write_<Attribute_name>()* and if necessary a method called *is_<Attribute_name>_allowed()*. The aim of the first one is to write the hardware. The *write_Position()* method receives a reference to the WAttribute object. The value to write is in this WAttribute object. The third method (*is_Position_allowed()*) aim is to allow or dis-allow, the attribute writing. In some cases, some attributes can be write only if some conditions are met. If this method returns true, the *write_<Attribute_name>()* method will be called. Otherwise, an error will be generated for the attribute. This method receives one argument which is an emumeration describing the attribute request type (read or write). For read/write attribute, this method is the same for reading and writing. The input argument value makes the difference.

For our example, it is always possible to write the SetPosition attribute. Therefore, the StepperMotor class only defines a *write_SetPosition()* method.

```
void StepperMotor::write_SetPosition(Tango::WAttribute &att)
{
att.get_write_value(sttr_SetPosition_write);

INFO_STREAM << "Attribute SetPosition value = ";
```

```
6 INFO_STREAM << attr_SetPosition_write << endl;
7
8 position[0] = attr_SetPosition_write;
9 }
```

Line 3 : Retrieve new attribute value
Line 5-6 : Send some messages using Tango Logging system
Line 8 : Set the hardware (pretty simple in our case)

8.4.11.1.6 Retrieving device properties Retrieving properties is fairly simple with the use of the database object. Each Tango device is an aggregate with a DbDevice object (see figure 8.1). This has been grouped in a method called *get_device_properties*(). The classes and methods of the Dbxxx objects are described in the Tango API documentation.

```
1 void DocDs::get_device_property()
2 {
3 Tango::DbData data;
4 data.push_back(DbDatum("Max"));
5 data.push_back(DbDatum("Min"));
6
7 get_db_device()->get_property(data);
8
9 if (data[0].is_empty()==false)
10 data[0] >> max;
11 if (data[1].is_empty()==false)
12 data[1] >> min;
13 }
```

Line 4-5 : Two DbDatum (one per property) are stored into a DbData object
Line 7 : Call the database to retrieve properties value
Line 9-10 : If the Max property is defined in the database, extract its value from the DbDatum object and store it in a device data member
Line 11-12 : If the Min property is defined in the database, extract its value from the DbDatum object and store it in a device data member

8.4.11.1.7 The remaining methods The remaining methods are the *dev_state, dev_status, always_executed_hook*, *dev_read_position* and *read_Direction()* methods. The *dev_state* method parameters are fixed. It does not receive any input parameter and must return a Tango_DevState data type. The *dev_status* parameters are also fixed. It does not receive any input parameter and must return a Tango string. The *always_executed_hook* receives nothing and return nothing. The *dev_read_position* method input parameter is the motor number as a long and the returned parameter is the motor position also as a long data type. The *read_Direction()* method is the method for reading the Direction attribute.

```
1 DevLong StepperMotor::dev_read_position(DevLong axis)
2 {
3
4 if (axis < 0 || axis > AGSM_MAX_MOTORS)
```

```
5 {
6 WARNING_STREAM << "Steppermotor::dev_read_position(): axis out of range !";
7 WARNING_STREAM << endl;
8
9 TangoSys_OMemStream o;
10
11 o << "Axis number " << axis << " out of range" << ends;
12 throw_exception((const char *)"StepperMotor_OutOfRange",
13 o.str(),
14 (const char *)"StepperMotor::dev_read_position");
15 }
16
17 return position[axis];
18 }
19
20 void always_executed_hook()
21 {
22 INFO_STREAM << "In the always_executed_hook method << endl;
23 }
24
25 Tango_DevState StepperMotor::dev_state()
26 {
27 INFO_STREAM << "In StepperMotor state command" << endl;
28 return DeviceImpl::dev_state();
29 }
30
31 Tango_DevString StepperMotor::dev_status()
32 {
33 INFO_STREAM << "In StepperMotor status command" << endl;
34 return DeviceImpl::dev_status();
35 }
36
37 void read_Direction(Tango::Attribute att)
38 {
39 att.set_value(attr_Direction_read);
40 }
```

Line 1-18 : The *dev_read_position* method

Line 6-14 : Throw exception to client if the received axis number is out of range

Line 9 : A TangoSys_OMemStream is used as stream. The TangoSys_OMemStream has been defined in improve portability across platform. For Unix like operating system, it is a ostrtream type. For operating system with a full implementation of the standard library, it is a ostringstream type.

Line 20-23 : The *always_executed_hook* method. It does nothing. It has been included here only as pedagogic usage.

Line 25-29 : The *dev_state* method. It does exactly what the default *dev_state* does. It has been included here only as pedagogic usage

Line 31-35 : The *dev_status* method. It does exactly what the default *dev_statu*s does. It has been included here only as pedagogic usage

Line 37-40 : The *read_Direction* method. Simply set the Attribute object internal value

8.4.11.2 Using Java

8.4.11.2.1 The constructor The constructor take a reference to the StepperMotorClass instance as first parameter[8]. The second parameter is the device name as a Java string.

```
package StepperMotor;

import java.util.*;
import org.omg.CORBA.*;
import fr.esrf.Tango.*;
import fr.esrf.TangoDs.*;

public class StepperMotor extends DeviceImpl implements TangoConst
{
protected final int SM_MAX_MOTORS = 8;

protected int[] axis = new int[SM_MAX_MOTORS];
protected int[] position = new int[SM_MAX_MOTORS];
protected int[] direction = new int[SM_MAX_MOTORS];
protected int[] state = new int[SM_MAX_MOTORS];

protected int[] attr_Direction_read = new int[1];
protected int[] attr_Position_read = new int[1];
protected int attr_SetPosition_write;


StepperMotor(DeviceClass cl,String s,String desc,
DevState state,String status) throws DevFailed
{
super(cl,s,desc,state,status);
init_device();
}

public void init_device()
{
System.out.println("StepperMotor() create motor " + dev_name);

int i;

for (i=0; i< SM_MAX_MOTORS; i++)
{
axis[i] = 0;
position[i] = 0;
direction[i] = 0;
state[i] = 0;
}

}
}
```

[8]The StepperMotorClass inherits from the DeviceClass and therefore is a DeviceClass

Line 3-6: Import different packages. The first one (**java.lang.util**) is a classical Java package from the JDK. The second one (**org.omg.CORBA**) is a package which contains all the CORBA related classes. The third one (**fr.esrf.Tango**) is the package generated by the IDL compiler from the Tango IDL file. The last one (**fr.esrf.TangoDs**) is the name of the package with all the root classes of the device server pattern.

Line 8 : The StepperMotor class inherits from the DeviceImpl class and implements the **TangoConst** interface. The TangoConst interface does not defines any method but simply defines constant variables. The TangoConst interface is a member of the TangoDs package.

Line 10 : Define an internal constant

Line 12-15 : Device internal variable

Line 17-19 : Device internal variable linked to attributes

Line 22-27 : The class constructor. It execute the DeviceImpl class constructor with five parameters. Then the *init_device* method is executed.

Line 29-43 : The *init_device* method. All the device data initialization is done in this method.

8.4.11.2.2 The methods used for the DevReadDirection command The DevReadDirection command is created using the template command method. Therefore, there is no specific class needed for this command but only one object of the TemplCommandInOut class. This command needs two methods which are the *dev_read_direction* method and the *direct_cmd_allowed* method. The *direct_cmd_allowed* method defines here implements exactly the same behavior than the default one. This method has been used only for pedagogic issue. The *dev_read_direction* method will be executed by the *execute* method of the TemplCommandInOut class. The *direct_cmd_allowed* method will be executed by the *is_allowed* method of the TemplCommandInOut class.

```
1 public int dev_read_direction(int axis) throws DevFailed
2 {
3 if (axis < 0 || axis > SM_MAX_MOTORS)
4 {
5 Util.out1.println("Steppermotor.dev_read_direction(): axis out of range !");
6
7 StringBuffer o = new StringBuffer("Axis number ");
8 o.append(axis);
9 o.append(" out of range");
10
11 Except.throw_exception("StepperMotor_AxisOutOfRange",
12 o.toString(),
13 "StepperMotor.dev_read_direction()");
14 }
15
16 return direction[axis];
17 }
18
19 public boolean direct_cmd_allowed(Any data_in)
20 {
21 Util.out2.println("In StepperMotor.direct_cmd_allowed method");
22
23 return true;
24 }
```

Line 1-17 : The *dev_read_direction* method

Line 3-14 : Throw exception to client if the received axis number is out of range

Line 19-24 : The *direct_cmd_allowed* method. The command input data is passed to this method in case of it is needed to take the decision. This data is still packed into the CORBA Any object.

8.4.11.2.3 The write attribute related method To enable writing of writable attributes, the StepperMotor class must re-define a method called *write_attr_hardware()*. The aim of this method is to write the hardware. This method receives a vector of Integer objects as parameters. These data are the indexes of the attributes to be written into the main attribute vector stored in the MultiAttribute object. Methods of the MultiAttribute class allow the retrieval of the the correct attribute object from these indexes. The value to be written is stored in the WAttribute object and can be retrieved with WAttribute class methods called *get_xx_write_value()*. A data member called *attr_<Attribute_name>_write* is foreseen to temporary store this extracted value.

```
1 public void write_attr_hardware(Vector attr_list)
2 {
3 Util.out2.println("In write_attr_hardware for "+attr_list.size()+" attribute(s)");
4
5 for (int i = 0;i < attr_list.size();i++)
6 {
7 int ind = ((Integer)(attr_list.elementAt(i))).intValue();
8 WAttribute att = dev_attr.get_w_attr_by_ind(ind);
9 String att_name = att.get_name();
10
11 if (att_name.equals("SetPosition") == true)
12 {
13 attr_SetPosition_write = att.get_lg_write_value();
14 Util.out2.println("Attribute SetPosition value = "+attr_SetPosition_write);
15 position[0] = attr_SetPosition_write;
16 }
17 }
18 }
```

Line 5 : A loop on each attribute to be written
Line 7-9 : Retrieve attribute name
Line 11 : A test on attribute name
Line 13 : Retrieve new attribute value
Line 15 : Set the hardware (very simple in our example)

8.4.11.2.4 The read attribute related methods To enable reading of attributes, the StepperMotor class must re-define two methods called *read_attr_hardware()* and *read_attr()*. The aim of the first one is to read the hardware. It will be called only once at the beginning of each read_attributes CORBA call. The second method aim is to build the exact data for the wanted attribute and to store this value into the Attribute object. This method will be called for each attribute to be read. Special care has been taken in order to minimize the number of data copy and allocation. The data passed to the Attribute object as attribute value is passed using pointers. It must be allocated by the method[9] and the Attribute object will not free this memory. Data members called attr_<Attribute_name>_read are foreseen for this usage. As for the *write_attr_hardware()* method, the *read_attr_hardware()* method receives a vector of long which are indexes into the main attributes vector of the attributes to be read. The *read_attr()* method receives a reference to the Attribute object.

[9] It can also be data declared as object data members or memory declared as static

```
1 public void read_attr_hardware(Vector attr_list)
2 {
3 Util.out2.println("In read_attr_hardware for "+attr_list.size()+" attribute(s)");
4 for (int i = 0;i< attr_list.size();i++)
5 {
6 int ind = ((Integer)(attr_list.elementAt(i))).intValue();
7 String attr_name = dev_attr.get_attr_by_ind(ind).get_name();
8
9 if (attr_name == "Position")
10 {
11 attr_Position_read[0] = position[0];
12 }
13 else if (attr_name == "Direction")
14 {
15 attr_Direction_read[0] = direction[0];
16 }
17 }
18 }
19
20
21 public void read_attr(Attribute attr) throws DevFailed
22 {
23 String attr_name = attr.get_name();
24 Util.out2.println("In read_attr for attribute "+attr_name);
25 if (attr_name.equals("Position") == true)
26 {
27 attr.set_value(attr_Position_read);
28 }
29 else if (attr_name.equals("Direction") == true)
30 {
31 attr.set_value(attr_Direction_read);
32 }
33 }
```

Line 4 : A loop on each attribute to be read
Line 6 -7: Get attribute name
Line9 : Test on attribute name
Line 11 : Read hardware (pretty simple in our case)
Line 23 : Get attribute name
Line 25 : Test on attribute name
Line 27 : Set attribute value in Attribute object

8.4.11.2.5 Retrieving device properties Retrieving properties is fairly simple with the use of the database object. Each Tango device is an aggregate with a DbDevice object (see figure 8.1). This has been grouped in a method called *get_device_properties*(). The classes and methods of the Dbxxx objects are described in the Tango API documentation.

```
1 void public get_device_property() throws DevFailed
2 {
3 String[] prop_names = {"Max","Min"};
```

```
4
5 DbDatum[] res_value = db_dev.get_property(prop_names);
6
7 if (res_value[0].is_empty() == false)
8 min = res_value[0].extractInt();
9 if (res_value[1].is_empty() == false)
10 max = res_value[1].extractInt();
11 }
```

Line 3 : Define the names of the properties to be retrieved

Line 5 : Call the database to retrieve properties value

Line 7-8 : If the Max property is defined in the database, extract its value from the DbDatum object and store it in a device data member

Line 9-10 : If the Min property is defined in the database, extract its value from the DbDatum object and store it in a device data member

8.4.11.2.6 The remaining methods The remaining methods are the *dev_state, dev_status, always_executed_hook()* and *dev_read_position* methods. The *dev_state* method parameters are fixed. It does not receive any input parameter and must return a DevState data type. The *dev_status* parameters are also fixed. It does not receive any input parameter and must return reference to a Java string. The *always_executed_hook* receives nothing and return nothing The *dev_read_position* method input parameter is the motor number as an int and the returned parameter is the motor position also as an int data type.

```
int dev_read_position(int axis) throws DevFailed
{

if (axis < 0 || axis > SM_MAX_MOTORS)
{
Util.out1.println("Steppermotor.dev_read_position(): axis out of range !");

StringBuffer o = new StringBuffer("Axis number ");
o.append(axis);
o.append(" out of range");

Except.throw_exception("StepperMotor_AxisOutOfRange",
o.toString(),
"StepperMotor.dev_read_position()");
}

return position[axis];
}

public void always_executed_hook()
{
Util.out2.println("In always_executed_hook method");
}

public DevState dev_state() throws DevFailed
{
Util.out2.println("In StepperMotor state command");
```

```
28 return super.dev_state();
29 }
30
31 public String dev_status() throws DevFailed
32 {
33 Util.out2.println("In StepperMotor status command");
34 return super.dev_status();
35 }
```

Line 1-18 : The *dev_read_position* method

Line 4-15 : Throw exception to client if the received axis number is out of range

Line 20-23 : The *always_executed_hook* method.It does nothing. It has been included here only as pedagogic usage.

Line 25-29 : The *dev_state* method. It does exactly what the default *dev_state* does. It has been included here only as pedagogic usage

Line 31-35 : The *dev_status* method. It does exactly what the default *dev_status* does. It has been included here only as pedagogic usage.

8.5 Device server under Windows

Two kind of programs are available under Windows. These kinds of programs are called console application or Windows application. A console application is started from a MS-DOS window and is very similar to classical UNIX program. A Windows application is most of the time not started from a MS-DOS window and is generally a graphical application without standard input/output. Writing a device server in a console application is straight forward following the rules described in the previous sub-chapters. Writing a device server in a Windows application needs some changes detailed in the following sub-chapters.

8.5.1 The Tango device server graphical interface

Within the Windows operating system, most of the running application has a window user interface. This is also true for the Windows Tango device server. Using or not this interface is up to the device server programmer. The choice is done with an argument to the *server_init()* method of the Tango::Util class. This interface is pretty simple and is based on three windows which are :

- The device server main window
- The device server console window
- The device server help window

8.5.1.1 The device server main window

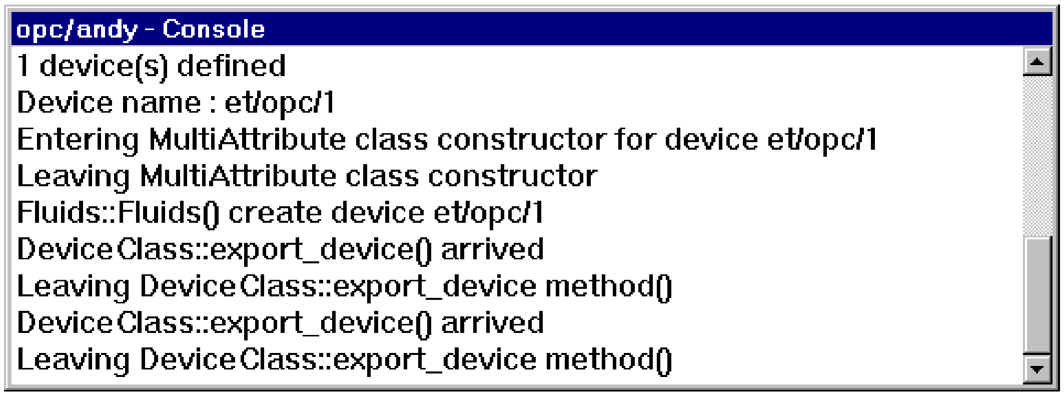
This window looks like :

Four menus are available in this window. The File menu allows the user to exit the device server. The View menu allows you to display/hide the device server console window. The Debug menu allows the user to change the server output verbose level. All the outputs goes to the console window even if it is hidden. The Help menu displays the help window. The device server name is displayed in the window title. The text displayed at the bottom of the window has a default value (the one displayed in this window dump) but may be changed by the device server programmer using the *set_main_window_text()* method of the Tango::Util class. If used, this method must be called prior to the call of the *server_init()* method. Refer to [8] for a complete description of this method.

Figure 8.7: Tango device server main window

8.5.1.2 The console window

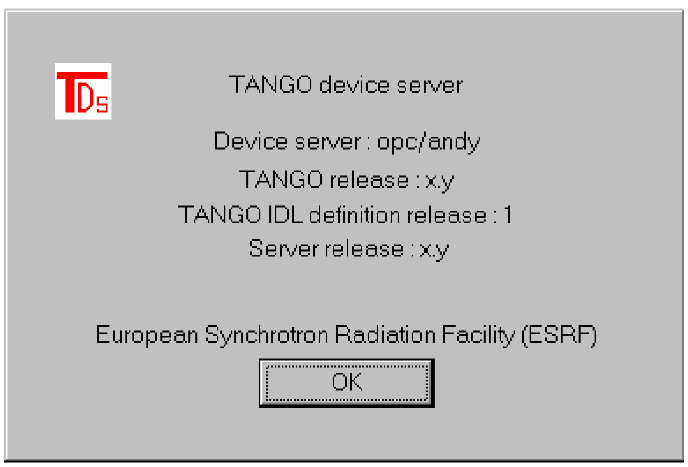
This window looks like :



It simply displays all the logging message when a console target is used in the device server.

8.5.1.3 The help window

This window looks like :



This window displays

- The device server name
- The Tango library release
- The Tango IDL definition release
- The device server release. The device server programmer may set this release number using the *set_server_version()* method of the Tango::Util class. If used, this must be done prior to the call of the *server_init()* method. If the *set_server_version()* method is not used, x.y is displays as version number. Refer to [8] for a complete description of this method.

8.5.2 MFC device server

There is no *main* function within a classical MFC program. Most of the time, your application is represented by one instance of a C++ class which inherits from the MFC CWinApp class. This CWinApp class has several methods that you may overload in your application class. For a device server to run correctly, you must overload two methods of the CWinApp class. These methods are the *InitInstance()* and *ExitInstance()* methods. The rule of these methods is obvious following their names.

Remember that if the Tango device server graphical user interface is used, you must link your device server with the Tango windows resource file. This is done by adding the Tango resource file to the Project Settings/Link/Input/Object, library modules window in VC++.

8.5.2.1 The InitInstance method

The code to be added here is the equivalent of the code written in a classical *main()* function. Don't forget to add the *tango.h* file in the list of included files.

```
BOOL FluidsApp::InitInstance()
{
AfxEnableControlContainer();

// Standard initialization
// If you are not using these features and wish to reduce the size
// of your final executable, you should remove from the following
// the specific initialization routines you do not need.

#ifdef _AFXDLL
Enable3dControls(); // Call this when using MFC in a shared DLL
#else
Enable3dControlsStatic(); // Call this when linking to MFC statically
#endif
Tango::Util *tg;
try
{

tg = Tango::Util::init(m_hInstance,m_nCmdShow);

tg->server_init(true);

tg->server_run();

}
catch (bad_alloc)
{
MessageBox((HWND)NULL,"Memory error","Command line",MB_ICONSTOP);
return(FALSE);
}
catch (Tango::DevFailed &e)
{
MessageBox((HWND)NULL,,e.errors[0].desc.in(),"Command line",MB_ICONSTOP);
return(FALSE);
}
catch (CORBA::Exception &)
{
```

```
38 MessageBox((HWND)NULL,"Exception CORBA","Command line",MB_ICONSTOP);
39 return(FALSE);
40 }
41
42 m_pMainWnd = new CWnd;
43 m_pMainWnd->Attach(tg->get_ds_main_window());
44
45 return TRUE;
46 }
```

Line 19 : Initialise Tango system. This method also analises the argument used in command line.

Line 21 : Create Tango classes requesting the Tango Windows graphical interface to be used

Line 23 : Start Network listener. Note that under NT, this call returns in the contrary of UNIX like operating system.

Line 26-30 : Display a message box in case of memory allocation error and leave method with a return value set to false in order to stop the process

Line 31-35 : Display a message box in case of error during server initialization phase.

Line 36-40 : Display a message box in case of error other than memory allocation. Leave method with a return value set to false in order to stop the process.

Line 37-38 : Create a MFC main window and attach the Tango graphical interface main window to this MFC window.

8.5.2.2 The ExitInstance method

This method is called when the application is stopped. For Tango device server, its rule is to destroy the Tango::Util singleton if this one has been correctly constructed.

```
int FluidsApp::ExitInstance()
{
bool del = true;

try
{
Tango::Util *tg = Tango::Util::instance();
}
catch(Tango::DevFailed)
{
del = false;
}

if (del == true)
delete (Tango::Util::instance());

return CWinApp::ExitInstance();
}
```

Line 7 : Try to retrieve the Tango::Util singleton. If this one has not been constructed correctly, this call will throw an exception.

Line 9-12 : Catch the exception in case of incomplete Tango::Util singleton construction

Line 14-15 : Delete the Tango::Util singleton. This will unregister the Tango device server from the Tango database.

Line 17 : Execute the *ExitInstance* method of the CWinApp class.

If you don't want to use the Tango device server graphical interface, do not pass any parameter to the *server_init()* method and instead of the code display in lines 37 and 38 in the previous example of the *InitInstance()* method, use your own code to initialize your own application.

8.5.2.3 Example of how to build a Windows device server MFC based

This sub-chapter gives an example of what it is needed to do to build a MFC Windows device server. Rather than being a list of actions to strictly follow, this is some general rules of how using VC++ to build a Tango device server using MFC.

1. Create your device server using Pogo. For a class named MyMotor, the following files will be needed : *class_factory.cpp*, *MyMotorClass.h*, *MyMotorClass.cpp*, *MyMotor.h* and *MyMotor.cpp*.

2. On a Windows computer running VC++, create a new project of type "MFC app Wizard (exe)" using static MFC libs. Ask for a dialog based project without ActiveX controls.

3. Copy the five files generated by Pogo to the Windows computer and add them to your project

4. Remove the dialog window files (xxxDlg.cpp and xxxDlg.h), the Resource include file and the resource script file from your project

5. Add #include <stdafx.h> as first line of the include files list in *class_factory.cpp*, *MyMotorClass.cpp* and *MyMotor.cpp* file. Also add your own directory and the Tango include directory to the project pre-compiler include directories list.

6. Enable RTTI in your project settings (see chapter 8.6.1.2)

7. Change your application class:

   (a) Add the definition of an *ExitInstance* method in the declaration file. (xxx.h file)

   (b) Remove the include of the dialog window file in the xxx.cpp file and add an include of the Tango master include files (tango.h)

   (c) Replace the *InitInstance*() method as described in previous sub-chapter. (xx.cpp file)

   (d) Add an *ExitInstance()* method as described in previous sub-chapter (xxx.cpp file)

8. Add all the libraries needed to compile a Tango device server (see chapter 8.6.1.2) and the Tango resource file to the linker Object/Libraries modules.

8.5.3 Win32 application

Even if it is more natural to use the C++ structure of the MFC class to write a Tango device server, it is possible to write a device server as a Win32 application. Instead of having a *main()* function as the application entry point, the operating system, provides a *WinMain()* function as the application entry point. Some code must be added to this *WinMain* function in order to support Tango device server. Don't forget to add the *tango.h* file in the list of included files. If you are using the project files generated by Pogo, don't forget to change the linker SUBSYSTEM option to "Windows" (Under Linker/System in the project properties window).

```
int APIENTRY WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow)
{
MSG msg;
Tango::Util *tg;

try
{
tg = Tango::Util::init(hInstance,nCmdShow);

string txt;
txt = "Blabla first line\n";
txt = txt + "Blabla second line\n";
txt = txt + "Blabla third line\n";
tg->set_main_window_text(txt);
tg->set_server_version("2.2");

tg->server_init(true);

tg->server_run();

}
catch (bad_alloc)
{
MessageBox((HWND)NULL,"Memory error","Command line",MB_ICONSTOP);
return (FALSE);
}
catch (Tango::DevFailed &e)
{
MessageBox((HWND)NULL,e.errors[0].desc.in(),"Command line",MB_ICONSTOP);
return (FALSE);
}
catch (CORBA::Exception &)
{
MessageBox((HWND)NULL,"Exception CORBA","Command line",MB_ICONSTOP);
return(FALSE);
}

while (GetMessage(&msg, NULL, 0, 0))
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}

delete tg;

return msg.wParam;
}
```

Line 11 : Create the Tango::Util singleton
Line 13-18 : Set parameters for the graphical interface
Line 20 : Initialize Tango device server requesting the display of the graphical interface
Line 22 : Run the device server
Line 25-39 : Display a message box for all the kinds of error during Tango device server initialization phase and exit WinMain function.
Line 41-45 : The Windows message loop
Line 47 : Delete the Tango::Util singleton. This class destructor unregisters the device server from the Tango database.

Remember that if the Tango device server graphical user interface is used, you must add the Tango windows resource file to your project.

If you don't want to use the tango device server graphical user interface, do not use any parameter in the call of the *server_init()* method and do not link your device server with the Tango Windows resource file.

8.5.4 Device server as NT service

With Windows, if you want to have processes which survive to logoff sequence and/or are automatically started during computer startup sequence, you have to write them as service. It is possible to write Tango device server as service. You need to

1. Write a class which inherits from a pre-written Tango class called NTService. This class must have a *start* method.
2. Write a main function following a predefined skeleton.

8.5.4.1 The service class

It must inherits from the *NTService* class and defines a *start* method. The NTService class must be constructed with one argument which is the device server executable name. The *start* method has three arguments which are the number of arguments passed to the method, the argument list and a reference to an object used to log info in the NT event system. The first two args must be passed to the Tango::Util::init method and the last one is used to log error or info messages. The class definition file looks like

```
1 #include <tango.h>
2 #include <ntservice.h>
3
4 class MYService: public Tango::NTService
5 {
6 public:
7 MYService(char *);
8
9 void start(int,char **,Tango::NTEventLogger *);
10 };
```

Line 1-2 : Some include files
Line 4 : The MYService class inherits from *Tango::NTService* class
Line 7 : Constructor with one parameter
Line 9 : The *start()* method

The class source code looks like

```
1 #include <myservice.h>
2 #include <tango.h>
3
4 using namespace std;
5
6 MYService::MYService(char *exec_name):NTService(exec_name)
7 {
8 }
9
10 void MYService::start(int argc,char **argv,Tango::NTEventLogger *logger)
11 {
12 Tango::Util *tg;
13 try
14 {
15 Tango::Util::_service = true;
16
17 tg = Tango::Util::init(argc,argv);
18
19 tg->server_init();
20
21 tg->server_run();
22 }
23 catch (bad_alloc)
24 {
25 logger->error("Can't allocate memory to store device object");
26 }
27 catch (Tango::DevFailed &e)
28 {
29 logger->error(e.errors[0].desc.in());
30 }
31 catch (CORBA::Exception &)
32 {
33 logger->error("CORBA Exception");
34 }
35 }
```

Line 6-8 : The MYService class constructor code.
Line 15 : Set to true the *_service* static variable of the *Tango::Util* class.
Line 17-21 : Classical Tango device server startup code
Line 23-34 : Exception management. Please, note that within a service. it is not possible to print data on a console. This method receives a reference to a logger object. This object sends all its output to the Windows event system. It is used to send messages when an exception has occurred.

8.5.4.2 The main function

The main function is used to create one instance of the class describing the service, to check the service option and to run the service. The code looks like :

```
1 #include <tango.h>
2 #include <MYService.h>
3
4 using namespace std;
5
6
7 int main(int argc,char *argv[])
8 {
9 MYService service(argv[0]);
10
11 int ret;
12 if ((ret = service.options(argc,argv)) <= 0)
13 return ret;
14
15 service.run(argc,argv);
16
17 return 0;
18 }
```

Line 9 : Create one instance of the MYService class with the executable name as parameter

Line 12 : Check service option with the *options()* method inherited from the NTService class.

Line 15 : Run the service. The *run()* method is inherited from the NTService class. This method will after some NT initialization sequence execute the user *start()* method.

8.5.4.3 Service options and messages

When a Tango device server is written as a Windows service, it supports several new options. These option are linked to Windows service usage.

Before it can be used, a service must be installed. A name and a title is associated to each service. For Tango device server used as service, the service name is build from the executable name followed by the underscore character and the instance name. For example, a device server service executable file named "opc" and started with "fluids" as instance name, will be named "opc_fluids". The title string is built from the service executable name followed by the sentence "Tango device server" and the instance name between parenthesis. In the previous example, the service title will be "opc Tango device server (fluids)". Once a service is installed, you can configure it with the "Services" application of the control panel. Services title are displayed by this application and allow the user to select one specific service. Once a service is selected, it is possible to start/stop it and to configure its startup type as manual (with the Services application) or as automatic. When the automatic mode is chosen, the service starts when the computer is started. In this case, the service executable code must resides on the computer local disk.

Tango device server logs message in the Windows event system when the service is started or stopped. You can see these messages with the "Event Viewer" application (Start->Programs->Administrative tools->Event Viewer) and choose the Application events.

The new options are -i, -s, -u, -h and -d.

- -i : Install the service
- -s : Install the service and choose the automatic startup mode
- -u : Un-install the service
- -dbg : Run in console mode to debug service. The service must have been installed prior to used it. The classical -v device server option can be used with the -d option.

On the command line, all these options must be used after the device server instance name ("opc fluids -i" to install the service, "opc fluids -u" to un-install the service, "opc fluids -v -d" to debug the service)

8.5.4.4 Tango device server using MFC as Windows service

If your Tango device server uses MFC and must be written as a Windows NT service, follow these rules :

- Don't forget to add the *stdafx.h* file as the first file included in all the source files making the project.
- Comment out the definition of VC_EXTRALEAN in the *stdafx.h* file.
- Change the pre-processor definitions, replace _WINDOWS by _CONSOLE
- Add the /SUBSYSTEM:CONSOLE option in the linker options window of the project settings.
- Add a call to initialize the MFC (*AfxWinInit()*) in the service main function

```
1 int main(int argc,char *argv[])
2 {
3 if (!AfxWinInit(::GetModuleHandle(NULL),NULL,::GetCommandLine(),0))
4 {
5 cerr << "Can't initialise MFC !" << endl;
6 return -1;
7 }
8
9 service serv(argv[0]);
10
11 int ret;
12 if ((ret = serv.options(argc,argv)) <= 0)
13 return ret;
14
15 serv.run(argc,argv);
16
17 return 0;
18 }
```

Line 3 : The MFC classes are initialized with the *AfxWinInit()* function call.

8.6 Compiling, linking and executing a TANGO device server process

8.6.1 Compiling and linking a C++ device server

8.6.1.1 On UNIX like operating system

8.6.1.1.1 Supported development tools The supported compiler for Linux is **gcc** release 3.3 and above. Please, note that to debug a Tango device server running under Linux, **gdb** release 7 and above is needed in order to correctly handle threads.

8.6.1.1.2 Compiling TANGO for C++ uses omniORB (release 4) as underlying CORBA Object Request Broker [11] and starting with Tango 8, the ZMQ library. To compile a TANGO device server, your include search path must be set to :

- The omniORB include directory
- The ZMQ include directory
- The Tango include directory
- Your development directory

8.6.1.1.3 Linking To build a running device server process, you need to link your code with several libraries. Nine of them are always the same whatever the operating system used is. These nine libraries are:

- The Tango libraries (called **libtango** and **liblog4tango**)
- Three omniORB package libraries (called **libomniORB4**, **libomniDynamic4** and **libCOS4)**
- The omniORB threading library (called **libomnithread**)
- The ZMQ library (callled **libzmq**)

On top of that, you need additional libraries depending on the operating system :

- For Linux, add the posix thread library (**libpthread**)

The following table summarizes the necessary options to compile a Tango C++ device server. Please, note that starting with Tango 8 and for gcc release 4.3 and later, some C++11 code has been used. This requires the compiler option "-std=c++0x". Obviously, the options -I and -L must be updated to reflect your file system organization.

Operating system	Compiling option	Linking option
Linux gcc	-D_REENTRANT -std=c++0x -I..	-L.. -ltango -llog4tango -lomniORB4 -lomniDynamic4 -lCOS4 -lomnithread -lzmq -lpthread

The following is an example of a Makefile for Linux. Obviously, all the paths are set to the ESRF file system structure.

```
#
#                 Makefile to generate a Tango server
#

CC = c++
BIN_DIR = ubuntu1104
TANGO_HOME = /segfs/tango

INCLUDE_DIRS = -I $(TANGO_HOME)/include/$(BIN_DIR) \
               -I .
```

```
11
12 LIB_DIRS = -L $(TANGO_HOME)/lib/$(BIN_DIR)
13
14
15 CXXFLAGS =  -D_REENTRANT -std=c++0x $(INCLUDE_DIRS)
16 LFLAGS =  $(LIB_DIRS) -ltango \
17                       -llog4tango \
18                       -lomniORB4 \
19                       -lomniDynamic4 \
20                       -lCOS4 \
21                       -lomnithread \
22                       -lzmq \
23                       -lpthread
24
25
26 SVC_OBJS =        main.o \
27                   ClassFactory.o \
28                   SteppermotorClass.o \
29                   Steppermotor.o \
30                   SteppermotorStateMachine.o
31
32
33 .SUFFIXES:        .o .cpp
34 .cpp.o:
35         $(CC) $(CXXFLAGS) -c $<
36
37
38 all: StepperMotor
39
40 StepperMotor:     $(SVC_OBJS)
41         $(CC) $(SVC_OBJS) -o $(BIN_DIR)/StepperMotor $(LFLAGS)
42
43 clean:
44         rm -f *.o core
```

Line 5-7 : Define Makefile macros
Line 9-10 : Set the include file search path
Line 12 : Set the linker library search path
Line 15 : The compiler option setting
Line 16-23 : The linker option setting
Line 26-30 : All the object files needed to build the executable
Line 33-35 : Define rules to generate object files
Line 38 : Define a "all" dependency
Line 40-41 : How to generate the StepperMotor device server executable
Line 43-44 : Define a "clean" dependency

8.6.1.2 On Windows using Visual Studio

Supported Windows compiler for Tango is Visual C++ 2008 (VC 9) and above. Most problems in building a Windows device server revolve around the /M compiler switch family. This switch family controls which run-time library names are embedded in the object files, and consequently which libraries are used during linking. Attempt to mix and match compiler settings and libraries can cause link error and even if successful, may produce undefined run-time behavior.

Selecting the correct /M switch in Visual Studio is done through a dialog box. To open this dialog box, click on the “Project” menu (once the correct project is selected in the Solution Explorer window) and select the “Properties” option. To change the compiler switch open the “C/C++” tree and select “Code Generation”. The following options are supported.

- Multithreaded = /MT
- Multithreaded DLL = /MD
- Debug Multithreaded = /MTd
- Debug Multithreaded DLL = /MDd

Compiling a file with a value of the /M switch family will impose at link phase the use of libraries also compiled with the same value of the /M switch family. If you compiled your source code with the /MT option (Multithreaded), you must link it with libraries also compiled with the /MT option.

On both 32 or 64 bits computer, omniORB and TANGO relies on the preprocessor identifier **WIN32** being defined in order to configure itself. If you build an application using static libraries (option /MT or /MTd), you must add **_WINSTATIC** to the list of the preprocessor identifiers. If you build an application using DLL (option /MD or /MDd), you must add **LOG4TANGO_HAS_DLL** and **TANGO_HAS_DLL** to the list of preprocessor identifiers.

To build a running device server process, you need to link your code with several libraries on top of the Windows libraries. These libraries are:

- The Tango libraries (called **tango.lib** and **log4tango.lib** or **tangod.lib** and **log4tangod.lib** for debug mode)
- The omniORB package libraries (see next table)

Compile mode	Libraries
Debug Multithreaded	omniORB4d.lib, omniDynamic4d.lib, omnithreadd.lib and COS4d.lib
Multithreaded	omniORB4.lib, omniDynamic4.lib, omnithread.lib and COS4.lib
Debug Multithreaded DLL	omniORB416_rtd.lib, omniDynamic416_rtd.lib, omnithread34_rtd.lib, and COS416_rtd.lib
Multithreaded DLL	omniORB416_rt.lib, omniDynamic416_rt.lib, omnithread34_rt.lib and COS416_rt.lib

- The ZMQ library (**zmq.lib** or **zmqd.lib** for debug mode)
- Windows network libraries (**mswsock.lib** and **ws2_32.lib**)
- Windows graphic library (**comctl32.lib**)

To add these libraries in Visual Studio, open the project property pages dialog box and open the “Link” tree. Select “Input” and add these library names to the list of library in the “Additional Dependencies” box.

The “Win32 Debug” or “Win32 Release” configuration that you change within the "Configuration Manager" window changes the /M switch compiler. For instance, if you select a “Win32 Debug” configuration in a "non-DLL" project, use the omniORB4d.lib, omniDynamic4d.lib and omnithreadd.lib libraries plus the tangod.lib, log4tangod.lib and zmqd.lib libraries. If you select the “Win32 Release” configuration, use the omniORB4.lib, omniDynamic4.lib and omnithread.lib libraries plus the tango.lib, log4tango.lib and zmq.lib libraries.

WARNING: In some cases, the Microsoft Visual Studio wizard used during project creation generates one include file called *Stdafx.h*. If this file itself includes windows.h file, you have to add the preprocessor macro _WIN32_WINNT and set it to 0x0500.

8.6.2 Running a C++ device server

To run a C++ Tango device server, you must set an environment variable. This environment variable is called **TANGO_HOST** and has a fixed syntax which is

TANGO_HOST=<host>:<port>

The host field is the host name where the TANGO database device server is running. The port field is the port number on which this server is listening. For instance, a valid syntax is TANGO_HOST=dumela:10000. For UNIX like operating system, setting environment variable is possible with the *export* or *setenv* command depending on the shell used. For Windows, setting environment variable is possible with the "Environment" tab of the "System" application in the control panel.

If you need to start a Tango device server on a pre-defined port (For Tango database device server or device server without database usage), you must use one of the underlying ORB option *endPoint* like

myserver myinstance_name -ORBendPoint giop:tcp::<port number>

8.6.3 Compiling a Java device server

8.6.3.1 Supported java release

Tango device server written using Java language needs release **1.5.0** (or above) of the Java environment.

8.6.3.2 Setting the CLASSPATH

To correctly compile a Java Tango device server, the CLASSPATH environment variable must be set to :

- The Tango jar file. All Tango and TangoDs package classes have been stored in this jar file. On top of that, this file also includes all the CORBA ORB classes (JacORB classes). This file is named TangORB.jar
- The jar file with all the JDK classes (not always necessary, could be implicit)
- Your own directory

For UNIX like operating system, setting environment variable is done with the *export* or *setenv* command depending on the shell used. For Windows, setting environment variable is possible with the "Environment" tab of the "System" application in the control panel.

8.6.3.3 Makefile

The following is an example of a Makefile for a Java Tango device server. Obviously, all the paths are set to the ESRF file system structure.

```
#
# Makefile to generate a TANGO java device server
#

JAVAC = javac -classpath $(CLASSPATH):..

# ------------------------------------
#
```

```
9 # The compiler flags
10 #
11 #----------------------------------------------------------------
12
13 JAVAFLAGS = -g
14
15 #----------------------------------------------------------------
16
17
18 CL_LIST = DevReadPositionCmd.class \
19 StepperMotor.class \
20 StepperMotorClass.class
21
22 PACKAGE = server
23
24 #
25 # Rule for compiling
26 #
27
28 .SUFFIXES: .class .java
29 .java.class:
30 $(JAVAC) $(JAVAFLAGS) $<
31
32 #-----------------------------------------------------------------
33
34
35 all: $(PACKAGE)
36
37 $(PACKAGE): $(CL_LIST)
38
39 clean:
40 rm -f *.class
```

Line 5 : Definition of the java compiler
Line 13 : The java compiler flag
Line 18 : List of class to be compiled
Line 28 : Define a dependency name
Line 29-30 : Define how source files must be compiled
Line 35 : The "all" dependency
Line 47 : The device server dependency
Line 39-40 : The "clean" dependency

8.6.3.4 Tango core software release number

All the Tango core classes are packaged in the Tango.jar file. A little utility tool called **TangoVers** allows a user to know which release of the Tango core classes he(she) is using. This utility is available only with Java 1.2 virtual machine. To run this utility, simply type

TangoVers <path to Tango.jar file>

if the directory /segfs/tango/bin is in your PATH environment variable.

8.6.4 Running a Java device server

A correct setting of the CLASSPATH environment variable is not enough to run a Java Tango device server. You must also set a Java system property. The name of the system property is **TANGO_HOST** and its syntax is the same than the syntax described in chapter 8.6.2. Setting a Java system property is done by using -D option of the java interpreter command. To run a Java Tango device server, the command line must start with

java -DTANGO_HOST=<host>:<port> xxxx

As all the device server files are part of a package, you have to run this command in the directory above the package directory. For instance, for our StepperMotor device server started with *et* as instance name, all files must be stored in a directory called StepperMotor and the command line must be

java -DTANGO_HOST=<host>:<port> StepperMotor/StepperMotor et

run from the directory above the StepperMotor one.

If you need to start a Tango device server on a pre-defined port (For Tango database device server or device server without database usage), you must use one of the underlying ORB option OAPort like

java -DOAPort=<port number> myserver myinstance_name

8.7 Advanced programming techniques

The basic techniques for implementing device server pattern are required by each device server programmer. In certain situations, it is however necessary to do things out of the ordinary. This chapter will look into programming techniques which permit the device server serve more than simply the network.

8.7.1 Receiving signal (C++ specific)

It is **UNSAFE** to use any CORBA call in a signal handler. It is also UNSAFE to use some system calls in a signal handler. Tango device server solved this problem by using threads. A specific thread is started to handle signals. Therefore, every Tango device server is automatically a threaded process. This allows the programmer to write the code which must be executed when a signal is received as ordinary code. All device server threads masks all signals except the specific signal thread which is permanently waiting for signal. If a signal is sent to a device server process, only the signal thread will receive it because it is the single thread which does not mask signals.

Nevertheless, signal management is not trivial and some care have to be taken. The signal management differs from operating system to operating system. It is not recommended that you install your own signal routine using any of the signal routines provided by the operating system calls or library.

8.7.1.1 Using signal

It is possible for C++ device server to receive signals from drivers or other processes. The TDSOM supports receiving signal at two levels: the device level and the class level. Supporting signal at the device level means that it is possible to specify interest into receiving signal on a device basis. This feature is supported via three methods defined in the DeviceImpl class. These methods are called *register_signal*, *unregister_signal* and s*ignal_handler*.

The ***register_signal*** method has one parameter which is the signal number. This method informs the device server signal system that the device want to be informed when the signal passed as parameter is received by the process. There is a special case for Linux as explained in the previous sub-chapter. It is possible to register a signal to be executed in the a signal handler context (with all its restrictions). This is done with a second parameter to this *register_signal* method. This second parameter is simply a boolean data. If it is true, the signal_handler will be executed in a signal handler context in the device server main thread. A default value (false) has been defined for this parameter.

The ***unregister_signal*** method also have an input parameter which is the signal number. This method removes the device from the list of object which should be warned when the signal is received by the process.

The ***signal_handler*** method is the method which is triggered when a signal is received if the corresponding *register_signal* has been executed. This method is defined as virtual and can be redefined by the user. It has one input argument which is the signal number.

The same three methods also exist in the DeviceClass class. Their action and their usage are similar to the DeviceImpl class methods. Installing a signal at the class level does not mean that all the device belonging to this class will receive the signal. This only means that the *signal_handler* method of the DeviceClass instance will be executed. This is useful if an action has to be executed once for a class of devices when a signal is received.

The following code is an example with our stepper motor device server configured via the database to serve three motors. These motors have the following names : id04/motor/01, id04/motor/02 and id04/motor/03. The signal SIGALRM (alarm signal) must be propagated only to the motor number 2 (id04/motor/02)

```
 1  void StepperMotor::init_device()
 2  {
 3      cout << "StepperMotor::StepperMotor() create motor " << dev_name << endl;
 4
 5      long i;
 6
 7      for (i=0; i< AGSM_MAX_MOTORS; i++)
 8      {
 9          axis[i] = 0;
10          position[i] = 0;
11          direction[i] = 0;
12      }
13
14      if (dev_name == "id04/motor/02")
15          register_signal(SIGALRM);
16  }
17
18  StepperMotor::~StepperMotor()
19  {
20      unregister_signal(SIGALRM);
21  }
22
23  void StepperMotor::signal_handler(long signo)
24  {
25      INFO_STREAM << "Inside signal handler for signal " << signo << endl;
26
27  //  Do what you want here
28
29  }
```

The *init_device* method is modified.

Line 14-15 : The device name is checked and if it is the correct name, the device is registered in the list of device wanted to receive the SIGALARM signal.

The destructor is also modified

Line 20 : Unregister the device from the list of devices which should receives the SIGALRM signal. Note that unregister a signal for a device which has not previously registered its interest for this signal does nothing.

The *signal_handler* method is redefined

Line 25 : Print signal number

Line 27 : Do what you have to do when the signal SIGALRM is received.

If all devices must be warned when the device server process receives the signal SIGALRM, removes line 14 in the *init_device* method.

8.7.1.2 Exiting a device server gracefully

A device server has to exit gracefully by unregistering itself from the database. The necessary action to gracefully exit are automatically executed on reception of the following signal :

- SIGINT, SIGTERM and SIGQUIT for device server running on Linux
- SIGINT, SIGTERM, SIGABRT and SIGBREAK for device server running on Windows

This does not prevents device server to also register interest at device or class levels for those signals. The user installed *signal_handler* method will first be called before the graceful exit.

8.7.2 Inheriting

This sub-chapter details how it is possible to inherit from an existing device pattern implementation. As the device pattern includes more than a single class, inheriting from an existing device pattern needs some explanations.

Let us suppose that the existing device pattern implementation is for devices of class A. This means that classes A and AClass already exists plus classes for all commands offered by device of class A. One new device pattern implementation for device of class B must be written with all the features offered by class A plus some new one. This is easily done with the inheritance. Writing a device pattern implementation for device of class B which inherits from device of class A means :

- Write the BClass class
- Write the B class
- Write B class specific commands
- Eventually redefine A class commands

8.7.2.1 Using C++

The miscellaneous code fragments given below detail only what has to be updated to support device pattern inheritance

8.7.2.1.1 Writing the BClass As you can guess, BClass has to inherit from AClass. The *command_factory* method must also be adapted.

```
namespace B
{

class BClass : public A::AClass
{
.....
}

BClass::command_factory()
{
```

```
11 A::AClass::command_factory();
12
13 command_list.push_back(....);
14 }
15
16 } /* End of B namespace */
```

Line 1 : Open the B namespace
Line 4 : BClass inherits from AClass which is defined in the A namespace.
Line 11 : Only the *command_factory* method of the BClass will be called at start-up. To create the AClass commands, the *command_factory* method of the AClass must also be executed. This is the reason of the line
Line 13 : Create BClass commands

8.7.2.1.2 Writing the B class As you can guess, B has to inherits from A.

```
1 namespace B
2 {
3
4 class B : public A:A
5 {
6 .....
7 };
8
9 B::B(Tango::DeviceClass *cl,const char *s):A::A(cl,s)
10 {
11 ....
12 init_device();
13 }
14
15 void B::init_device()
16 {
17 ....
18 }
19
20 } /* End of B namespace */
```

Line 1 : Open the B namespace.
Line 4 : B inherits from A which is defined in the A namespace
Line 9 : The B constructor calls the right A constructor

8.7.2.1.3 Writing B class specific command Noting special here. Write these classes as usual

8.7.2.1.4 Redefining A class command It is possible to redefine a command which already exist in class A **only if the command is created using the inheritance model** (but keeping its input and output argument types). The method which really execute the class A command is a method implemented in the A class. This method must be defined as **virtual.** In class B, you can redefine the method executing the command and implement it following the needs of the B class.

8.7.2.2 Using Java

The miscellaneous code fragments given below detail only what has to be updated to support device pattern inheritance

8.7.2.2.1 Writing the BClass As you can guess, BClass has to inherit from AClass. Some change must be done in the definition of the *init* and *instance* methods. The *command_factory* method must also be adapted.

```
1 public class BClass extends AClass implements TangoConst
2 {
3 public static AClass init(String name) throws DevFailed
4 {
5
6 }
7
8 public static AClass instance()
9 {
10
11 }
12
13 public void command_factory()
14 {
15 super.command_factory();
16
17 command_list.addElement(....);
18 }
19 };
```

Line 1 : BClass inherits from AClass and implements TangoConst interface

Line 3 : The return data type of the *init* method must be the same as the type defines in the AClass (therefore a reference to AClass) otherwise, the compiler complains. BClass inherits from AClass and a reference to a BClass is also a reference to the AClass

Line 8 : The return data type of the *instance* method must also be adapted as explained for the *init* method

Line 15 : Only the *command_factory* method of the BClass will be called at start-up. To create the AClass commands, the *command_factory* method of the AClass must also be executed. This is the reason of the line

Line 17 : Create BClass commands

8.7.2.2.2 Writing the B class As you can guess, B has to inherits from A. The *init_device* method must be adapted, the constructor has to be modified and an instance variable must be added

```
public class B extends A implements TangoConst
{
boolean constructed = false;

A(DeviceClass cl,String s)
{
```

```
7 super(cl,s);
8 constructed = true;
9 ...
10 init_device();
11 }
12
13 public void init_device()
14 {
15 if (constructed == false)
16 {
17 return;
18 }
19 super.init_device();
20
21 ...
22 }
23 };
```

Line 1 : B inherits from A and implements TangoConst interface
Line 3 : A boolean initialized to false is added as instance variable
Line 8 : The constructor is modified to set the constructed boolean to true after all the super classes have been created and before the call to the *init_device* method.
Line 15-18 : The *init_device* method immediately returns if the constructed boolean is false (if the super classes are not correctly created)
Line 19 : The *init_device* method of class A is called

8.7.2.2.3 Writing B class specific command Noting special here. Write these classes as usual

8.7.2.2.4 Redefining A class command It is possible to to redefine a command which already exist in class A **only if the command is created using the inheritance model** (but keeping its input and output argument types). The method which really execute the class A command is a method implemented in the A class. With Java, it is possible to redefine all methods except those which are declared as "final". Therefore, in class B, you can redefine the method executing the command and implement it following the needs of the B class. The following is an example for a command xxx which is programmed to call a *my_cmd* method[10].

```
public class A extends DeviceImpl implements TangoConst
{
public void my_cmd(long input)
{
}
}

public class B extends A implements TangoConst
{
public void my_cmd(long input)
{
}
}
```

[10] In the command *execute* method

Line 3 : The *my_cmd* method is defined in class A

Line 10 : The *my_cmd* method is redefined in class B

Inside the device pattern, the device object is created as an instance of class B[11]. Java will call the *my_cmd* method of the B class when the command is received. It is still possible to call the *my_cmd* method of the A class with the help of the Java "super" keyword inside the code of the *my_cmd* method of the B class.

8.7.3 Using another device pattern implementation within the same server

It is often necessary that inside the same device server, a method executing a command needs a command of another class to be executed. For instance, a device pattern implementation for a device driven by a serial line class can use the command offered by a serial line class embedded within the same device server process. To execute one of the command (or any other CORBA operations/attributes) of the serial line class, just call it as a normal client will do by using one instance of the Deviceproxy class. The ORB will recognize that all the devices are inside the same process and will execute calls as a local calls. To create the DeviceProxy class instance, the only thing you need to know is the name of the device you gave to the serial line device. Retrieving this could be easily done by a Tango device property. The DeviceProxy class is fully described in chapters related to the Java or C++ Tango Application Programming Interface (API)

[11] By the *device_factory* method of the BClass class

Chapter 9

Advanced features

9.1 Attribute alarms

Each Tango attribute two several alarms. These alarms are :

- A four thresholds level alarm
- The read different than set (RDS) alarm

9.1.1 The level alarms

This alarm is defined for all Tango attribute read type and for numerical data type. The action of this alarm depend on the attribute value when it is read :

- If the attribute value is below or equal the attribute configuration **min_alarm** parameter, the attribute quality factor is switched to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM.
- If the attribute value is below or equal the attribute configuration **min_warning** parameter, the attribute quality factor is switched to Tango::ATTR_WARNING and if the device state is Tango::ON, it is switched to Tango::ALARM.
- If the attribute value is above or equal the attribute configuration **max_warning** parameter, the attribute quality factor is switched to Tango::ATTR_WARNING and if the device state is Tango::ON, it is switched to Tango::ALARM.
- If the attribute value is above or equal the attribute configuration **max_alarm** parameter, the attribute quality factor is switched to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM.

If the attribute is a spectrum or an image, then the alarm is set if any one of the attribute value satisfies the above criterium. By default, these four parameters are not defined and no check will be done.

The following figure is a drawing of attribute quality factor and device state values function of the the attribute value.

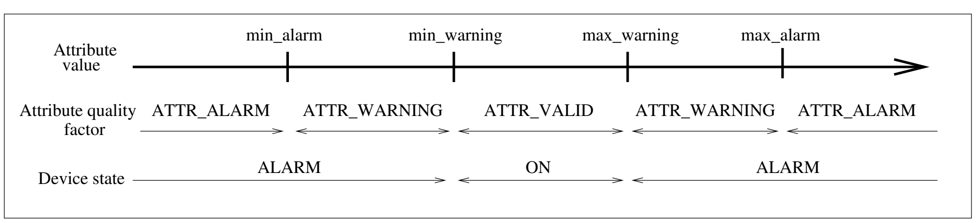


Figure 9.1: Level alarm

If the min_warning and max_warning parameters are not set, the attribute quality factor will simply change between Tango::ATTR_ALARM and Tango::ATTR_VALID function of the attribute value.

9.1.2 The Read Different than Set (RDS) alarm

This alarm is defined only for attribute of the Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. When the attribute is read (or when the device state is requested), if the difference between its read value and the last written value is something more than or equal to an authorized delta and if at least a certain amount of milli seconds occurs since the last write operation, the attribute quality factor will be set to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM. If the attribute is a spectrum or an image, then the alarm is set if any one of the attribute value's satisfies the above criterium. This alarm configuration is done with two attribute configuration parameters called **delta_val** and **delta_t**. By default, these two parameters are not defined and no check will be done.

9.2 Device polling

9.2.1 Introduction

Each tango device server automatically have a separate polling thread pool. Polling a device means periodically executing command on a device (or reading device attribute) and storing the results (or the thrown exception) in a polling buffer. The aim of this polling is threefold :

- Speed-up response time for slow device
- Get a first-level history of device command output or attribute value
- Be the data source for the Tango event system

Speeding-up response time is achieved because the command_inout CORBA operation is able to get its data from the polling buffer or from the a real access to the device. For "slow" device, getting the data from the buffer is much faster than accessing the device. Returning a first-level command output history (or attribute value history) to a client is possible due to the polling buffer which is managed as a circular buffer. The history is the contents of this circular buffer. Obviously, the history depth is limited to the depth of the circular buffer. The polling is also the data source for the event system because detecting an event means being able to regularly read the data, memorize it and declaring that it is an event after some comparison with older values.

9.2.2 Configuring the polling system

9.2.2.1 Configuring what has to be polled and how

It is possible to configure the polling in order to poll :

- Any command which does not need input parameter
- Any attribute

Configuring the polling is done by sending command to the device server administration device automatically implemented in every device server process. Seven commands are dedicated to this feature. These commands are

AddObjPolling It add a new object (command or attribute) to the list of object(s) to be polled. It is also with this command that the polling period is specified.

RemObjPolling To remove one object (command or attribute) from the polled object(s) list

UpdObjPollingPeriod Change one object polling period

StartPolling Starts polling for the whole process

StopPolling Stops polling for the whole process

PolledDevice Allow a client to know which device are polled

DevPollStatus Allow a client to precisely knows the polling status for a device

All the necessary parameters for the polling configuration are stored in the Tango database. Therefore, the polling configuration is not lost after a device server process stop and restart (or after a device server process crash!!).

It is also possible to automatically poll a command (or an attribute) without sending command to the device server administration device. This request some coding (a method call) in the device server software during the command or attribute creation. In this case, for every devices supporting this command or this attribute, polling configuration will be automatically updated in the database and the polling will start automatically at each device server process startup. It is possible to stop this behavior on a device basis by sending a RemObjPolling command to the device server administration device. The following piece of code shows how the source code should be written.

```
 1
 2  void DevTestClass::command_factory()
 3  {
 4  ...
 5      command_list.push_back(new IOStartPoll("IOStartPoll",
 6                                              Tango::DEV_VOID,
 7                                              Tango::DEV_LONG,
 8                                              "Void",
 9                                              "Constant number"));
10      command_list.back()->set_polling_period(400);
11  ...
12  }
13
14
15  void DevTestClass::attribute_factory(vector<Tango::Attr *> &att_list)
16  {
17  ...
18      att_list.push_back(new Tango::Attr("String_attr",
19                                          Tango::DEV_STRING,
20                                          Tango::READ));
21      att_list.back()->set_polling_period(250);
22  ...
23  }
```

A polling period of 400 mS is set for the command called “IOStartPoll” at line 10 with the *set_polling_period* method of the Command class. Therefore, for a device of this class, the polling thread will start polling its IOStartPoll command at process start-up except if a RemObjPolling indicating this device and the IOStartPoll command has already been received by the device server administration device. This is exactly the same behavior for attribute. The polling period for attribute called “String_attr” is defined at line 20.

Configuring the polling means defining device attribute/command polling period. The polling period has to be chosen with care. If reading an attribute needs 200 mS, there is no point to poll this attribute with a polling period equal or even below 200 mS. You should also take into account that some "free" time has to be foreseen for external request(s) on the device. On average, for one attribute needing X mS as reading time, define a polling period which is equal to 1.4 X (280 mS for our example of one attribute needing 200 mS as reading time). In case the polling tuning is given to external user, Tango provides a way to define polling period minimun threshold. This is done using device properties. These properties are named *min_poll_period*, *cmd_min_poll_period* and *attr_min_poll_period*. The property min_poll_period (mS) defined a minimun polling period for the device. The property cmd_min_poll_period allows the definition of a minimun polling period for a specific device command. The property attr_min_poll_period allows the definition of a minimun polling period for one device attribute. In case these properties are defined, it is not possible to poll the device command/attribute with a polling period below those defined by these properties. See Appendix A on device parameter to get a precise syntax description for these properties.

The Jive[21] tool also allows a graphical device polling configuration.

9.2.2.2 Configuring the polling threads pool

Starting with Tango release 7, a Tango device server process may have several polling threads managed as a pool. For instance, this could be usefull in case of devices within the same device server process but accessed by different hardware channel when one of the channel is not responding (Thus generating long timeout and de-synchronising the polling thread). By default, the polling threads pool size is set to 1 and all the polled object(s) are managed by the same thread (idem polling system in Tango releases older than release 7) . The configuration of the polling thread pool is done using two properties associated to the device server administration device. These properties are named:

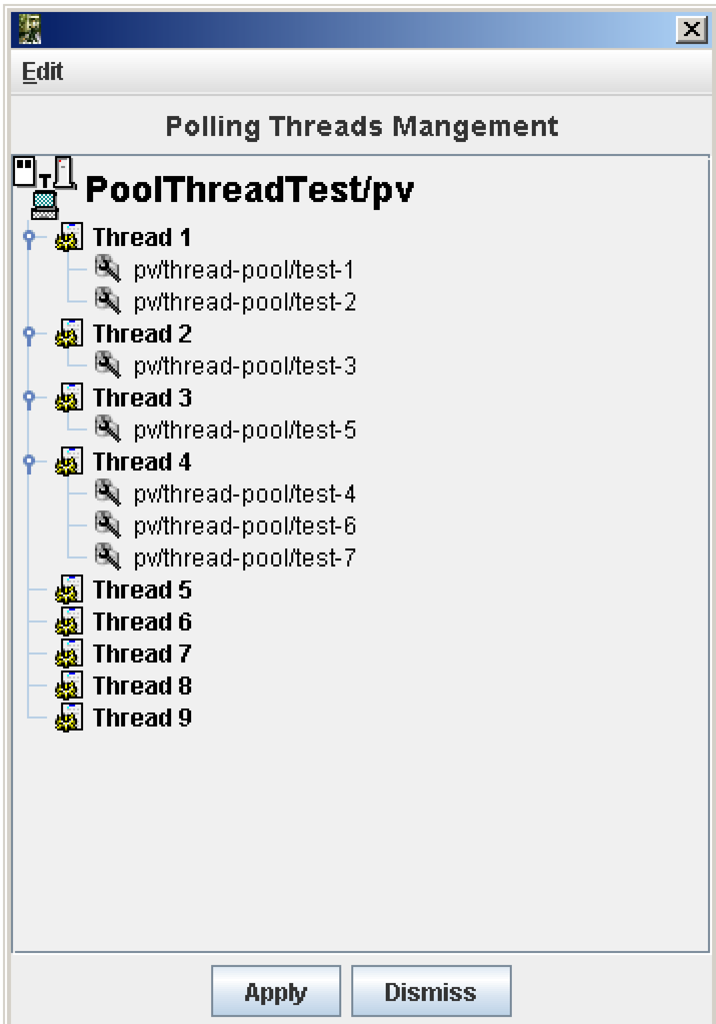
- *polling_threads_pool_size* defining the maximun number of threads that you can have in the pool
- *polling_threads_pool_conf* defining which threads in the pool manages which device

The granularity of the polling threads pool tuning is the device. You cannot ask the polling threads pool to have thread number 1 in charge of attribute *att1* of device *dev1* and thread number 2 to be in charge of *att2* of the same device *dev1*.

When you require a new object (command or attribute) to be polled, two main cases may arrive:

1. Some polled object(s) belonging to the device are already polled by one of the polling threads in the pool: There is no new thread created. The object is simply added to the list of objects to be polled for the existing thread
2. There is no thread already created for the device. We have two sub-cases:
   (a) The number of polling threads is less than the polling_threads_pool_size: A new thread is created and started to poll the object (command or attribute)
   (b) The number of polling threads is already equal to the polling_threads_pool_size: The software search for the thread with the smallest number of polled objects and add the new polled object to this thread

Each time the polling threads pool configuration is changed, it is written in the database using the polling_threads_pool_conf property. If the behaviour previously described does not fulfill your needs, it is possible to update the polling_threads_pool_conf property in a graphical way using the Tango Astor [19] tool or manually using the Jive tool [21]. These changes will be taken into account at the next device server process start-up. At start-up, the polling threads pool will allways be configured as required by the polling_threads_pool_conf property. The syntax used for this property is described in the Reference part of the Appendix A. The following window dump is the Astor tool window which allows polling threads pool management.



In this example, the polling threads pool size to set to 9 but only 4 polling threads are running. Thread 1 is in charge of all polled objects related to device pv/thread-pool/test-1 and pv/thread-pool/test-2. Thread 2 is in charge of all polled objects related to device pv/thread-pool/test-3. Thread 3 is in charge of all polled objects related to device pv/thread-pool/test-5 anf finally, thread 4 is in charge of all polled objects for devices pv/thread-pool/test-4, pv/thread-pool/test-6 and pv/thread-pool/test-7.

It's also possible to define the polling threads pool size programmatically in the main function of a device server process using the *Util::set_polling_threads_pool_size()* method before the call to the *Util::server_init()* method

9.2.3 Reading data from the polling buffer

For a polled command or a polled attribute, a client has three possibilities to get command result or attribute value (or the thrown exception) :

- From the device itself
- From the polling buffer
- From the polling buffer first and from the device if data in the polling buffer are invalid or if the polling is badly configured.

The choice is done during the command_inout CORBA operation by positioning one of the operation parameter. When reading data from the polling buffer, several error cases are possible

- The data in the buffer are not valid any more. Every time data are requested from the polling buffer, a check is done between the client request date and the date when the data were stored in the buffer. An exception is thrown if the delta is greater than the polling period multiplied by a “too old” factor. This factor has a default value and is up-datable via a device property. This is detailed in the reference part of this manual.
- The polling is correctly configured but there is no data yet in the polling buffer.

9.2.4 Retrieving command/attribute result history

The polling thread stores the command result or attribute value in circular buffers. It is possible to retrieve an history of the command result (or attribute value) from these polling buffers. Obviously the history is limited by the depth of the circular buffer. For commands, a CORBA operation called *command_inout_history_2* allows this retrieval. The client specifies the command name and the record number he want to retrieve. For each record, the call returns the date when the command was executed, the command result or the exception stack in case of the command failed when it was executed by the polling thread. In such a case, the exception stack is sent as a structure member and not as an exception. The same thing is available for attribute. The CORBA operation name is *read_attribute_history_2*. For these two calls, there is no check done between the call date and the record date in contrary of the call to retrieve the last command result (or attribute value).

9.2.5 Externally triggered polling (only for C++ device server)

Sometimes, rather than polling a command or an attribute regulary with a fixed period, it is more interesting to "manually" decides when the polling must occurs. The Tango polling system also supports this kind of usage. This is called *externally triggered polling*. To define one attribute (or command) as externally triggered, simply set its polling period to 0. This can be done with the device server administration device AddObjPolling or UpdObjPollingPeriod command. Once in this mode, the attribute (or command) polling is triggered with the *trigger_cmd_polling()* method (or *trigger_attr_polling()* method) of the Util class. The following piece of code shows how this method could be used for one externally triggered command.

```
1  .....
2
3  string ext_polled_cmd("MyCmd");
4  Tango::DeviceImpl *device = .....;
5
6  Tango::Util *tg = Tango::Util::instance();
7
8  tg->trigger_cmd_polling(device,ext_polled_cmd);
9
10 .....
```

line 3 : The externally polled command name
line 4 : The device object
line 8 : Trigger polling of command MyCmd

9.2.6 Filling polling buffer (only for C++ device server)

Some hardware to be interfaced already returned an array of pair value, timestamp. In order to be read with the *command_inout_history* or *read_attribute_history* calls, this array has to be transferred in the attribute or command polling buffer. This is possible only for attribute or command configured in the externally triggered polling mode. Once in externally triggered polling mode, the attribute (or command) polling buffer is filled with the *fill_cmd_polling_buffer()* method (or *fill_attr_polling_buffer()* method) of the Util class. For command, the user uses a template class called *TimedCmdData* for each element of the command history. Each element is stored in a stack in one instance of a template class called *CmdHistoryStack*. This object is one of the argument of the fill_cmd_polling_buffer() method. Obviously, the stack depth cannot be larger than the polling buffer depth. See A.1.4 to learn how the polling buffer depth is defined. The same way is used for attribute with the *TimedAttrData* and *AttrHistoryStack* template classes. These classes are documented in [8]. The following piece of code fills the polling buffer for a command called MyCmd which is already in externally triggered mode. It returns a DevVarLongArray data type with three elements. This example is not really something you will find in a real hardware interface. It is only to demonstrate the fill_cmd_polling_buffer() method usage. Error management has also been removed.

```
1  ....
2
3  Tango::DevVarLongArray dvla_array[4];
4
5  for(int i = 0;i < 4;i++)
6  {
7      dvla_array[i].length(3);
8      dvla_array[i][0] = 10 + i;
9      dvla_array[i][1] = 11 + i;
10     dvla_array[i][2] = 12 + i;
11 }
12
13 Tango::CmdHistoryStack<DevVarLongArray> chs;
14 chs.length(4);
15
16 for (int k = 0;k < 4;k++)
17 {
18     time_t when = time(NULL);
19
20     Tango::TimedCmdData<DevVarLongArray> tcd(&dvla_array[k],when);
21     chs.push(tcd);
22 }
23
24 Tango::Util *tg = Tango::Util::instance();
25 string cmd_name("MyCmd");
26 DeviceImpl *dev = ....;
27
28 tg->fill_cmd_polling_buffer(dev,cmd_name,chs);
29
30 .....
```

Line 3-11 : Simulate data coming from hardware

Line 13-14 : Create one instance of the CmdHistoryStack class and reserve space for one history of 4 elements

Line 16-17 : A loop on each history element

Line 18 : Get date (hardware simulation)

Line 20 : Create one instance of the TimedCmdData class with data and date

Line 21 : Store this command history element in the history stack. The element order will be the insertion order whatever the element date is.

Line 28 : Fill command polling buffer

After one execution of this code, a command_inout_history() call will return one history with 4 elements. The first array element of the oldest history record will have the value 10. The first array element of the newest history record will have the value 13. A command_inout() call with the data source parameter set to CACHE will return the newest history record (ie an array with values 13,14 and 15). A command_inout() call with the data source parameter set to DEVICE will return what is coded is the command method. If you execute this code a second time, a command_inout_history() call will return an history of 8 elements.

The next example fills the polling buffer for an attribute called MyAttr which is already in externally triggered mode. It is a scalar attribute of the DevString data type. This example is not really something you will find in a real hardware interface. It is only to demonstrate the fill_attr_polling_buffer() method usage with memory management issue. Error management has also been removed.

```
 1  ....
 2
 3  AttrHistoryStack<DevString> ahs;
 4  ahs.length(3);
 5
 6  for (int k = 0;k < 3;k++)
 7  {
 8      time_t when = time(NULL);
 9
10      DevString *ptr = new DevString [1];
11      ptr = CORBA::string_dup("Attr history data");
12
13      TimedAttrData<DevString> tad(ptr,Tango::ATTR_VALID,true,when);
14      ahs.push(tad);
15  }
16
17  Tango::Util *tg = Tango::Util::instance();
18  string attr_name("MyAttr");
19  DeviceImpl *dev = ....;
20
21  tg->fill_attr_polling_buffer(dev,attr_name,ahs);
22
23  .....
```

Line 3-4 : Create one instance of the AttrHistoryStack class and reserve space for an history with 3 elements

Line 6-7 : A loop on each history element

Line 8 : Get date (hardware simulation)

Line 10-11 : Create a string. Note that the DevString object is created on the heap

Line 13 : Create one instance of the TimedAttrData class with data and date requesting the memory to be released.

Line 14 : Store this attribute history element in the history stack. The element order will be the insertion order whatever the element date is.

Line 21 : Fill command polling buffer

It is not necessary to return the memory allocated at line 10. The *fill_attr_polling_buffer()* method will do it for you.

9.2.7 Setting and tuning the polling in a Tango class

Even if the polling is normally set and tuned with external tool like Jive, it is possible to set it directly into the code of a Tango class. A set of methods belonging to the *DeviceImpl* class allows the user to deal with polling. These methods are:

- *is_attribute_polled()* and *is_command_polled()* to check if one command/attribute is polled
- *get_attribute_poll_period()* and *get_command_poll_period()* to get polled object polling period
- *poll_attribute()* and *poll_command()* to poll command or attribute
- *stop_poll_attribute()* and *stop_poll_command()* to stop polling a command or an attribute

The following code snippet is just an exmaple of how these methods could be used. They are documented in [24]

```
void MyClass::read_attr(Tango::Attribute &attr)
{
    ...
    ...

    string att_name("SomeAttribute");
    string another_att_name("AnotherAttribute");

    if (is_attribute_polled(att_name) == true)
        stop_poll_attribute(att_name);
    else
        poll_attribute(another_att_name,500);

    ....
    ....

}
```

9.3 Threading

When used with C++, Tango used omniORB as underlying ORB. This CORBA implementation is a threaded implementation and therefore a C++ Tango device server or client are multi-threaded processes.

9.3.1 C++ device server process

A classical Tango device server without any connected clients has eight threads. These threads are :

- The main thread waiting in the ORB main loop
- Two ORB implementation threads (the POA thread)
- The ORB scavanger thread
- The signal thread

- The heartbeat thread (needed by the Tango event system)
- Two Zmq implementation threads

On top of these eight threads, you have to add the thread(s) used by the polling threads pool. This number depends on the polling thread pool configuration and could be between 0 (no polling at all) and the maximun number of threads in the pool.

A new thread is started for each connected client. Device server are mostly used to interface hardware which most of the time does not support multi-threaded access. Therefore, all remote calls executed from a client are serialized within the device server code by using mutual exclusion. See chapter 9.3.1.1 on which serialization model are available. In order to limit thread number, the underlying ORB (omniORB) is configured to shutdown threads dedicated to client if the connection is inactive for more than 3 minutes. To also limit thread number, the ORB is configured to create one thread per connection up to 55 threads. When this level is reached, the threading model is automatically switch to a "thread pool" model with up to 100 threads. If the number of threads decrease down to 50, the threading model will return to "thread per connection" model.

If you are using event, the event system for its internal heartbeat system periodically (every 200 seconds) sends a command to the device server administration device. As explained above, a thread is created to execute these command. The omniORB scavanger will terminate this thread before the next event system heartbeat command arrives. For example, if you have a device server with three connected clients using only event, the process thread number will permanently change between 8 and 11 threads.

In summary, the number of threads in a device server process can be evaluated with the following formula:

$$\mathbf{8 + k + m}$$

k is the number of polling threads used from the polling threads pool and m is the number of threads used for connected clients.

9.3.1.1 Serialization model within a device server

Four serialization models are available within a device server. These models protect all requests coming from the network but also requests coming from the polling thread. These models are:

1. Serialization by device. All access to the same device are serialized. As an example, let's take a device server implementing one class of device with two instances (dev1 and dev2). Two clients are connected to these devices (client1 and client2). Client2 will not be able to access dev1 if client1 is using it. Nevertheless, client2 is able to access dev2 while client1 access dev1 (There is one mutual exclusion object by device)

2. Serialization by class. With non multi-threaded legacy software, the preceding scenario could generate problem. In this mode of serialization, client2 is not able to access dev2 while client1 access dev1 because dev2 and dev1 are instances of the same class (There is one mutual exclusion object by class)

3. Serialization by process. This is one step further than the previous case. In this mode, only one client can access any device embedded within the device server at a time. There is only one mutual exclusion object for the whole process)

4. No serialization. This is an exotic kind of serialization and **should be used with extreme care** only with device which are fully thread safe. In this model, most of the device access are not serialized at all. Due to Tango internal structure, the *get_attribute_config*, *set_attribute_config*, *read_attributes* and *write_attributes* CORBA calls are still protected. Reading the device state and status via commands or via CORBA attribute is also protected.

By default, every Tango device server is in serialization by device mode. A method of the Tango::Util class allows to change this default behavior.

```
1  #include <tango.h>
2
3  int main(int argc,char *argv[])
4  {
5
6      try
7      {
8
9          Tango::Util *tg = Tango::Util::init(argc,argv);
10
11         tg->set_serial_model(Tango::BY_CLASS);
12
13         tg->server_init();
14
15         cout << "Ready to accept request" << endl;
16         tg->server_run();
17     }
18     catch (bad_alloc)
19     {
20          cout << "Can't allocate memory!!!" << endl;
21          cout << "Exiting" << endl;
22     }
23     catch (CORBA::Exception &e)
24     {
25          Tango::Except::print_exception(e);
26
27          cout << "Received a CORBA::Exception" << endl;
28          cout << "Exiting" << endl;
29     }
30
31     return(0);
32 }
```

The serialization model is set at line 11 before the server is initialized and the infinite loop is started. See [8] for all details on the methods to set/get serialization model.

9.3.1.2 Attribute Serialization model

Even with the serialization model described previously, in case of attributes carrying a large number of data and several clients reading this attribute, a device attribute serialization has to be followed. Without this level of serialization, for attribute using a shared buffer, a thread scheduling may happens while the device server process is in the CORBA layer transferring the attribute data on the network. Three serialization models are available for attribute serialization. The default is well adapted to nearly all cases. Nevertheless, if the user code manages several attributes data buffer or if it manages its own buffer protection by one way or another, it could be interesting to tune this serialization level. The available models are:

1. Serialization by kernel. This is the default case. The kernel is managing the serialization
2. Serialization by user. The user code is in charge of the serialization. This serialization is done by the use of a omni_mutex object. An omni_mutex is an object provided by the omniORB package. It is the user responsability to lock this mutex when appropriate and to give this mutex to the Tango kernel before leaving the attribute read method

3. No serialization.

By default, every Tango device attribute is in serialization by kernel. Methods of the Tango::Attribute class allow to change the attribute serialization behavior and to give the user omni_mutex object to the kernel.

```
1 void MyClass::init_device()
2 {
3    ...
4    ...
5    Tango::Attribute &att = dev_attr->get_attr_by_name("TheAttribute");
6    att.set_attr_serial_model(Tango::ATTR_BY_USER);
7    ....
8    ....
9
10 }
11
12
13 void MyClass::read_TheAttribute(Tango::Attribute &attr)
14 {
15    ....
16    ....
17    the_mutex.lock();
18    ....
19    // Fill the attribute buffer
20    ....
21    attr.set_value(buffer,....);
22    attr->set_user_attr_mutex(&the_mutex);
23 }
24
```

The serialization model is set at line 6 in the init_device() method. The user omni_mutex is passed to the Tango kernel at line 22. This omni_mutex object is a device data member. See [8] for all details on the methods to set attribute serialization model.

9.3.2 C++ client process

Clients are also multi threaded processes. The underlying C++ ORB (omniORB) try to keep system resources to a minimum. To decrease process file descriptors usage, each connection to server is automatically closed if it is idle for more than 2 minutes and automatically re-opened when needed. A dedicated thread is spawned by the ORB to manage this automatic closing connection (the ORB scavenger thread).

Threrefore, a Tango client has two threads which are:

1. The main thread
2. The ORB scavanger thread

If the client is using the event system and as Tango is using the event push-push model, it has to be a server for receiving the events. This increases the number of threads. The client now has 6 threads which are:

- The main thread
- The ORB scavenger thread
- Two Zmq implementation threads
- Two Tango event system related threads (the KeepAliveThread and the EventConsumer thread)

9.4 Generating events in a device server

The server is at the origin of events. It will fire events as soon as they occur. Standard events (*change*, *periodic* and *archive*) are detected automatically in the polling thread and fired as soon as they are detected. The *periodic* events can only be handled by the polling thread. *Change, Data ready* and *archive* events can also be pushed from the device server code. To allow a client to subscribe to events of non polled attributes the server has to declare that events are pushed from the code. Three methods are available for this purpose:

```
Attr::set_change_event(bool implemented, bool detect = true);
Attr::set_archive_event(bool implemented, bool detect = true);
Attr::set_data_ready_event( bool implemented);
```

where *implemented*=true indicates that events are pushed manually from the code and *detect*=true (when used) triggers the verification of the same event properties as for events send by the polling thread. When setting *detect*=false, no value checking is done on the pushed value! The class DeviceImpl also supports the first two methods with an addictional parameter attr_name defining the attribute name.

To push events manually from the code a set of data type dependent methods can be used:

```
DeviceImpl::push_change_event (string attr_name, ....);
DeviceImpl::push_archive_event(string attr_name, ....);
```

For the data ready event, a DeviceImpl class method has to be used to push the event.

```
DeviceImpl::push_data_ready_event(string attr_name,Tango::DevLong ctr);
```

See the class documentation for all available interfaces.

For non-standard events a single call exists for pushing the data to the CORBA Notification Service (omniNotify). Clients who are subscribed to this event have to know what data type is in the DeviceAttribute and unpack it accordingly.

To push non-standard events, use the following api call is available to all device servers :

```
DeviceImpl::push_event( string attr_name,
             vector<string> &filterable_names,
             vector<double> &filterable_vals,
             Attribute &att)
```

where *attr_name* is the name of the attribute. *Filterable_names* and *filterable_vals* represent any filterable data which can be used by clients to filter on. Here is a typical example of what a server will need to do to send its own events. We are in the read method of the "Sinusoide" attribute. This attribute is readable as any other attribute but an event is sent if its value is positive when it is read. On top of that, this event is sent with one filterable field called "value" which is set to the attribute value.

```
void MyClass::read_Sinusoide(Tango::Attribute &attr)
{
  ...
     struct timeval tv;
     gettimeofday(&tv, NULL);
     sinusoide = 100 * sin( 2 * 3.14 * frequency * tv.tv_sec);

     if (sinusoide >= 0)
     {
         vector<string> filterable_names;
         vector<double> filterable_value;
```

```
12
13          filterable_names.push_back("value");
14          filterable_value.push_back((double)sinusoide);
15
16          push_event( attr.get_name(),
17                      filterable_names, filterable_value,
18                      &sinusoide);
19      }
20    ....
21    ....
22
23 }
```

line 13-14 : The filter pair name/value is initialised
line 16-18 : The event is pushed

9.5 Memorized attribute

It is possible to ask Tango to store in its database the last written value for attribute of the SCALAR data format and obviously only for READ_WRITE or READ_WITH_WRITE attribute. This is fully automatic. During device startup phase, for all device memorized attributes, the value written in the database is fetched and applied. A write_attribute call can be generated to apply the memorized value to the attribute or only the attribute set point can be initialised. The following piece of code shows how the source code should be written to set an attribute as memorized and to initialise only the attribute set point.

```
1 void DevTestClass::attribute_factory(vector<Tango::Attr *> &att_list)
2  {
3      ...
4      att_list.push_back(new String_attrAttr());
5      att_list.back()->set_memorized();
6      att_list.back()->set_memorized_init(false);
7      ...
8  }
```

Line 4 : The attribute to be memorized is created and inserted in the attribute vector.
Line 5 : The *set_memorized()* method of the attribute base class is called to define the attribute as memorized.
Line 6 : The set_memorized_init() method is called with the parameter false to define that only the set point should be initialsied.

9.6 Transfering images

Some optimized methods have been written to optimize image transfer between client and server using the attribute DevEncoded data type. All these methods have been merged in a class called EncodedAttribute. Within this class, you will find methods to:

- Encode an image in a compressed way (JPEG) for images coded on 8 (gray scale), 24 or 32 bits

- Encode a grey scale image coded on 8 or 16 bits
- Encode a color image coded on 24 bits
- Decode images coded on 8 or 16 bits (gray scale) and returned a 8 or 16 bits grey scale image
- Decode color images transmitted using a compressed format (JPEG) and returns a 32 bits RGB image

The following code snippets are examples of how these methods have to be used in a server and in a client. On the server side, creates an instance of the EncodedAttribute class within your object

```
1 class MyDevice::Tango::Device_4Impl
2  {
3       ...
4       Tango::EncodedAttribute jpeg;
5       ...
6  }
```

In the code of your device, use an encoding method of the EncodedAttribute class

```
1 void MyDevice::read_Encoded_attr_image(Tango::Attribute &att)
2 {
3       ....
4       jpeg.encode_jpeg_gray8(imageData,256,256,50.0);
5       att.set_value(&jpeg);
6 }
```

Line 4: Image encoding. The size of the image is 256 by 256. Each pixel is coded using 8 bits. The encoding quality is defined to 50 in a scale of 0 - 100. imageData is the pointer to the image data (pointer to unsigned char)

Line 5: Set the value of the attribute using a *Attribute::set_value()* method.

On the client side, the code is the following (without exception management)

```
1    ....
2    DeviceAttribute da;
3    EncodedAttribute att;
4    int width,height;
5    unsigned char *gray8;
6
7    da = device.read_attribute("Encoded_attr_image");
8    att.decode_gray8(&da,&width,&height,&gray8);
9    ....
10   delete [] gray8;
11   ...
```

The attribute named Encoded_attr_image is read at line7. The image is decoded at line 8 in a 8 bits gray scale format. The image data are stored in the buffer pointed to by "gray8". The memory allocated by the image decoding at line 8 is returned to the system at line 10.

9.7 Device server with user defined event loop

Sometimes, it could be usefull to write your own process event handling loop. For instance, this feature can be used in a device server process where the ORB is only one of several components that must perform event handling. A device server with a graphical user interface must allow the GUI to handle windowing events in addition to allowing the ORB to handle incoming requests. These types of device server therefore perform non-blocking event handling. They turn the main thread of control over each of the vvarious event-handling sub-systems while not allowing any of them to block for significants period of time. The *Tango::Util* class has a method called *server_set_event_loop()* to deal with such a case. This method has only one argument which is a function pointer. This function does not receive any argument and returns a boolean. If this boolean is true, the device server process exits. The device server core will call this function in a loop without any sleeping time between the call. It is the user responsability to implement in this function some kind of sleeping mechanism in order not to make this loop too CPU consuming. The code of this function is executed by the device server main thread. The following piece of code is an example of how you can use this feature.

```

bool my_event_loop()
{
   bool ret;

   some_sleeping_time();

   ret = handle_gui_events();

   return ret;
}

int main(int argc,char *argv[])
{
   Tango::Util *tg;
   try
   {
      // Initialise the device server
      //----------------------------------------
      tg = Tango::Util::init(argc,argv);

      tg->set_polling_threads_pool_size(5);

      // Create the device server singleton
      //        which will create everything
      //----------------------------------------
      tg->server_init(false);

      tg->server_set_event_loop(my_event_loop);

      // Run the endless loop
      //----------------------------------------
      cout << "Ready to accept request" << endl;
      tg->server_run();
   }
   catch (bad_alloc)
```

```
37     {
38     ...
```

The device server main event loop is set at line 29 before the call to the Util::server_run() method. The function used as server loop is defined between lines 2 and 11.

9.8 Device server using file as database

For device servers not able to access the Tango database (most of the time due to network route or security reason), it is possible to start them using file instead of a real database. This is done via the device server

-file=<file name>

command line option. In this case,

- Getting, setting and deleting class properties
- Getting, setting and deleting device properties
- Getting, setting and deleting class attribute properties
- Getting, setting and deleting device attribute properties

are handled using the specified file instead of the Tango database. The file is an ASCII file and follows a well-defined syntax with predefined keywords. The simplest way to generate the file for a specific device server is to use the Jive application. See [21] to get Jive documentation. The Tango database is not only used to store device configuration parameters, it is also used to store device network access parameter (the CORBA IOR). To allow an application to connect to a device hosted by a device server using file instead of database, you need to start it on a pre-defined port, and you must use one of the underlying ORB option called *endPoint* like

myserver myinstance_name -file=/tmp/MyServerFile -ORBendPoint giop:tcp::<port number>

to start your device server. The device name passed to the client application must also be modified in order to refect the non-database usage. See C.1 to learn about Tango device name syntax. Nevertheless, using this Tango feature prevents some other features to be used :

- No check that the same device server is running twice.
- No device or attribute alias name.
- In case of several device servers running on the same host, the user must manually manage a list of already used network port.

9.9 Device server without database

In some very specific cases (Running a device server within a lab during hardware development...), it could be very useful to have a device server able to run even if there is no database in the control system. Obviously, running a Tango device server without a database means loosing Tango features. The lost features are :

- No check that the same device server is running twice.
- No device configuration via properties.
- No event generated by the server.

- No memorized attributes
- No device attribute configuration via the database.
- No check that the same device name is used twice within the same control system.
- In case of several device servers running on the same host, the user must manually manage a list of already used network port.

To run a device server without a database, the **-nodb** command line option must be used. One problem when running a device server without the database is to pass device name(s) to the device server. Within Tango, it is possible to define these device names at two different levels :

1. At the command line with the **-dlist** option: In case of device server with several device pattern implementation, the device name list given at command line is only for the last device pattern created in the *class_factory()* method. In the device name list, the device name separator is the comma character.
2. At the device pattern implementation level: In the class inherited from the Tango::DeviceClass class via the re-definition of a well defined method called *device_name_factory()*

If none of these two possibilities is used, the tango core classes defined one default device name for each device pattern implementation. This default device name is *NoName*. Device definition at the command line has the highest priority.

9.9.1 Example of device server started without database usage

Without database, you need to start a Tango device server on a pre-defined port, and you must use one of the underlying ORB option called *endPoint* like

myserver myinstance_name -ORBendPoint giop:tcp::<port number> -nodb -dlist a/b/c

The following is two examples of starting a device server not using the database when the *device_name_factory()* method is not re-defined.

- StepperMotor et -nodb -dlist id11/motor/1,id11/motor/2
  This command line starts the device server with two devices named *id11/motor/1* and *id11/motor/2*
- StepperMotor et -nodb
  This command line starts a device server with one device named *NoName*

When the *device_name_factory()* method is re-defined within the StepperMotorClass class.

```
void StepperMotorClass::device_name_factory(vector<string> &list)
{
    list.push_back("sr/cav-tuner/1");
    list.push_back("sr/cav-tuner/2");
}
```

- StepperMotor et -nodb
  This commands starts a device server with two devices named *sr/cav-tuner/1* and *sr/cav-tuner/2*.
- StepperMotor et -nodb -dlist id12/motor/1
  Starts a device server with only one device named id12/motor/1

9.9.1.1 Java device server without the database

It is also possible to start a Java device server without the database using exactly the principle described in the above lines. Nevertheless, a java device server process retrieves its list of device pattern implementation from the database! Therefore, a *add_class()* method is defined in the java Util class and the main method must be updated.

```
package StepperMotor

import java.util.*;
import org.omg.CORBA.*;
import fr.esrf.Tango.*;
import fr.esrf.TangoDs.*;

public class StepperMotor extends DeviceImpl implements TangoConst
{
    public static void main(String[] argv)
    {
        try
        {

            Util tg = Util.init(argv,"StepperMotor");

            tg.add_class("StepperMotor");
            tg.server_init();

            System.out.println("Ready to accept request");

            tg.server_run();
        }
        catch (OutOfMemoryError ex)
        {
            System.err.println("Can't allocate memory !!!!");
            System.err.println("Exiting");
        }
        catch (UserException ex)
        {
            Except.print_exception(ex);

            System.err.println("Received a CORBA user exception");
            System.err.println("Exiting");
        }
        catch (SystemException ex)
        {
            Except.print_exception(ex);

            System.err.println("Received a CORBA system exception");
            System.err.println("Exiting");
        }

        System.exit(-1);

    }
```

```
47 }
```

The add_class() method is used at line 17 before the device pattern(s) implementation initialization.

9.9.1.2 Start a java device server without database

Without database, you need to start a Tango device server on a pre-defined port, and you must use one of the underlying ORB option OAPort like

java -DOAPort=<port number> myserver myinstance_name -nodb -dlist id11/motor/1,id11/motor/2

9.9.2 Connecting client to device within a device server started without database

In this case, the host and port on which the device server is running are part of the device name. If the device name is *a/b/c*, the host is *mycomputer* and the port *1234*, the device name to be used by client is

mycomputer:1234/a/b/c#dbase=no

See appendix C.1 for all details about Tango object naming.

9.10 Multiple database servers within a Tango control system

Tango uses MySQL as database and allows access to this database via a specific Tango device server. It is possible for the same Tango control system to have several Tango database servers. The host name and port number of the database server is known via the TANGO_HOST environment variable. If you want to start several database servers in order to prevent server crash, use the following TANGO_HOST syntax

TANGO_HOST=<host_1>:<port_1>,<host_2>:<port_2>,<host_3>:<port_3>

All calls to the database server will automatically switch to a running servers in the given list if the one used dies.

9.11 The Tango controlled access system

9.11.1 User rights definition

Within the Tango controlled system, you give rights to a user. User is the name of the user used to log-in the computer where the application trying to access a device is running. Two kind of users are defined:

1. Users with defined rights
2. Users without any rights defined in the controlled system. These users will have the rights associated with the pseudo-user called "All Users"

The controlled system manages two kind of rights:

- Write access meaning that all type of requests are allowed on the device
- Read access meaning that only read-like access are allowed (write_attribute, write_read_attribute and set_attribute_config network calls are forbidden). Executing a command is also forbidden except for commands defined as "**Allowed commands**". Getting a device state or status using the command_inout call is always allowed. The definition of the allowed commands is done at the device class level. Therefore, all devices belonging to the same class will have the allowed commands set.

The rights given to a user is the check result splitted in two levels:

1. At the host level: You define from which hosts the user may have write access to the control system by specifying the host name. If the request comes from a host which is not defined, the right will be Read access. If nothing is defined at this level for the user, the rights of the "All Users" user will be used. It is also possible to specify the host by its IP address. You can define a host family using wide-card in the IP address (eg. 160.103.11.* meaning any host with IP address starting with 160.103.11). Only IP V4 is supported.

2. At the device level: You define on which device(s) request are allowed using device name. Device family can be used using widecard in device name like domin/family/*

Therefore, the controlled system is doing the following checks when a client try to access a device:

- Get the user name
- Get the host IP address
- If rights defined at host level for this specific user and this IP address, gives user temporary write acccess to the control system
- If nothing is specified for this specific user on this host, gives to the user a temporary access right equal to the host access rights of the "All User" user.
- If the temporary right given to the user is write access to the control system
  - If something defined at device level for this specific user
    - If there is a right defined for the device to be accessed (or for the device family), give user the defined right
    - Else
      - If rights defined for the "All Users" user for this device, give this right to the user
      - Else, give user the Read Access for this device
  - Else
    - If there is a right defined for the device to be accessed (or for the device family) for the "All User" user, give user this right
    - Else, give user the Read Access right for this device
- Else, access right will be Read Access

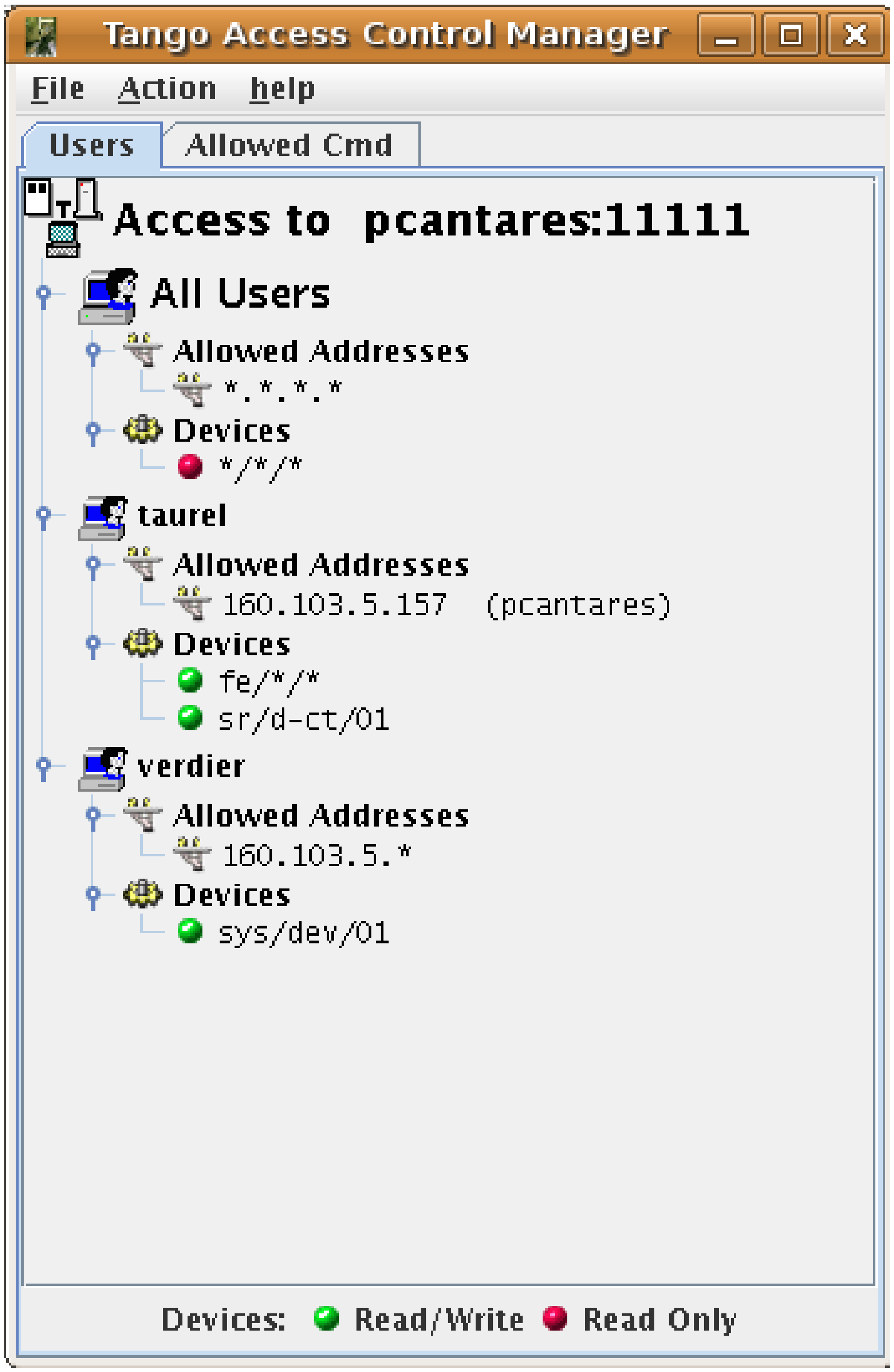
Then, when the client tries to access the device, the following algorithm is used:

- If right is Read Access
  - If the call is a write type call, refuse the call
  - If the call is a command execution
    - If the command is one of the command defined in the "Allowed commands" for the device class, send the call
    - Else, refuse the call

All these checks are done during the DeviceProxy instance constructor except those related to the device class allowed commands which are checked during the command_inout call.

To simplify the rights management, give the "All Users" user host access right to all hosts ("*.*.*.*") and read access to all devices ("*/*/*"). With such a set-up for this user, each new user without any rights defined in the controlled access will have only Read Access to all devices on the control system but from any hosts. Then, on request, gives Write Access to specific user on specific host (or family) and on specific device (or family).

The rights managements are done using the Tango Astor[19] tool which has some graphical windows allowing to grant/revoke user rights and to define device class allowed commands set. The following window dump shows this Astor window.



In this example, the user "taurel" has Write Access to the device "sr/d-ct/1" and to all devices belonging to the domain "fe" but only from the host "pcantares" He has read access to all other devices but always only from the host pcantares. The user "verdier" has write access to the device "sys/dev/01" from any host on the network "160.103.5" and Read Access to all the remaining devices from the same network. All the other users has only Read Access but from any host.

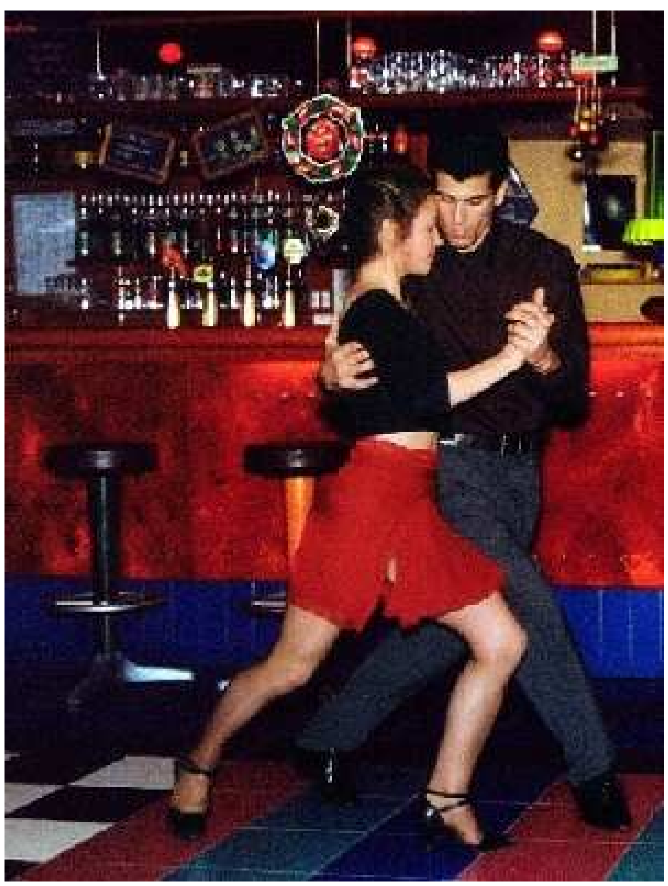
9.11.2 Running a Tango control system with the controlled access

All the users rights are stored in two tables of the Tango database. A dedicated device server called **TangoAccessControl** access these tables without using the classical Tango database server. This TangoAccessControl device server must be configured with only one device. The property **Services** belonging to the free object **CtrlSystem** is used to run a Tango control system with its controlled access. This property is an array of string with each string describing the service(s) running in the control system. For controlled access, the service name is "AccessControl". The service instance name has to be defined as "tango". The device name associated with this service must be the name of the TangoAccessControl server device. For instance, if the TangoAccessControl device server device is named *sys/access_control/1*, one element of the Services property of the CtrlSystem object has to be set to

AccessControl/tango:sys/access_control/1

If the service is defined but without a valid device name corresponding to the TangoAccessControl device server, all users from any host will have write access (simulating a Tango control system without controlled access). Note that this device server connects to the MySQL database and therefore may need the MySQL connection related environment variables MYSQL_USER and MYSQL_PASSWORD described in A.12.3.3

Even if a controlled access system is running, it is possible to by-pass it if, in the environment of the client application, the environment variable SUPER_TANGO is defined to "true". If for one reason or another, the controlled access server is defined but not accessible, the device right checked at that time will be Read Access.



Appendix A

Reference part

This chapter is only part of the TANGO device server reference guide. To get reference documentation about the C++ library classes, see [8]. To get reference documentation about the Java classes, also see [8].

A.1 Device parameter

A black box, a device description field, a device state and status are associated with each TANGO device.

A.1.1 The device black box

The device black box is managed as a circular buffer. It is possible to tune the buffer depth via a device property. This property name is

device name->blackbox_depth

A default value is hard-coded to 50 if the property is not defined. This black box depth property is retrieved from the Tango property database during the device creation phase.

A.1.2 The device description field

There are two ways to intialise the device description field.

- At device creation time. Some constructors of the DeviceImpl class supports this field as parameter. If these constructor are not used, the device description field is set to a default value which is *A Tango device*.
- With a property. A description field defines with this method overrides a device description defined at construction time. The property name is

  device name->description

A.1.3 The device state and status

Some constructors of the DeviceImpl class allows the initialisation of device state and/or status or device creation time. If these fields are not defined, a default value is applied. The default state is Tango::UNKOWN, the default status is *Not Initialised.*

A.1.4 The device polling

Seven device properties allow the polling tunning. These properties are described in the following table

Property name	property rule	default value
poll_ring_depth	Polling buffer depth	10
cmd_poll_ring_depth	Cmd polling buffer depth	
attr_poll_ring_depth	Attr polling buffer depth	
poll_old_factor	"Data too old" factor	4
min_poll_period	Minimun polling period	
cmd_min_poll_period	Min. polling period for cmd	
attr_min_poll_period	Min. polling period for attr	

The rule of the poll_ring_depth property is obvious. It defines the polling ring depth for all the device polled command(s) and attribute(s). Nevertheless, when filling the polling buffer via the fill_cmd_polling_buffer() (or fill_attr_polling_buffer()) method, it could be helpfull to define specific polling ring depth for a command (or an attribute). This is the rule of the cmd_poll_ring_depth and attr_poll_ring_depth properties. For each polled object with specific polling depth (command or attribute), the syntax of this property is the object name followed by the ring depth (ie State,20,Status,15). If one of these properties is defined, for the specific command or attribute, it will overwrite the value set by the poll_ring_depth property. The poll_old_factor property allows the user to tune how long the data recorded in the polling buffer are valid. Each time some data are read from the polling buffer, a check is done between the date when the data were recorded in the polling buffer and the date when the user request these data. If the interval is greater than the object polling period multiply by the value of the poll_old_factor factory, an exception is returned to the caller. These two properties are defined at device level and therefore, it is not possible to tune this parameter for each polled object (command or attribute). The last 3 properties are dedicated to define a polling period minimum threshold. The property min_poll_period defines in (mS) a device minimum polling period. Property cmd_min_poll_period defines (in mS) a minimum polling period for a specific command. The syntax of this property is the command name followed by the minimum polling period (ie MyCmd,400). Property attr_min_poll_period defines (in mS) a minimum polling period for a specific attribute. The syntax of this property is the attribute name followed by the minimum polling period (ie MyAttr,600). These two properties has a higher priority than the min_poll_period property. By default these three properties are not defined mening that there is no minimun polling period.

Four other properties are used by the Tango core classes to manage the polling thread. These properties are :

- polled_cmd to memorize the name of the device polled command
- polled_attr to memorize the name of the device polled attribute
- non_auto_polled_cmd to memorize the name of the command which shoule not be polled automatically at the first request
- non_auto_polled_attr to memorize the name of the attribute which should not be polled automatically at the first request

You don't have to change these properties values by yourself. They are automatically created/modified/deleted by Tango core classes.

A.1.5 The device logging

The Tango Logging Service (TLS) uses device properties to control device logging at startup (static configuration). These properties are described in the following table

Property name	property rule	default value

logging_level	Initial device logging level	WARN
logging_target	Initial device logging target	No default
logging_rft	Logging rolling file threshold	2 Mega bytes
logging_path	Logging file path	/tmp/tango-<logging name> or C:/tango-<logging name> (Windows)

- The logging_level property controls the initial logging level of a device. Its set of possible values is: "OFF", "FATAL", "ERROR", "WARN", "INFO" or "DEBUG". This property is overwritten by the verbose command line option (-v).
- The logging_target property is a multi-valued property containing the initial target list. Each entry must have the following format: target_type::target_name (where target_type is one of the supported target types and target_name, the name of the target). Supported target types are: *console*, *file* and *device*. For a device target, target_name must contain the name of a log consumer device (as defined in A.8). For a file target, target_name is the name of the file to log to. If omitted the device's name is used to build the file name (domain_family_member.log). Finally, target_name is ignored in the case of a console target. The TLS does not report any error occurred while trying to setup the initial targets.
  - Logging_target property example :

    logging_target = ["console", "file", "file::/home/me/mydevice.log", "device::tmp/log/1"]

    In this case, the device will automatically logs to the standard output, to its default file (which is something like domain_family_member.log), to a file named mydevice.log and located in /home/me. Finally, the device logs are also sent to a log consumer device named tmp/log/1.
- The logging_rft property specifies the rolling file threshold (rft), of the device's file targets. This threshold is expressed in Kb in the range [500, 20480]. When the size of a log file reaches the so-called rolling-file-threshold (rft), it is backuped as "*current_log_file_name*" + "*_1*" and a new current_log_file_name is opened. Obviously, there is only one backup file at a time (i.e. any existing backup is destroyed before the current log file is backuped). The default threshold is 2Mb, the minimum is 500 Kb and the maximum is 20 Mb.
- The logging_path property overwrites the TANGO_LOG_PATH environment variable. This property can only be applied to a DServer class device and has no effect on other devices.

A.2 Device attribute

Attribute are configured with two kind of parameters: Parameters hard-coded in source code and modifiable parameters

A.2.1 Hard-coded device attribute parameters

Seven attribute parameters are defined at attribute creation time in the device server source code. Obviously, these parameters are not modifiable except with a new source code compilation. These parameters are

Parameter name	Parameter description

name	Attribute name
data_type	Attribute data type
data_format	Attribute data format
writable	Attribute read/write type
max_dim_x	Maximum X dimension
max_dim_y	Maximum Y dimension
writable_attr_name	Associated write attribute
level	Attribute display level

A.2.1.1 The Attribute data type

Thirteen data types are supported. These data types are

- Tango::DevBoolean
- Tango::DevShort
- Tango::DevLong
- Tango::DevLong64
- Tango::DevFloat
- Tango::DevDouble
- Tango::DevUChar
- Tango::DevUShort
- Tango::DevULong
- Tango::DevULong64
- Tango::DevString
- Tango::DevState
- Tango::DevEncoded

A.2.1.2 The attribute data format

Three data format are supported for attribute

Format	Description
Tango::SCALAR	The attribute value is a single number
Tango::SPECTRUM	The attribute value is a one dimension number
Tango::IMAGE	The attribute value is a two dimension number

A.2.1.3 The max_dim_x and max_dim_y parameters

These two parameters defined the maximum size for attributes of the SPECTRUM and IMAGE data format.

data format	max_dim_x	max_dim_y
Tango::SCALAR	1	0
Tango::SPECTRUM	User Defined	0
Tango::IMAGE	User Defined	User Defined

For attribute of the Tango::IMAGE data format, all the data are also returned in a one dimension array. The first array is value[0],[0], array element X is value[0],[X-1], array element X+1 is value[1][0] and so forth.

A.2.1.4 The attribute read/write type

Tango supports four kind of read/write attribute which are :

- Tango::READ for read only attribute
- Tango::WRITE for writable attribute
- Tango::READ_WRITE for attribute which can be read and write
- Tango::READ_WITH_WRITE for a readable attribute associated to a writable attribute (For a power supply device, the current really generated is not the wanted current. To handle this, two attributes are defined which are *generated_current* and *wanted_current*. The *wanted_current* is a Tango::WRITE attribute. When the *generated_current* attribute is read, it is very convenient to also get the *wanted_current* attribute. This is exactly what the Tango::READ_WITH_WRITE attribute is doing)

When read, attribute values are always returned within an array even for scalar attribute. The length of this array and the meaning of its elements is detailed in the following table for scalar attribute.

Name	Array length	Array[0]	Array[1]
Tango::READ	1	Read value	
Tango::WRITE	1	Last write value	
Tango::READ_WRITE	2	Read value	Last write value
Tango::READ_WITH_WRITE	2	Read value	Associated attributelast write value

When a spectrum or image attribute is read, it is possible to code the device class in order to send only some part of the attribute data (For instance only a Region Of Interest for an image) but never more than what is defined by the attribute configuration parameters max_dim_x and max_dim_y. The number of data sent is also transferred with the data and is named **dim_x** and **dim_y**. When a spectrum or image attribute is written, it is also possible to send only some of the attribute data but always less than max_dim_x for spectrum and max_dim_x * max_dim_y for image. The following table describe how data are returned for spectrum attribute. dim_x is the data size sent by the server when the attribute is read and dim_x_w is the data size used during the last attribute write call.

Name	Array length	Array[0->dim_x-1]	Array[dim_x -> dim_x + dim_x_w -1]
Tango::READ	dim_x	Read values	
Tango::WRITE	dim_x_w	Last write values	
Tango::READ_WRITE	dim_x + dim_x_w	Read value	Last write values
Tango::READ_WITH_WRITE	dim_x + dim_x_w	Read value	Associated attributelast write values

The following table describe how data are returned for image attribute. dim_r is the data size sent by the server when the attribute is read (dim_x * dim_y) and dim_w is the data size used during the last attribute write call (dim_x_w * dim_y_w).

Name	Array length	Array[0->dim_r-1]	Array[dim_r-> dim_r + dim_w -1]
Tango::READ	dim_r	Read values	
Tango::WRITE	dim_w	Last write values	
Tango::READ_WRITE	dim_r + dim_w	Read value	Last write values
Tango::READ_WITH_WRITE	dim_r + dim_w	Read value	Associated attributelast write values

Until a write operation has been performed, the last write value is initialized to *0* for scalar attribute of the numeriacal type, to *"Not Initialised"* for scalar string attribute and to *true* for scalar boolean attribute. For spectrum or image attribute, the last write value is initialized to an array of one element set to *0* for numerical type, to an array of one element set to *true* for boolean attribute and to an array of one element set to "*Not initialized*" for string attribute

A.2.1.5 The associated write attribute parameter

This parameter has a meaning only for attribute with a Tango::READ_WITH_WRITE read/write type. This is the name of the associated write attribute.

A.2.1.6 The attribute display level parameter

This parameter is only an help for graphical application. It is a C++ enumeration starting at 0 or a final class for Java. The code associated with each attribute display level is defined in the following table (Tango::DispLevel).

name	Value
Tango::OPERATOR	0
Tango::EXPERT	1

This parameter allows a graphical application to support two types of operation :

- An operator mode for day to day operation
- An expert mode when tuning is necessary

According to this parameter, a graphical application knows if the attribute is for the operator mode or for the expert mode.

A.2.2 Modifiable attribute parameters

Each attribute has a configuration set of 20 modifiable parameters. These can be grouped in three different purposes:

1. General purpose parameters
2. Alarm related parameters
3. Event related parameters

A.2.2.1 General purpose parameters

Eight attribute parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
description	Attribute description
label	Attribute label
unit	Attribute unit
standard_unit	Conversion factor to MKSA unit
display_unit	The attribute unit in a printable form
format	How to print attribute value
min_value	Attribute min value
max_value	Attribute max value

The **description** parameter describes the attribute. The **label** parameter is used by graphical application to display a label when this attribute is used in a graphical application. The **unit** parameter is the attribute value unit. The **standard_unit** parameter is the conversion factor to get attribute value in MKSA units. Even if this parameter is a number, it is returned as a string by the device *get_attribute_config* call. The **display_unit** parameter is the string used by graphical application to display attribute unit to application user.

A.2.2.1.1 The format attribute parameter This parameter specifies how the attribute value should be printed. It is not valid for string attribute. This format is a string of C++ streams manipulators separated by the **;** character. The supported manipulators are :

- fixed
- scientific
- uppercase
- showpoint
- showpos
- setprecision()
- setw()

Their definition are the same than for C++ streams. An example of format parameter is

scientific;uppercase;setprecision(3)

. A class called Tango::AttrManip has been written to handle this format string. Once the attribute format string has been retrieved from the device, its value can be printed with

cout << Tango::AttrManip(format) << value << endl;

.

A.2.2.1.2 The min_value and max_value parameters These two parameters have a meaning only for attribute of the Tango::WRITE read/write type and for numerical data types. Trying to set the value of an attribute to something less than or equal to the min_value parameter is an error. Trying to set the value of the attribute to something more or equal to the max_value parameter is also an error. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

These two parameters have no meaning for attribute with data type DevString, DevBoolean or DevState. An exception is thrown in case the user try to set them for attribute of these 3 data types.

A.2.2.2 The alarm related configuration parameters

Six alarm related attribute parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
min_alarm	Attribute low level alarm
max_alarm	Attribute high level alarm
min_warning	Attribute low level warning
max_warning	Attribute high level warning
delta_t	delta time for RDS alarm (mS)
delta_val	delta value for RDS alarm (absolute)

These parameters have no meaning for attribute with data type DevString, DevBoolean or DevState. An exception is thrown in case the user try to set them for attribute of these 3 data types.

A.2.2.2.1 The min_alarm and max_alarm parameters These two parameters have a meaning only for attribute of the Tango::READ, Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. When the attribute is read, if its value is something less than or equal to the min_alarm parameter or if it is something more or equal to the max_alarm parameter, the attribute quality factor will be set to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

A.2.2.2.2 The min_warning and max_warning parameters These two parameters have a meaning only for attribute of the Tango::READ, Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. When the attribute is read, if its value is something less than or equal to the min_warning parameter or if it is something more or equal to the max_warning parameter, the attribute quality factor will be set to Tango::ATTR_WARNING and if the device state is Tango::ON, it is switched to Tango::ALARM. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

A.2.2.2.3 The delta_t and delta_val parameters These two parameters have a meaning only for attribute of the Tango::READ_WRITE and Tango::READ_WITH_WRITE read/write type and for numerical data type. They specify if and how the RDS alarm is used. When the attribute is read, if the difference between its read value and the last written value is something more than or equal to the delta_val parameter and if at least delta_val milli seconds occurs since the last write operation, the attribute quality factor will be set to Tango::ATTR_ALARM and if the device state is Tango::ON, it is switched to Tango::ALARM. Even if these parameters are numbers, they are returned as strings by the device *get_attribute_config()* call.

A.2.2.3 The event related configuration parameters

Six event related attribute parameters are modifiable at run-time via a device call or via the property database.

Parameter name	Parameter description
rel_change	Relative change triggering change event
abs_change	Absolute change triggering change event
period	Period for periodic event
archive_rel_change	Relative change for archive event
archive_abs_change	Absolute change for archive event
archive_period	Period for change archive event

A.2.2.3.1 The rel_change and abs_change parameters Rel_change is a property with a maximum of 2 values (comma separated). It specifies the increasing and decreasing relative change of the attribute value (w.r.t. the value of the previous change event) which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. It's the absolute value of these values which is taken into account. If only one value is specified then it is used for the increasing and decreasing change.

Abs_change is a property of maximum 2 values (comma separated). It specifies the increasing and decreasing absolute change of the attribute value (w.r.t the value of the previous change event) which will trigger the event. If the attribute is a spectrum or an image then a change event is generated if any one of the attribute value's satisfies the above criterium. If only one value is specified then it is used for the increasing and decreasing change. If no values are specified then the relative change is used.

A.2.2.3.2 The periodic period parameter The minimum time between events (in milliseconds). If no property is specified then a default value of 1 second is used.

A.2.2.3.3 The archive_rel_change, archive_abs_change and archive_period parameters archive_rel_change is an array property of maximum 2 values which specifies the positive and negative relative change w.r.t. the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then a default fo +-10% is used

archive_abs_change is an array property of maximum 2 values which specifies the positive and negative absolute change w.r.t the previous attribute value which will trigger the event. If the attribute is a spectrum or an image then an archive event is generated if any one of the attribute value's satisfies the above criterium. If only one property is specified then it is used for the positive and negative change. If no properties are specified then the relative change is used.

archive_period is the minimum time between archive events (in milliseconds). If no property is specified, no periodic archiving events are send.

A.2.3 Setting modifiable attribute parameters

A default value is given to all modifiable attribute parameters by the Tango core classes. Nevertheless, it is possible to modify these values in source code at attribute creation time or via the database. Values retrieved from the database have a higher priority than values given at attribute creation time. The attribute parameters are therefore initialized from:

1. The Database
2. If nothing in database, from the Tango class default
3. If nothing in database nor in Tango class default, from the library default value

The default value set by the Tango core library are

Parameter type	Parameter name	Library default value
general purpose	description	"No description"
	label	attribute name
	unit	"No unit"
	standard_unit	"No standard unit"
	display_unit	"No display unit"
	format	6 characters with 2 decimal
	min_value	"Not specified"
	max_value	"Not specified"
alarm parameters	min_alarm	"Not specified"
	max_alarm	"Not specified"
	min_warning	"Not specified"
	max_warning	"Not specified"
	delta_t	"Not specified"
	delta_val	"Not specified"
event parameters	rel_change	"Not specified"
	abs_change	"Not specified"
	period	1000 (mS)
	archive_rel_change	"Not specified"
	archive_abs_change	"Not specified"
	archive_period	"Not specified"

It is possible to set modifiable parameters via the database at two levels :

1. At class level
2. At device level. Each device attribute have all its modifiable parameters sets to the value defined at class level. If the setting defined at class level is not correct for one device, it is possible to re-define it.

If we take the example of a class called *BumperPowerSupply* with three devices called *sr/bump/1*, *sr/bump/2* and *sr/bump/3* and one attribute called *wanted_current*. For the first two bumpers, the max_value is equal to 500. For the third one, the max_value is only 400. If the max_value parameter is defined at class level with the value 500, all devices will have 500 as max_value for the *wanted_current* attribute. It is necessary to re-defined this parameter at device level in order to have the max_value for device sr/bump/3 set to 400.

For the description, label, unit, standard_unit, display_unit and format parameters, it is possible to return them to their default value by setting them to an empty string.

A.2.4 Resetting modifiable attribute parameters

It is possible to reset attribute parameters to their default value at any moment. This could be done via the network call available through the DeviceProxy::set_attribute_config() method family. This call takes attribute parameters as strings. The following table describes which string has to be used to reset attribute parameters to their default value. In this table, the user default are the values given within Pogo in the "Properties" tab of the attribute edition window (or in in Tango class code using the Tango::UserDefaultAttrProp class).

Input string	Action
"Not specified"	Reset to **library** default
"" (empty string)	Reset to **user** default if any. Otherwise, reset to **library** default
"NaN"	Reset to Tango **class** default if any Otherwise, reset to **user** default (if any) or to **library** default

Let's take one exemple: For one attribute belonging to a device, we have the following attribute parameters:

Parameter name	Def. class	Def. user	Def. lib
unit			No unit
min_value		5	Not specified
max_value	50		Not specified
rel_change	5	10	Not specified

The string "Not specified" sent to each attribute parameter will set attribute parameter value to "No unit" for unit, "Not specified" for min_value, "Not specified" for max_value and "Not specified" as well for rel_change. The empty string sent to each attribute parameter will result with "No unit" for unit, 5 for min_value, "Not specified" for max_value and 10 for rel_change. The string "NaN" will give "No unit" for unit, 5 for min_value, 50 for max_value and 5 for rel_change.

C++ specific: Instead of the string "Not specified" and "NaN", the preprocessor define **AlrmValueNotSpec** and **NotANumber** can be used.

A.3 Device class parameter

A device documentation field is also defined at Tango device class level. It is defined as Tango device class level because each device belonging to a Tango device class should have the same behaviour and therefore the same documentation. This field is store in the DeviceClass class. It is possible to set this field via a class property. This property name is

class name->doc_url

and is retrieved when instance of the DeviceClass object is created. A default value is defined for this field.

A.4 The device black box

This black box is a help tool to ease debugging session for a running device server. The TANGO core software records every device request in this black box. A tango client is able to retrieve the black box contents with a specific CORBA operation availabble for every device. Each black box entry is returned as a string with the following information :

- The date where the request has been executed by the device. The date format is dd/mm/yyyy hh24:mi:ss:SS (The last field is the second hundredth number).
- The type of CORBA requests. In case of attributes, the name of the requested attribute is returned. In case of operation, the operation type is returned. For "command_inout" operation, the command name is returned.
- The client host name

A.5 Automatically added commands

As already mentionned in this documentation, each Tango device supports at least three commands which are State, Status and Init. The following array details command input and output data type

Command name	Input data type	Output data type
State	void	Tango::DevState
Status	void	Tango::DevString
Init	void	void

A.5.1 The State command

This command gets the device state (stored in its *device_state* data member) and returns it to the caller. The device state is a variable of the Tango_DevState type (packed into a CORBA Any object when it is returned by a command)

A.5.2 The Status command

This command gets the device status (stored in its *device_status* data member) and returns it to the caller. The device status is a variable of the string type.

A.5.3 The Init command

This commands re-initialise a device keeping the same network connection. After an Init command executed on a device, it is not necessary for client to re-connect to the device. This command first calls the device *delete_device()* method and then execute its *init_device()* method. For C++ device server, all the memory allocated in the *init_device()* method must be freed in the *delete_device()* method. The language device desctructor automatically calls the *delete_device()* method.

A.6 DServer class device commands

As already explained in 8.1.7.2, each device server process has its own Tango device. This device supports the three commands previously described plus 28 commands (for C++ device server, only 26 for Java device server) which are DevRestart, RestartServer, QueryClass, QueryDevice, Kill, QueryWizardClassProperty, QueryWizardDevProperty, QuerySubDevice, the polling related commands which are StartPolling, StopPolling, AddObjPolling, RemObjPolling, UpdObjPollingPeriod, PolledDevice and DevPollStatus, the device locking related commands which are LockDevice, UnLockDevice, ReLockDevices and DevLockStatus, the event related commands called EventSubscriptionChange and ZmqEventSubscriptionChange (only for C++) and finally the logging related commands which are AddLoggingTarget, RemoveLoggingTarget, GetLoggingTarget, GetLoggingLevel, SetLoggingLevel, StopLogging and StartLogging. The following table give all commands input and output data types

Command name	Input data type	Output data type
State	void	Tango::DevState
Status	void	Tango::DevString
Init	void	void
DevRestart	Tango::DevString	void
RestartServer	void	void
QueryClass	void	Tango::DevVarStringArray
QueryDevice	void	Tango::DevVarStringArray
Kill	void	void
QueryWizardClassProperty	Tango::DevString	Tango::DevVarStringArray
QueryWizardDevProperty	Tango::DevString	Tango::DevVarStringArray
QuerySubDevice	void	Tango::DevVarStringArray
StartPolling	void	void
StopPolling	void	void
AddObjPolling	Tango::DevVarLongStringArray	void
RemObjPolling	Tango::DevVarStringArray	void
UpdObjPollingPeriod	Tango::DevVarLongStringArray	void
PolledDevice	void	Tango::DevVarStringArray
DevPollStatus	Tango::DevString	Tango::DevVarStringArray
LockDevice	Tango::DevVarLongStringArray	void
UnLockDevice	Tango::DevVarLongStringArray	Tango::DevLong
ReLockDevices	Tango::DevVarStringArray	void
DevLockStatus	Tango::DevString	Tango::DevVarLongStringArray
EventSubscribeChange	Tango::DevVarStringArray	Tango::DevLong
ZmqEventSubscriptionChange	Tango::DevVarStringArray	Tango::DevVarLongStringArray
AddLoggingTarget	Tango::DevVarStringArray	void
RemoveLoggingTarget	Tango::DevVarStringArray	void
GetLoggingTarget	Tango::DevString	Tango::DevVarStringArray
GetLoggingLevel	Tango::DevVarStringArray	Tango::DevVarLongStringArray
SetLoggingLevel	Tango::DevVarLongStringArray	void
StopLogging	void	void
StartLogging	void	void

The device description field is set to “A device server device”. Device server started with the -file command line option also supports a command called QueryEventChannelIOR. This command is used interanally by the Tango kernel classes when the event system is used with device server using database on file.

A.6.1 The State command

This device state is always set to ON

A.6.2 The Status command

This device status is always set to “The device is ON” followed by a new line character and a string describing polling thread status. This string is either “The polling is OFF” or “The polling is ON” according to polling state.

A.6.3 The DevRestart command

The DevRestart command restart a device. The name of the device to be re-started is the command input parameter. The command destroys the device by calling its destructor and re-create it from its constructor.

A.6.4 The RestartServer command

The DevRestartServer command restarts all the device pattern(s) embedded in the device server process. Therefore, all the devices implemented in the server process are destroyed and re-built[1]. The network connection between client(s) and device(s) implemented in the device server process is destroyed and re-built.

Executing this command allows a complete restart of the device server without stopping the process.

A.6.5 The QueryClass command

This command returns to the client the list of Tango device class(es) embedded in the device server. It returns only class(es) implemented by the device server programmer. The DServer device class name (implemented by the TANGO core software) is not returned by this command.

A.6.6 The QueryDevice command

This command returns to the client the list of device name for all the device(s) implemented in the device server process. Each device name is returned using the following syntax :

<class name>::<device name>

The name of the DServer class device is not returned by this command.

A.6.7 The Kill command

This command stops the device server process. In order that the client receives a last answer from the server, this command starts a thread which will after a short delay, kills the device server process.

A.6.8 The QueryWizardClassProperty command

This command returns the list of property(ies) defined for a class stored in the device server process property wizard. For each property, its name, a description and a default value is returned.

A.6.9 The QueryWizardDevProperty command

This command returns the list of property(ies) defined for a device stored in the device server process property wizard. For each property, its name, a description and a default value is returned.

[1] Their black-box is also destroyed and re-built

A.6.10 The QuerySubDevice command

This command returns the list of sub-device(s) imported by each device within the server. A sub-device is a device used (to execute command(s) and/or to read/write attribute(s)) by one of the device server process devices. There is one element in the returned strings array for each sub-device. The syntax of each string is the device name, a space and the sub-device name. In case of device server process starting threads using a sub-device, it is not possible to link this sub-device to any process devices. In such a case, the string contains only the sub-device name

A.6.11 The StartPolling command

This command starts the polling thread

A.6.12 The StopPolling command

This command stops the polling thread

A.6.13 The AddObjPolling command

This command adds a new object in the list of object(s) to be polled. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and three strings. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
svalue[1]	Object type (“command“ or “attribute“)
svalue[2]	Object name
lvalue[0]	polling period in mS

The object type string is case independent. The object name string (command name or attribute name) is case dependant. This command does not start polling if it is stopped. This command is not allowed in case the device is locked and the command requester is not the lock owner.

A.6.14 The RemObjPolling command

This command removes an object of the list of polled objects. The command input data type is a Tango::DevVarStringArray with three strings. These strings meaning are :

String	Meaning
string[0]	Device name
string[1]	Object type (“command“ or “attribute“)
string[2]	Object name

The object type string is case independent. The object name string (command name or attribute name) is case dependant. This command is not allowed in case the device is locked and the command requester is not the lock owner.

A.6.15 The UpdObjPollingPeriod command

This command changes the polling period for a specified object. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and three strings. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
svalue[1]	Object type (“command“ or “attribute“)
svalue[2]	Object name
lvalue[0]	new polling period in mS

The object type string is case independent. The object name string (command name or attribute name) is case dependant. This command does not start polling if it is stopped. This command is not allowed in case the device is locked and the command requester is not the lock owner.

A.6.16 The PolledDevice command

This command returns the name of device which are polled. Each string in the Tango::DevVarStringArray returned by the command is a device name which has at least one command or attribute polled. The list is alphabetically sorted.

A.6.17 The DevPollStatus command

This command returns a polling status for a specific device. The input parameter is a device name. Each string in the Tango::DevVarStringArray returned by the command is the polling status for each polled device objects (command or attribute). For each polled objects, the polling status is :

- The object name
- The object polling period (in mS)
- The object polling ring buffer depth
- The time needed (in mS) for the last command execution or attribute reading
- The time since data in the ring buffer has not been updated. This allows a check of the polling thread
- The delta time between the last records in the ring buffer. This allows checking that the polling period is respected by the polling thread.
- The exception parameters in case of the last command execution or the last attribute reading failed.

A new line character is inserted between each piece of information.

A.6.18 The LockDevice command

This command locks a device for the calling process. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and one string. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
lvalue[0]	Lock validity

A.6.19 The UnLockDevice command

This command unlocks a device. The command input parameters are embedded within a Tango::DevVarLongStringArray data type with one long data and one string. The input parameters are:

Command parameter	Parameter meaning
svalue[0]	Device name
lvalue[0]	Force flag

The force flag parameter allows a client to unlock a device already locked by another process (for admin usage only)

A.6.20 The ReLockDevices command

This command re-lock devices. The input argument is the list of devices to be re-locked. It's an error to re-lock a device which is not already locked.

A.6.21 The DevLockStatus command

This command returns a device locking status to the caller. Its input parameter is the device name. The output parameters are embedded within a Tango::DevVarLongStringArray data type with three strings and six long. These data are

Command parameter	Parameter meaning
svalue[0]	Locking string
svalue[1]	CPP client host IP address or "Not defined"
svalue[2]	Java VM main class for Java client or "Not defined"
lvalue[0]	Lock flag (1 if locked, 0 othterwise)
lvalue[1]	CPP client host IP address or 0 for Java locker
lvalue[2]	Java locker UUID part 1or 0 for CPP locker
lvalue[3]	Java locker UUID part 2 or 0 for CPP locker
lvalue[4]	Java locker UUID part 3 or 0 for CPP locker
lvalue[5]	Java locker UUID part 4 or 0 for CPP locker

A.6.22 The EventSubscriptionChange command (C++ server only)

This command is used as a piece of the "heartbeat" system between an event client and the device server generating the event. There is no reason to generate events if there is no client which has subscribed to it. It is used by the *DeviceProxy::subscribe_event()* method and one of the event thread on the client side to inform the server to keep on generating events for the attribute in question. It reloads the subscription timer with the current time. Events are not generated when there are no clients subscribed within the last 10 minutes. The input parameters are:

Command parameter	Parameter meaning
argin[0]	Device name
argin[1]	Attribute name
argin[2]	action ("subscribe" or "unsubsribe")
argin[3]	event name ("change", "periodic", "archive","attr_conf")

The command output data is the simply the Tango release used by the device server process. This is necessary for compatibility reason.

A.6.23 The ZmqEventSubscriptionChange command (C++ server only)

This command is used as a piece of the "heartbeat" system between an event client and the device server generating the event when client and/or device server uses Tango release 8 or above. There is no reason to generate events if there is no client which has subscribed to it. It is used by the *DeviceProxy::subscribe_event()* method and one of the event thread on the client side to inform the server to keep on generating events for the attribute in question. It reloads the subscription timer with the current time. Events are not generated when there are no clients subscribed within the last 10 minutes. The input parameters are the same than the one used for the EventSubscriptionChange command. They are:

Command in parameter	Parameter meaning
argin[0]	Device name
argin[1]	Attribute name
argin[2]	action ("subscribe" or "unsubsribe")
argin[3]	event name ("change", "periodic", "archive","attr_conf")

The command output parameters aer all the necessary data to build one event connection between a client and the device server process generating the events. This means:

Command out parameter	Parameter meaning
svalue[0]	Heartbeat ZMQ socket connect end point
svalue[1]	Event ZMQ socket connect end point
lvalue[0]	Tango lib release used by device server
lvalue[1]	Device IDL release
lvalue[2]	Subscriber HWM
lvalue[3]	Rate (Multicasting related)
lvalue[4]	IVL (Multicasting related)

A.6.24 The AddLoggingTarget command

This command adds one (or more) logging target(s) to the specified device(s). The command input parameter is an array of string logically composed of {device_name, target_type::target_name} groups where the elements have the following semantic:

- device_name is the name of the device which logging behavior is to be controlled. The wildcard "*" is supported to apply the modification to all devices encapsulated within the device server (e.g. to ask all devices to log to the same device target).
- target_type::target_name: target_type is one of the supported target types and target_name, the name of the target. Supported target types are: *console*, *file* and *device*. For a device target, target_name must contain the name of a log consumer device (as defined in A.8). For a file target, target_name is the full path to the file to log to. If omitted the device's name is used to build the file name (domain_family_member.log). Finally, target_name is ignored in the case of a console target and can be omitted.

This command is not allowed in case the device is locked and the command requester is not the lock owner.

A.6.25 The RemoveLoggingTarget command

Remove one (or more) logging target(s) from the specified device(s).The command input parameter is an array of string logically composed of {device_name, target_type::target_name} groups where the elements have the following semantic:

- device_name: the name of the device which logging behavior is to be controlled. The wildcard "*" is supported to apply the modification to all devices encapsulated within the device server (e.g. to ask all devices to stop logging to a given device target).
- target_type::target_name: target_type is one of the supported target types and target_name, the name of the target. Supported target types are: *console*, *file* and *device*. For a device target, target_name must contain the name of a log consumer device (as defined in A.8). For a file target, target_name is the full path to the file to log to. If omitted the device's name is used to build the file name (domain_family_member.log). Finally, target_name is ignored in the case of a console target and can be omitted.

The wildcard "*" is supported for target_name. For instance, RemoveLoggingTarget (["*", "device::*"]) removes all the device targets from all the devices running in the device server. This command is not allowed in case the device is locked and the command requester is not the lock owner.

A.6.26 The GetLoggingTarget command

Returns the current target list of the specified device. The command parameter device_name is the name of the device which logging target list is requested. The list is returned as a DevVarStringArray containing target_type::target_name elements.

A.6.27 The GetLoggingLevel command

Returns the logging level of the specified devices. The command input parameter device_list contains the names of the devices which logging target list is requested. The wildcard "*" is supported to get the logging level of all the devices running within the server. The string part of the result contains the name of the devices and its long part contains the levels. Obviously, result.lvalue[i] is the current logging level of the device named result.svalue[i].

A.6.28 The SetLoggingLevel command

Changes the logging level of the specified devices. The string part of the command input parameter contains the device names while its long part contains the logging levels. The set of possible values for levels is: 0=OFF, 1=FATAL, 2=ERROR, 3=WARNING, 4=INFO, 5=DEBUG.

The wildcard "*" is supported to assign all devices the same logging level. For instance, SetLoggingLevel (["*"] [3]) set the logging level of all the devices running within the server to WARNING. This command is not allowed in case the device is locked and the command requester is not the lock owner.

A.6.29 The StopLogging command

For all the devices running within the server, StopLogging saves their current logging level and set their logging level to OFF.

A.6.30 The StartLogging command

For each device running within the server, StartLogging restores their logging level to the value stored during a previous StopLogging call.

A.7 DServer class device properties

This device has two properties related to polling threads pool management. These properties are described in the following table

Property name	property rule	default value
polling_threads_pool_size	Max number of thread in the polling pool	1
polling_threads_pool_conf	Polling threads pool configuration	

The rule of the polling_threads_pool_size is to define the maximun number of thread created for the polling threads pool size. The rule of the polling_threads_pool_conf is to define which thread in the pool is in charge of all the polled object(s) of which device. This property is an array of strings with one string per used thread in the pool. The content of the string is simply a device name list with device name splitted by a comma. Example of polling_threads_pool_conf property for 3 threads used:

```
dserver/<ds exec name>/<inst. name>/polling_threads_pool_conf-> the/dev/01
                  the/dev/02,the/dev/06
                  the/dev/03
```

Thread number 2 is in charge of 2 devices. Note that there is an entry in this list only for the used threads in the pool.

A.8 Tango log consumer

A.8.1 The available Log Consumer

One implementation of a log consumer associated to a graphical user interface is available within Tango. It is a standalone java application called **LogViewer** based on the publicly available chainsaw application from the log4j package. It supports two way of running which are:

- The static mode: In this mode, LogViewer is started with a parameter which is the name of the log consumer device implemented by the application. All messages sent by devices with a logging target type set to *device* and with a logging target name set to the same device name than the device name passed as application parameter will be displayed (if the logging level allows it).
- The dynamic mode: In this mode, the name of the log consumer device implemented by the application is build at application startup and is dynamic. The user with the help of the graphical interface chooses device(s) for which he want to see log messages.

A.8.2 The Log Consumer interface

A Tango Log Consumer device is nothing but a tango device supporting the following tango command :

void log (Tango::DevVarStringArray details)

where details is an array of string carrying the log details. Its structure is:

- details[0] : the timestamp in millisecond since epoch (01.01.1970)
- details[1] : the log level
- details[2] : the log source (i.e. device name)
- details[3] : the log message
- details[4] : the log NDC (contextual info) - Not used but reserved
- details[5] : the thread identifier (i.e. the thread from which the log request comes from)

These log details can easily be extended. Any tango device supporting this command can act as a device target for other devices.

A.9 Control system specific

It is possible to define a few control system parameters. By control system, we mean for each set of computers having the same database device server (the same TANGO_HOST environment variable)

A.9.1 The device class documentation default value

Each control system may have it's own default device class documentation value. This is defined via a class property. The property name is

Default->doc_url

It's retrieved if the device class itself does not define any doc_url property. If the Default->doc_url property is also not defined, a hard-coded default value is provided.

A.9.2 The services definition

The property used to defined control system services is named **Services** and belongs to the free object **CtrlSystem**. This property is an array of strings. Each string defines a service available within the control system. The syntax of each service definition is

Service name/Instance name:service device name

A.9.3 Tuning the event system buffers (HWM)

Starting with Tango release 8, ZMQ is used for the event based communication between clients and device server processes. ZMQ implementation provides asynchronous communication in the sense that the data to be transmitted is first stored in a buffer and then really sent on the network by dedicated threads. The size of this buffers (on client and device server side) is called High Water Mark (HWM) and is tunable. This is tunable at several level.

1. The library set a default value of **1000** for both buffers (client and device server side)
2. Control system properties used to tune these size are named **DSEventBufferHwm** (device server side) and **EventBufferHwm** (client side). They both belongs to the free object **CtrlSystem**. Each property is the max number of events storable in these buffer.
3. At client or device server level using the library calls *Util::set_ds_event_buffer_hwm()* documented in [24] or *ApiUtil::set_event_buffer_hwm()* documented in 6.7
4. Using environment variables TANGO_DS_EVENT_BUFFER_HWM or TANGO_EVENT_BUFFER_HWM

A.9.4 Allowing NaN when writing attributes (floating point)

A property named **WAttrNaNAllowed** belonging to the free object **CtrlSystem** allows a Tango control system administrator to allow or disallow NaN numbers when writing attributes of the DevFloat or DevDouble data type. This is a boolean property and by default, it's value is taken as false (Meaning NaN values are rejected).

A.9.5 Summary of CtrlSystem free object properties

The following table summarizes properties defined at control system level and belonging to the free object CtrlSystem

Property name	property rule	default value
Services	List of defined services	No default
DsEventBufferHwm	DS event buffer high water mark	1000
EventBufferHwm	Client event buffer high water mark	1000
WAttrNaNAllowed	Allow NaN when writing attr.	false

A.10 C++ specific

A.10.1 The Tango master include file (tango.h)

Tango has a master include file called

tango.h

This master include file includes the following files :

- Tango configuration include file : **tango_config.h**
- CORBA include file : **idl/tango.h**
- Some network include files for WIN32 : **winsock2.h** and **mswsock.h**

- C++ streams include file :
  - **iostream**, **sstream** and **fstream**
- Some standard C++ library include files : **memory, string** and **vector**
- The Tango database and device API include files : **dbapi.h** and **devapi.h**
- A list of other Tango include files : **tango_const.h**, **utils.h**, **device.h**, **command.h, except.h, seqvec.h, device_2.h, device_3.h, device_4.h. log4tango.h, attrmanip.h, apiexcept.h, devasyn.h, group.h, filedatabase.h, tango_monitor.h, auto_tango_monitor.h, dserver.h** and **event.h**

A.10.2 Tango specific pre-processor define

The tango.h previously described also defined some pre-processor macros allowing Tango release to be checked at compile time. These macros are:

- TANGO_VERSION_MAJOR
- TANGO_VERSION_MINOR
- TANGO_VERSION_PATCH

For instance, with Tango release 8.1.2, TANGO_VERSION_MAJOR will be set to 8 while TANGO_VERSION_MINOR will be 1 and TANGO_VERSION_PATCH will be 2.

A.10.3 Tango specific types

Operating system free type

Some data type used in the TANGO core software have been defined. They are described in the following table.

Type name	C++ name
TangoSys_MemStream	stringstream
TangoSys_OMemStream	ostringstream
TangoSys_Pid	int
TangoSys_Cout	ostream

These types are defined in the tango_config.h file

A.10.3.1 Template command model related type

As explained in 8.4.8, command created with the template command model uses static casting. Many type definition have been written for these casting.

Class name	Command allowed method (if any)	Command execute method
TemplCommand	Tango::StateMethodPtr	Tango::CmdMethPtr
TemplCommandIn	Tango::StateMethodPtr	Tango::CmdMethPtr_xxx
TemplCommandOut	Tango::StateMethodPtr	Tango::xxx_CmdMethPtr
TemplCommandInOut	Tango::StateMethodPtr	Tango::xxx_CmdMethPtr_yyy

The **Tango::StateMethPtr** is a pointer to a method of the DeviceImpl class which returns a boolean and has one parameter which is a reference to a const CORBA::Any obect.

The **Tango::CmdMethPtr** is a pointer to a method of the DeviceImpl class which returns nothing and needs nothing as parameter.

The **Tango::CmdMethPtr_xxx** is a pointer to a method of the DeviceImpl class which returns nothing and has one parameter. xxx must be set according to the method parameter type as described in the next table

Tango type	short cut (xxx)
Tango::DevBoolean	Bo
Tango::DevShort	Sh
Tango::DevLong	Lg
Tango::DevFloat	Fl
Tango::DevDouble	Db
Tango::DevUshort	US
Tango::DevULong	UL
Tango::DevString	Str
Tango::DevVarCharArray	ChA
Tango::DevVarShortArray	ShA
Tango::DevVarLongArray	LgA
Tango::DevVarFloatArray	FlA
Tango::DevVarDoubleArray	DbA
Tango::DevVarUShortArray	USA
Tango::DevVarULongArray	ULA
Tango::DevVarStringArray	StrA
Tango::DevVarLongStringArray	LSA
Tango::DevVarDoubleStringArray	DSA
Tango::DevState	Sta

For instance, a pointer to a method which takes a Tango::DevVarStringArray as input parameter must be statically casted to a Tango::CmdMethPtr_StrA, a pointer to a method which takes a Tango::DevLong data as input parameter must be statically casted to a Tango::CmdMethPtr_Lg.

The **Tango::xxx_CmdMethPtr** is a pointer to a method of the DeviceImpl class which returns data of one of the Tango type and has no input parameter. xxx must be set according to the method return data type following the same rules than those described in the previous table. For instance, a pointer to a method which returns a Tango::DevDouble data must be statically casted to a Tango::Db_CmdMethPtr.

The **Tango::xxx_CmdMethPtr_yyy** is a pointer to a method of the DeviceImpl class which returns data of one of the Tango type and has one input parameter of one of the Tango data type. xxx and yyy must be set according to the method return data type and parameter type following the same rules than those described in the previous table. For instance, a pointer to a method which returns a Tango::DevDouble data and which takes a Tango::DevVarLongStringArray must be statically casted to a Tango::Db_CmdMethPtr_LSA.

All those type are defined in the tango_const.h file.

A.10.4 Tango device state code

The Tango::DevState type is a C++ enumeration starting at 0. The code associated with each state is defined in the following table.

State name	Value
Tango::ON	0
Tango::OFF	1
Tango::CLOSE	2
Tango::OPEN	3
Tango::INSERT	4
Tango::EXTRACT	5
Tango::MOVING	6
Tango::STANDBY	7
Tango::FAULT	8
Tango::INIT	9
Tango::RUNNING	10
Tango::ALARM	11
Tango::DISABLE	12
Tango::UNKNOWN	13

A strings array called **Tango::DevStateName** can be used to get the device state as a string. Use the Tango device state code as index into the array to get the correct string.

A.10.5 Tango data type

A "define" has been created for each Tango data type. This is summarized in the following table

Type name	Type code	Value
Tango::DevBoolean	Tango::DEV_BOOLEAN	1
Tango::DevShort	Tango::DEV_SHORT	2
Tango::DevLong	Tango::DEV_LONG	3
Tango::DevFloat	Tango::DEV_FLOAT	4
Tango::DevDouble	Tango::DEV_DOUBLE	5
Tango::DevUShort	Tango::DEV_USHORT	6
Tango::DevULong	Tango::DEV_ULONG	7
Tango::DevString	Tango::DEV_STRING	8
Tango::DevVarCharArray	Tango::DEVVAR_CHARARRAY	9
Tango::DevVarShortArray	Tango::DEVVAR_SHORTARRAY	10
Tango::DevVarLongArray	Tango::DEVVAR_LONGARRAY	11
Tango::DevVarFloatArray	Tango::DEVVAR_FLOATARRAY	12
Tango::DevVarDoubleArray	Tango::DEVVAR_DOUBLEARRAY	13
Tango::DevVarUShortArray	Tango::DEVVAR_USHORTARRAY	14
Tango::DevVarULongArray	Tango::DEVVAR_ULONGARRAY	15
Tango::DevVarStringArray	Tango::DEVVAR_STRINGARRAY	16
Tango::DevVarLongStringArray	Tango::DEVVAR_LONGSTRINGARRAY	17
Tango::DevVarDoubleStringArray	Tango::DEVVAR_DOUBLESTRINGARRAY	18
Tango::DevState	Tango::DEV_STATE	19
Tango::ConstDevString	Tango::CONST_DEV_STRING	20
Tango::DevVarBooleanArray	Tango::DEVVAR_BOOLEANARRAY	21
Tango::DevUChar	Tango::DEV_UCHAR	22
Tango::DevLong64	Tango::DEV_LONG64	23

Tango::DevULong64	Tango::DEV_ULONG64	24
Tango::DevVarLong64Array	Tango::DEVVAR_LONG64ARRAY	25
Tango::DevVarULong64Array	Tango::DEVVAR_ULONG64ARRAY	26
Tango::DevInt	Tango::DEV_INT	27
Tango::DevEncoded	Tango::DEV_ENCODED	28

For command which do not take input parameter, the type code Tango::DEV_VOID (value = 0) has been defined.

A strings array called **Tango::CmdArgTypeName** can be used to get the data type as a string. Use the Tango data type code as index into the array to get the correct string.

A.10.6 Tango command display level

Like attribute, Tango command has a display level. The Tango::DispLevel type is a C++ enumeration starting at 0. The code associated with each command display level is already described in page 326

As for attribute, this parameter allows a graphical application to support two types of operation :

- An operator mode for day to day operation
- An expert mode when tuning is necessary

According to this parameter, a graphical application knows if the command is for the operator mode or for the expert mode.

A.11 Java specific

A.11.1 Packages

All the Tango core classes are bundled in the a Java package called **fr.esrf.TangoDs**. All the classes generated by the IDL compiler are bundled in a Java package called **fr.esrf.Tango**. All the Tango Java API classes are bundled in Java packages called **fr.esrf.TangoApi** and **fr.esrf.TangoApi.Group**. All the CORBA related classes are stored in a package called **org.omg.CORBA**. These package Tango, TangoDs, TangoApi, Group and CORBA are stored in the same jar file called **TangORB.jar**.

A.12 Device server process option and environment variables

A.12.1 Classical device server

The synopsis of a device server process is

ds_name instance_name [OPTIONS]

The supported options are :

- **-h, -? -help**
  Print the device server synopsis and a list of instance name defined in the database for this device server. An instance name in not mandatory in the command line to use this option
- **-v[trace level]**
  Set the verbose level. If no trace level is given, a default value of 4 is used
- **-file=<file name path>**
  Start a device server using an ASCII file instead of the Tango database.

- **-nodb**
  Start a device server without using the database.
- **-dlist <device name list>**
  Give the device name list. This option is supported only with the -nodb option.
- **ORB options** (started with -ORBxxx)
  Options directly passed to the underlying ORB. Should be rarely used except the -ORBendPoint option for device server not using the database

A.12.2 Device server process as Windows service

When used as a Windows service, a Tango device server supports several new options. These options are :

- **-i**
  Install the service
- **-s**
  Install the service and choose the automatic startup mode
- **-u**
  Un-install the service
- **-dbg**
  Run in console mode to debug service. The service must have been installed prior to use it.

Note that these options must be used after the device server instance name.

A.12.3 Environment variables

A few environment variables can be used to tune a Tango control system. TANGO_HOST is the most important one but on top it, some Tango features like Tango logging service or controlled access (if used) can be tuned using environment variable. If these environment variables are not defined, the software searches in the file **$HOME/.tangorc** for its value. If the file is not defined or if the environment variable is also not defined in this file, the software searches in the file **/etc/tangorc** for its value. For Windows, the file is **$TANGO_ROOT/tangorc** TANGO_ROOT being the mandatory environment variable of the Windows binary distribution.

A.12.3.1 TANGO_HOST

This environment variable is the anchor of the system. It specifies where the Tango database server is running. Most of the time, its syntax is

TANGO_HOST=<host>:<port>

host is the name of the computer where the database server is running and port is th eport number on which it is litenning. If you want to have a Tango control system which has several database servers (but only one database) in order to survive a database server crashes, use the following syntax

TANGO_HOST=<host_1>:<port_1>,<host_2>:<port_2>,<host_3>:<port_3>

Obviously, host_1 is the name of the computer where the first database server is running, port_1 is the port number on which this server is listenning. host_2 is the name of the computer where the second database server is running and port_2 is its port number. All access to database will automatically switch from one server to another one in the list if the one which was used has died.

A.12.3.2 Tango Logging Service (TANGO_LOG_PATH)

The TANGO_LOG_PATH environment variable can be used to specify the log files location. If not set it defaults to /tmp/tango-<user logging name> under Unix and C:/tango-<user logging name> under Windows. For a given device-server, the files are actually saved into $TANGO_LOG_PATH/{ server_name}/{ server_instance_name}. This means that all the devices running within the same process log into the same directory.

A.12.3.3 The database and controlled access server (MYSQL_USER, MYSQL_PASSWORD and MYSQL_HOST)

The Tango database server and the controlled access server (if used) need to connect to the MySQL database. They are using three environment variables called MYSQL_USER, MYSQL_PASSWORD to know which user/password they must use to access the database and MYSQL_HOST in case the MySQL database is running on another host. The MYSQL_HOST environment variable allows you to specify the host and port number where MySQL is running. Its syntax is

host:port

The port definition is optional. If it is not specified, the default MySQL port will be used. If these environment variables are not defined, they will connect to the DBMS using the "root" login on localhost with the MySQL default port number (3306).

A.12.3.4 The controlled access

Even if a controlled access system is running, it is possible to by-pass it if in the environment of the client application the environment variable SUPER_TANGO is defined to "true".

A.12.3.5 The event buffer size

If required, the event buffer used by the ZMQ software could be tuned using environment variables. These variables are named TANGO_DS_EVENT_BUFFER_HWM for the event buffer on a device server side and TANGO_EVENT_BUFFER_HWM for the event buffer on the client size. Both of them are a number which is the maximum number of events which could be stored in these buffers.

Appendix B

The TANGO IDL file : Module Tango

The fundamental idea of a device as a network object which has methods and data has been retained for TANGO. In TANGO objects are real C++/Java objects which can be instantiated and accessed via their methods and data by the client as if they were local objects. This interface is defined in CORBA IDL. The fundamental interface is Device. All TANGO control objects will be of this type i.e. they will implement and offer the Device interface. Some wrapper classes group in an API will hide the calls to the Device interface from the client so that the client will only see the wrapper classes. All CORBA details will be hidden from the client as far as possible.

B.1 Aliases

AttributeConfigList

typedef sequence<AttributeConfig> AttributeConfigList;

AttributeConfigList_2

typedef sequence<AttributeConfig_2> AttributeConfigList_2;

AttributeConfigList_3

typedef sequence<AttributeConfig_3> AttributeConfigList_3;

AttributeDimList

typedef sequence<AttributeDim> AttributeDimList;

AttributeValueList

typedef sequence<AttributeValue> AttributeValueList;

AttributeValueList_3

typedef sequence<AttributeValue_3> AttributeValueList_3;

AttributeValueList_4

typedef sequence<AttributeValue_4> AttributeValueList_4;

AttrQualityList

typedef sequence<AttrQuality> AttrQualityList;

CppClntIdent

typedef unsigned long CppClntIdent;

DevAttrHistoryList

typedef sequence<DevAttrHistory> DevAttrHistoryList;

DevAttrHistoryList_3

typedef sequence<DevAttrHistory_3> DevAttrHistoryList_3;

DevBoolean

typedef boolean DevBoolean;

DevCmdHistoryList

typedef sequence<DevCmdHistory> DevCmdHistoryList

DevCmdInfoList

typedef sequence<DevCmdInfo> DevCmdInfoList;

DevCmdInfoList_2

typedef sequence<DevCmdInfo_2> DevCmdInfoList_2;

DevDouble

typedef double DevDouble;

DevErrorList

typedef sequence<DevError> DevErrorList;

DevErrorListList

typedef sequence<DevErrorList> DevErrorListList;

DevFloat

typedef float DevFloat;

DevLong

typedef long DevLong;

DevShort

typedef short DevShort;

DevString

typedef string DevString;

DevULong

typedef unsigned long DevULong;

DevUShort

typedef unsigned short DevUShort;

DevVarCharArray

typedef sequence<octet> DevVarCharArray;

DevVarDoubleArray

typedef sequence<double> DevVarDoubleArray;

DevVarEncodedArray

typedef sequence<DevEncoded> DevVarEncodedArray;

DevVarFloatArray

typedef sequence<float> DevVarFloatArray;

DevVarLongArray

typedef sequence<long> DevVarLongArray;

DevVarShortArray

typedef sequence<short> DevVarShortArray;

DevVarStateArray

typedef sequence<DevState> DevVarStateArray;

DevVarStringArray

```
typedef sequence<string> DevVarStringArray;
```

DevVarULongArray

```
typedef sequence<unsigned long> DevVarULongArray;
```

DevVarUShortArray

```
typedef sequence<unsigned short> DevVarUShortArray;
```

EltInArrayList

```
typedef sequence<EltInArray> EltInArrayList;
```

JavaUUID

```
typedef unsigned long long JavaUUID[2];
```

NamedDevErrorList

```
typedef sequence<NamedDevError> NamedDevErrorList;
```

TimeValList

```
typedef sequence<TimeVal> TimeValList;
```

B.2 Enums

AttrDataFormat

```
enum AttrDataFormat
{
  SCALAR,
  SPECTRUM,
  IMAGE,
  FMT_UNKNOWN
};
```

AttributeDataType

```
enum AttributeDataType
{
  ATT_BOOL,
  ATT_SHORT,
  ATT_LONG,
  ATT_LONG64,
  ATT_FLOAT,
  ATT_DOUBLE,
  ATT_UCHAR,
```

```
    ATT_USHORT,
    ATT_ULONG,
    ATT_ULONG64,
    ATT_STRING,
    ATT_STATE,
    DEVICE_STATE,
    ATT_ENCODED,
    NO_DATA
  };
```

AttrQuality

```
  enum AttrQuality
  {
    ATTR_VALID,
    ATTR_INVALID,
    ATTR_ALARM,
    ATTR_CHANGING,
    ATTR_WARNING
  };
```

AttrWriteType

```
  enum AttrWriteType
  {
    READ,
    READ_WITH_WRITE,
    WRITE,
    READ_WRITE
  };
```

DispLevel

```
  enum DispLevel
  {
    OPERATOR,
    EXPERT
  };
```

DevSource

```
  enum DevSource
  {
    DEV,
    CACHE,
    CACHE_DEV
  };
```

DevState

```
enum DevState
{
  ON,
  OFF,
  CLOSE,
  OPEN,
  INSERT,
  EXTRACT,
  MOVING,
  STANDBY,
  FAULT,
  INIT,
  RUNNING,
  ALARM,
  DISABLE,
  UNKNOWN
};
```

ErrSeverity

```
enum ErrSeverity
{
  WARN,
  ERR,
  PANIC
};
```

LockerLanguage

```
enum LockerLanguage
{
  CPP,
  JAVA
};
```

B.3 Structs

ArchiveEventProp

```
struct ArchiveEventProp
{
  string rel_change;
  string abs_change;
  string period;
  DevVarStringArray extensions;
};
```

AttributeAlarm

```
struct AttributeAlarm
{
  string min_alarm;
  string max_alarm;
  string min_warning;
```

```
    string max_warning;
    string delta_t;
    string delta_val;
    DevVarStringArray extensions;
};
```

AttDataReady

```
struct AttributeAlarm
{
    string name;
    long data_type;
    long ctr;
};
```

AttributeConfig

```
struct AttributeConfig
{
    string name;
    AttrWriteType writable;
    AttrDataFormat data_format;
    long data_type;
    long max_dim_x;
    long max_dim_y;
    string description;
    string label;
    string unit;
    string standard_unit;
    string display_unit;
    string format;
    string min_value;
    string max_value;
    string min_alarm;
    string max_alarm;
    string writable_attr_name;
    DevVarStringArray extensions;
};
```

AttributeConfig_2

```
struct AttributeConfig_2
{
    string name;
    AttrWriteType writable;
    AttrDataFormat data_format;
    long data_type;
    long max_dim_x;
    long max_dim_y;
    string description;
    string label;
    string unit;
    string standard_unit;
    string display_unit;
    string format;
```

```
  string min_value;
  string max_value;
  string min_alarm;
  string max_alarm;
  string writable_attr_name;
  DispLevel level;
  DevVarStringArray extensions;
};
```

AttributeConfig_3

```
struct AttributeConfig_3
{
  string name;
  AttrWriteType writable;
  AttrDataFormat data_format;
  long data_type;
  long max_dim_x;
  long max_dim_y;
  string description;
  string label;
  string unit;
  string standard_unit;
  string display_unit;
  string format;
  string min_value;
  string max_value;
  string writable_attr_name;
  DispLevel level;
  AttributeAlarm alarm;
  EventProperties event_prop;
  DevVarStringArray extensions;
  DevVarStringArray sys_extensions;
};
```

AttributeDim

```
struct AttributeDim
{
  long dim_x;
  long dim_y;
};
```

AttributeValue

```
struct AttributeValue
{
  any value;
  AttrQuality quality;
  TimeVal time;
  string name;
  long dim_x;
```

```
  long dim_y;
};
```

AttributeValue_3

```
struct AttributeValue_3
{
  any value;
  AttrQuality quality;
  TimeVal time;
  string name;
  AttributeDim r_dim;
  AttributeDim w_dim;
  DevErrorList err_list;
};
```

AttributeValue_4

```
struct AttributeValue_4
{
  AttrValUnion value;
  AttrQuality quality;
  AttrDataFormat data_format;
  TimeVal time;
  string name;
  AttributeDim r_dim;
  AttributeDim w_dim;
  DevErrorList err_list;
};
```

ChangeEventProp

```
struct ChangeEventProp
{
  string rel_change;
  string abs_change;
  DevVarStringArray extensions;
};
```

DevAttrHistory

```
struct DevAttrHistory
{
  boolean attr_failed;
  AttributeValue value;
  DevErrorList errors;
};
```

DevAttrHistory_3

```
struct DevAttrHistory_3
{
  boolean attr_failed;
  AttributeValue_3 value;
};
```

DevAttrHistory_4

```
struct DevAttrHistory_4
{
  string name;
  TimeValList dates;
  any value;
  AttrQualityList quals;
  EltInArrayList quals_array;
  AttributeDimList r_dims;
  EltInArrayList r_dims_array;
  AttributeDimList w_dims;
  EltInArrayList w_dims_array;
  DevErrorListList errors;
  EltInArrayList errors_array;
};
```

DevCmdHistory

```
struct DevCmdHistory
{
  TimeVal time;
  boolean cmd_failed;
  any value;
  DevErrorList errors;
};
```

DevCmdHistory_4

```
struct DevCmdHistory_4
{
  TimeValList dates;
  any value;
  AttributeDimList dims;
  EltInArrayList dims_array;
  DevErrorListList errors;
  EltInArrayList errors_array;
  long cmd_type;
};
```

DevCmdInfo

```
struct DevCmdInfo
{
  string cmd_name;
  long cmd_tag;
  long in_type;
  long out_type;
  string in_type_desc;
  string out_type_desc;
};
```

DevCmdInfo_2

```
struct DevCmdInfo_2
{
  string cmd_name;
  DispLevel level;
  long cmd_tag;
  long in_type;
  long out_type;
  string in_type_desc;
  string out_type_desc;
};
```

DevEncoded

```
struct DevEncoded
{
  DevString encoded_format;
  DevVarCharArray encoded_data;
};
```

DevError

```
struct DevError
{
  string reason;
  ErrSeverity severity;
  string desc;
  string origin;
};
```

DevInfo

```
struct DevInfo
{
  string dev_class;
  string server_id;
  string server_host;
  long server_version;
  string doc_url;
};
```

DevInfo_3

```
struct DevInfo_3
{
  string dev_class;
  string server_id;
  string server_host;
  long server_version;
  string doc_url;
  string dev_type;
};
```

DevVarDoubleStringArray

```
struct DevVarDoubleStringArray
{
  DevVarDoubleArray dvalue;
  DevVarStringArray svalue;
};
```

DevVarLongStringArray

```
struct DevVarLongStringArray
{
  DevVarLongArray lvalue;
  DevVarStringArray svalue;
};
```

EltInArray

```
struct EltInArray
{
  long start;
  long nb_elt;
};
```

EventProperties

```
struct EventProperties
{
  ChangeEventProp ch_event;
  PeriodicEventProp per_event;
  ArchiveEventProp arch_event;
};
```

JavaClntIdent

```
struct JavaClntIdent
{
  string MainClass;
  JavaUUID uuid;
};
```

NamedDevError

```
struct NamedDevError
{
  string name;
  long index_in_call;
  DevErrorList err_list;
};
```

PeriodicEventProp

```
struct PeriodicEventProp
{
  string period;
  DevVarStringArray extensions;
};
```

TimeVal

```
struct TimeVal
{
  long tv_sec;
  long tv_usec;
  long tv_nsec;
};
```

ZmqCallInfo

```
struct ZmqCallInfo
{
  long version;
  unsigned long ctr;
  string method_name;
  DevVarCharArray oid;
  boolean call_is_except;
};
```

B.4 Unions

AttrValUnion

```
union AttrValUnion switch (AttributeDataType)
{
case ATT_BOOL:
  DevVarBooleanArray bool_att_value;
case ATT_SHORT:
  DevVarShortArray short_att_value;
case ATT_LONG:
  DevVarLongArray long_att_value;
case ATT_LONG64:
  DevVarLong64Array long64_att_value;
case ATT_FLOAT:
  DevVarFloatArray float_att_value;
```

```
case ATT_DOUBLE:
  DevVarDoubleArray double_att_value;
case ATT_UCHAR
  DevVarCharArray uchar_att_value;
case ATT_USHORT:
  DevVarUShortArray ushort_att_value;
case ATT_ULONG:
  DevVarULongArray ulong_att_value;
case ATT_ULONG64:
  DevVarULong64Array ulong64_att_value;
case ATT_STRING:
  DevVarStringArray string_att_value;
case ATT_STATE:
  DevVarStateArray state_att_value;
case DEVICE_STATE:
  DevState dev_state_att;
case ATT_ENCODED:
  DevVarEncodedArray encoded_att_value;
case NO_DATA:
  DevBoolean union_no_data;
};
```

ClntIdent

```
union ClntIdent switch (LockerLanguage)
{
case CPP:
  CppClntIdent cpp_clnt;
case JAVA:
  JavaClntIdent java_clnt;
};
```

B.5 Exceptions

DevFailed

```
exception DevFailed
{
  DevErrorList errors;
};
```

MultiDevFailed

```
exception MultiDevFailed
{
  NamedDevErrorList errors;
};
```

B.6 Interface Tango::Device

The fundamental interface for all TANGO objects. Each Device is a network object which can be accessed locally or via network. The network protocol on the wire will be IIOP. The Device interface implements all the basic functions needed for doing generic synchronous and asynchronous I/O on a device. A Device object has data and actions. Data are represented in the form of Attributes. Actions are represented in the form of Commands. The CORBA Device interface offers attributes and methods to access the attributes and commands. A client will either use these methods directly from C++ or Java or access them via wrapper classes implemented in a API. The Device interface describes only the remote network interface. Implementation features like threads, command security, priority etc. are dealt with in server side of the device server model.

B.6.1 Attributes

adm_name

readonly attribute string adm_name;
adm_name (readonly) - administrator device unique ascii identifier

description

readonly attribute string description;
description (readonly) - general description of device

name

readonly attribute string name;
name (readonly) - unique ascii identifier

state

readonly attribute DevState state;
state (readonly) - device state

status

readonly attribute string status;
status (readonly) - device state as ascii string

B.6.2 Operations

black_box

DevVarStringArray black_box(in long number)
raises(DevFailed);

read list of last N commands executed by clients

Parameters:
number – of commands to return
Returns:
list of command and clients

command_inout

any command_inout(in string command, in any argin)
raises(DevFailed);

execute a command on a device synchronously with no input parameter and one one output parameter

Parameters:
command – ascii string e.g. "On"
argin – command input parameter e.g. float
Returns:
command result.

command_list_query

DevCmdInfoList command_list_query()
raises(DevFailed);

query device to see what commands it supports

Returns:
list of commands and their types

command_query

DevCmdInfo command_query(in string command)
raises(DevFailed);

query device to see command argument

Parameters:
command – name
Returns:
command and its types

get_attribute_config

AttributeConfigList get_attribute_config(in DevVarStringArray names)
raises(DevFailed);

read the configuration for a variable list of attributes from a device

Parameters:
name – list of attribute names to read
Returns:
list of attribute configurations read

info

DevInfo info()
raises(DevFailed);

return general information about object e.g. class, type, ...

Returns:
device info

ping

void ping()
raises(DevFailed);

ping a device to see if it alive

read_attributes

AttributeValueList read_attributes(in DevVarStringArray names)
raises(DevFailed);

read a variable list of attributes from a device

Parameters:
name – list of attribute names to read
Returns:
list of attribute values read

set_attribute_config

void set_attribute_config(in AttributeConfigList new_conf)
raises(DevFailed);

set the configuration for a variable list of attributes from the device

Parameters:
new_conf – list of attribute configuration to be set

write_attributes

void write_attributes(in AttributeValueList values)
raises(DevFailed);

write a variable list of attributes to a device

Parameters:
values – list of attribute values to write

B.7 Interface Tango::Device_2

interface Device_2 inherits from Tango::Device

The updated Tango device interface. It inherits from Tango::Device and therefore supports all attribute/operation defined in the Tango::Device interface. Two CORBA operations have been modified to support more parameters (command_inout_2 and read_attribute_2). Three CORBA operations now retrun a different data type (command_list_query_2, command_query_2 and get_attribute_config)

B.7.1 Operations

command_inout_2

any command_inout_2(in string command, in any argin, in DevSource source)
raises(DevFailed);

execute a command on a device synchronously with no input parameter and one one output parameter

Parameters:
command – ascii string e.g. "On"
argin – command input parameter
source – data source
Returns:
command result.

command_inout_history_2

DevCmdHistoryList command_inout_history_2(in string command, in long n)
raises(DevFailed);

Get command result history from polling buffer. Obviously, the command must be polled.

Parameters:
command – ascii string e.g. "On"
n – record number
Returns:
list of command result (or exception parameters if the command failed).

command_list_query_2

DevCmdInfoList_2 command_list_query_2()
raises(DevFailed);

query device to see what commands it supports

Returns:
list of commands and their types

command_query_2

DevCmdInfo_2 command_query_2(in string command)
raises(DevFailed);

query device to see command argument

Parameters:
command – name
Returns:
command and its types

get_attribute_config_2

AttributeConfigList_2 get_attribute_config_2(in DevVarStringArray names)
raises(DevFailed);

read the configuration for a variable list of attributes from a device

Parameters:
name – list of attribute names to read
Returns:
list of attribute configurations read

read_attributes_2

AttributeValueList read_attributes_2(in DevVarStringArray names, in DevSource source)
raises(DevFailed)

read a variable list of attributes from a device

Parameters:
name – list of attribute names to read
Returns:
list of attribute values read

read_attribute_history_2

DevAttrHistoryList read_attributes_history_2(in string name, in long n)
raises(DevFailed)

Get attribute value history from polling buffer. Obviously, the attribute must be polled.

Parameters:
name – Attribute name to read history
n – Record number
Returns:
list of attribute value (or exception parameters if the attribute failed).

B.8 Interface Tango::Device_3

interface Device_3 inherits from Tango::Device_2

The updated Tango device interface for Tango release 5. It inherits from Tango::Device_2 and therefore supports all attribute/operation defined in the Tango::Device_2 interface. Six CORBA operations now return a different data type (read_attributes_3, write_attributes_3, read_attribute_history_3, info_3, get_attribute_config_3 and set_attribute_config_3)

B.8.1 Operations

read_attributes_3
AttributeValueList_3 read_attributes_3(in DevVarStringArray names, in DevSource source)
raises(DevFailed);

read a variable list of attributes from a device

Parameters:
name – list of attribute names to read
source – data source
Returns:
list of attribute values read

write_attributes_3

void write_attributes_3(in AttributeValueList values)
raises(DevFailed, MultiDevFailed);

write a variable list of attributes to a device

Parameters:
values – list of attribute values to write

read_attribute_history_3

DevAttrHistoryList_3 read_attributes_history_3(in string name, in long n)
raises(DevFailed)

Get attribute value history from polling buffer. Obviously, the attribute must be polled.

Parameters:
name – Attribute name to read history
n – Record number
Returns:
list of attribute value (or exception parameters if the attribute failed).

info_3

DevInfo_3 info()
raises(DevFailed);

return general information about object e.g. class, type, ...

Returns:
device info

get_attribute_config_3

AttributeConfigList_3 get_attribute_config_3(in DevVarStringArray names)
raises(DevFailed);

read the configuration for a variable list of attributes from a device

Parameters:
name – list of attribute names to read
Returns:
list of attribute configurations read

set_attribute_config_3

void set_attribute_config_3(in AttributeConfigList_3 new_conf)
raises(DevFailed);

set the configuration for a variable list of attributes from the device

Parameters:
new_conf – list of attribute configuration to be set

B.9 Interface Tango::Device_4

interface Device_4 inherits from Tango::Device_3

The updated Tango device interface for Tango release 7. It inherits from Tango::Device_3 and therefore supports all attribute/operation defined in the Tango::Device_3 interface.

B.9.1 Operations

read_attributes_4
AttributeValueList_4 read_attributes_4(in DevVarStringArray names, in DevSource source,in ClntIdent cl_ident)
raises(DevFailed);

read a variable list of attributes from a device

Parameters:
name – list of attribute names to read
source – data source
cl_ident – client identificator
Returns:
list of attribute values read

write_attributes_4

void write_attributes_3(in AttributeValueList_4 values, in ClniIdent cl_ident)
raises(DevFailed, MultiDevFailed);

write a variable list of attributes to a device

Parameters:
values – list of attribute values to write
cl_ident – client identificator

command_inout_4

any command_inout_4(in string command, in any argin, in DevSource source, In ClntIdent cl_ident)
raises(DevFailed);

Execute a command on a device synchronously with one input parameter and one one output parameter

Parameters:
command – ascii string e.g. "On"
argin – command input parameter
source – data source
cl_ident – client identificator
Returns:
command result

read_attribute_history_4

DevAttrHistory_4 read_attributes_history_4(in string name, in long n)
raises(DevFailed)

Get attribute value history from polling buffer. Obviously, the attribute must be polled.

Parameters:
name – Attribute name to read history
n – Record number
Returns:
Attribute value (or exception parameters if the attribute failed) coded in a structure.

command_inout_history_4

DevCmdHistory_4 command_inout_history_4(in string command, in long n)
raises(DevFailed);

Get command value history from polling buffer. Obviously, the command must be polled.

Parameters:
name – Command name to read history
n – Record number
Returns:
Command value (or exception paramteters) coded in a structure

write_read_attribute_4

AttributeValueList_4 write_read_attribute_4(in AttributeValueList_4 values, in ClntIdent cl_ident)
raises(DevFailed,MultiDevFailed);

Write then read a variable list of attributes from a device

Parameters:
values – list of attribute values to write
cl_ident – client identificator
Returns:
list of attribute values read

set_attribute_config_4

void set_attribute_config_4(in AttributeConfigList_3 new_conf, in ClntIdent cl_ident)
raises(DevFailed);

set the configuration for a variable list of attributes from the device

Parameters:
new_conf – list of attribute configuration to be set
cl_ident – client identificator

Appendix C

Tango object naming (device, attribute and property)

C.1 Device name

A Tango device name is a three fields name. The field separator is the **/** character. The first field is named **domain**, the second field is named **family** and the last field is named **member**.A tango device name looks like

domain/family/member

It is a hierarchical notation. The member specifies which element within a family. The family specifies which kind of equipment within a domain. The domain groups devices related to which part of the accelerator/experiment they belongs to. At ESRF, some of the machine control system domain name are SR for the storage ring, TL1 for the transfer line 1 or SY for the synchrotron booster. For experiment, ID11 is the domain name for all devices belonging to the experiment behind insertion device 11. Here are some examples of Tango device name used at the ESRF :

- **sr/d-ct/1** : The current transformer. The domain part is sr for storage ring. The family part is d-ct for diagnostic/current transformer and the member part is 1
- **fe/v-pen/id11-1** : A Penning gauge. The domain part is fe for front-end. The family part is v-pen for vacuum/penning and the member name is id11-1 to specify that this is the first gauge on the front-end part after the insertion device 11

C.2 Full object name

The device name as described above is not enough to cover all Tango usage like device server without database or device access for multi control system. With the naming schema, we must also be able to name attribute and property. Therefore, the full naming schema is

[protocol://][host:port/]device_name[/attribute][->property][#dbase=xx]

The protocol, host, port, attribute, property and dbase fields are optional. The meaning of these fields are :

protocol : Specifies which protocol is used (Tango or Taco). Tango is the default

dbase=xx : The supported value for xx is *yes* and *no*. This field is used to specify that the device is a device served by a device server started with or without database usage. The default value is *dbase=yes* host:port : This field has different meaning according to the dbase value. If *dbase=yes* (the default), the host is the host where the control system database server is running and port is the database server port. It has a higher priority than the value defined by the TANGO_HOST environment variable. If *dbase=no*, host is the host name where the device server process serving the device is running and port is the device server process port.

attribute : The attribute name

property : The property name

The host:port and dbase=xx fields are necessary only when creating the DeviceProxy object used to remotely access the device. The -> characters are used to specify a property name.

C.2.1 Some examples

C.2.1.1 Full device name examples

- **gizmo:20000/sr/d-ct/1** : Device sr/d-ct/1 running in a specified control system with the database server running on a host called gizmo and using the port number 20000. The TANGO_HOST environment variable will not be used.
- **tango://freak:2345/id11/rv/1#dbase=no** : Device served by a device server started without database. The server is running on a host called freak and use port number 2345. //freak:2345/id11/rv/1#dbase=no is also possible for the same device.
- **Taco://sy/ps-ki/1** : Taco device sy/ps-ki/1

C.2.1.2 Attribute name examples

- **id11/mot/1/Position** : Attribute position for device id11/mot/1
- **sr/d-ct/1/Lifetime** : Attribute lifetime for Tango device sr/d-ct/1

C.2.1.3 Attribute property name

- **id11/rv/1/temp->label** : Property label for attribute temp for device id11/rv/1.
- **sr/d-ct/1/Lifetime->unit** : The unit property for the Lifetime attribute of the sr/d-ct/1 device

C.2.1.4 Device property name

- **sr/d-ct/1->address** : the address property for device sr/d-ct/1

C.2.1.5 Class property name

- **Starter->doc_url** : The doc_url property for a class called Starter

C.3 Device and attribute name alias

Within Tango, each device or attribute can have an alias name defined in the database. Every time a device or an attribute name is requested by the API's, it is possible to use the alias. The alias is simply an open string stored in the database. The rule of the alias is to give device or attribute name a name more natural from the physicist point of view. Let's imagine that for experiment, the sample position is described by angles called teta and psi in physics book. It is more natural for physicist when they move the motor related to sample position to use *teta* and *psi* rather device name like *idxx/mot/1* or *idxx/mot/2*. An attribute alias is a synonym for the four fields used to name an attribute. For instance, the attribute *Current* of a power-supply device called *sr/ps/dipole* could have an alias *DipoleCurrent*. This alias can be used when creating an instance of a AttributeProxy class instead of the full attribute name which is *sr/ps/dipole/Current*. Device alias name are uniq within a Tango control system. Attribute alias name are also uniq within a Tango control system.

C.4 Reserved words and characters, limitations

From the naming schema described above, the reserved characters are **:**,**#**,**/** and the reserved string is : **->**. On top of that, the dbt_update tool (tool to fulfill database from the content of a file) reserved the **device** word

The device name, its domain, member and family fields and its alias are stored in the Tango database. The default maximum size for these items are :

Item	max length
device name	255
domain field	85
family field	85
member field	85
device alias name	255

The device name, the command name, the attribute name, the property name, the device alias name and the device server name are **case insensitive**.

Appendix D

Starting a Tango control system

D.1 Without database

When used without database, there is no additional process to start. Simply starts device server using the -nodb option (and eventually the -dlist option) on specific port. See 9.9 to find informations on how to start/write Tango device server not using the database.

D.2 With database

Starting the Tango control system simply means starting its database device server on a well defined host using a well defined port. Use the host name and the port number to build the TANGO_HOST environment variable. See 8.6.2 to find how starting a device server on a specific host. Obviously, the underlying database software (MySQL) must be started before the Tango database device server. The Tango database server connects to MySQL using a default logging name set to "root". You can change this behaviour with the MYSQL_USER and MYSQL_PASSWORD environment variables. Define them before starting the database server.

If you are using the Tango administration graphical tool called **Astor**, you also need to start a specific Tango device server called **Starter** on each host where Tango device server(s) are running. See [19] for Astor documentation. This starter device server is able to start even before the Tango database device server is started. In this case, it will enter a loop in which it periodically tries to access the Tango database device. The loop exits and the server starts only if the database device access succeed.

D.3 With database and event

D.3.1 For Tango releases lower than 8

On top of what is described in the previous chapter, using event means using CORBA Notification service. Start one Notification Service daemon on each host where device server(s) used via events are running. The Notification Service daemon event channel factory IOR has to be registered in the Tango database. This is done with the **notifd2db** command. The notification daemon is a process with a high thread number. By default, Unix like operating systems reserve a big amount of memory for each thread stack (8 MByte for Linux/Ubuntu, 10 MByte for Linux/RedHat 4). If your process has several hundreds of threads, this could generate a too high memory requirement on virtual memory and even exceed the maximun allowed memory per process (3 GBytes on Linux for 32 bits computer). The notification service daemon works very well with a value of only 2 Mybtes for thread stack. The Unix command line "ulimit -s 2048" asks the operating system to give 2 Mbytes for each thread stack. Example of starting and registering a Notification Service daemon on a UNIX like operating system

```
1  ulimit -s 2048
2  notifd -n -DDeadFilterInterval=300 &
3  notifd2db
```

The Notification Service daemon is started at line 2. Its "-DDeadFilterInterval" option is used to specify some internal cleaning of dead objects within the notification service. The "-n" option is used to disable the use of the CORBA Naming Service for registering the default event channel factory. The registration of the Notification Service daemon in the Tango database is done at line 2.

It differs on a Windows computer

```
notifd -n -DDeadFilterInterval=300 -DFactoryIORFileName=C:\Temp\evfact.ior &
notifd2db C:\Temp\evfact.ior
```

D.3.2 For release 8 and above

A new event system has been implemented starting with Tango release 8. With this new event system, the CORBA Notification service is not needed any more. This means that **as soon** as all Tango device server processes running on a host and all clients using events from their devices used Tango 8, it is not required to start any process other than the device servers and the clients. You can forget the previous sub-chapter!

D.4 With file used as database

When used with database on file, there is no additional process to start. Simply starts device server using the -file option specifying file name port. See 9.8 to find informations on how to start Tango device server using database on file.

D.5 With file used as database and event

D.5.1 For Tango releases lower than 8

Using event means using CORBA Notification service. Start one Notification Service daemon on the host where device server(s) using events are running. The Notification Service daemon event channel factory IOR has to be registered in the file(s) use as database. This is done with the **notifd2db** command. Example of starting and registering a Notification Service daemon on a UNIX like operating system

```
1  notifd -n -DDeadFilterInterval=300 &
2  notifd2db -o /var/myfile.res
```

The Notification Service daemon is started at line 1. Its "-n" option is used to disable the use of the CORBA Naming Service for registering the default event channel factory. The registration of the Notification Service daemon in the file used as database is done at line 2 with its **-o** command line option.

It differs on a Windows computer because the name of the file used by the CORBA notification service to store its channel factory IOR must be specified using its -D command line option. This file name has also to be passed to the notifd2db command.

```
notifd -n -DDeadFilterInterval=300 -DFactoryIORFileName=C:\Temp\evfact.ior &
notifd2db C:\Temp\evfact.ior -o C:\Temp\myfile.res
```

D.5.2 For release 8 and above

A new event system has been implemented starting with Tango release 8. With this new event system, the CORBA Notification service is not needed any more. This means that **as soon** as all clients using events from devices embedded in the device server use Tango 8, it is not required to start any process other than the device server and its clients.

D.6 With the controlled access

Using the Tango controlled access means starting a specific device server called TangoAccessControl. By default, this server has to be started with the instance name set to "1" and its device name is "sys/access_control/1". The command line to start this device server is:

```
TangoAccessControl 1
```

This server connects to MySQL using a default logging name set to "root". You can change this behaviour with the MYSQL_USER and MYSQL_PASSWORD environment variables. Define them before starting the controlled access device server. This server also uses the MYSQL_HOST environment variable if you need to connect it to some MySQL server running on another host. The syntax of this environment varaible is "host:port". Port is optional and if it is not defined, the MySQL default port is used (3306). If it is not defined at all, a connection to the localhost is made. This controlled access system uses the Tango database to retrieve user rights and it is not possible to run it in a Tango control system running without database.

Appendix E

The notifd2db utility

E.1 The notifd2db utility usage (For Tango releases lower than 8)

The notifd2db utility is used to pass to Tango the necessary information for the Tango servers or clients to build connection with the CORBA notification service. Its usage is:

notifd2db [notifd2db_IOR_file] [host] [-o Device_server_database_file_name] [-h]

The [notifd2db_IOR_file] parameter is used to specify the file name used by the notification service to store its main IOR. This parameter is not mandatoty. Its default value is /tmp/rdfact.ior. The [host] parameter is ued to specify on which host the notification service should be exported. The default value is the host on which the command is run. The [-o Device_server_database_file_name] is used in case of event and device server started with the file as database (the -file device server command line option). The file name used here must be the file name used by the device server in its -file option. The [-h] option is just to display an help message. Notifd2db utility usage example:

notifd2db

to register notification service on the current host using the default notifictaion service IOR file name.

notifd C:\Temp\nd.ior

to register a notification service with IOR file named C:\Temp\nd.ior.

notifd -o /var/my_ds_file.res

to register notification service in the /var/my_ds_file.res file used by a device server started with the device server -file command line option.

Appendix F

The property file syntax

F.1 Property file usage

A property file is a file where you store all the property(ies) related to device(s) belonging to a specific device server process. In this file, one can find:

- Which device(s) has to be created for each Tango class embedded in the device server process
- Device(s) properties
- Device(s) attribute properties

This type of file is not required by a Tango control system. These informations are stored in the Tango database and having them also in a file could generate some data duplication issues. Nevertheless, in some cases, it could very very helpful to generate this type of file. These cases are:

1. If you want to run a device server process on a host which does not have access to the Tango control system database. In such a case, the user can generate the file from the database content and run the device server process using this file as database (-file option of device server process)
2. In case of massive property changes where no tool will be more adapted than your favorite text editor. In such a case, the user can generate a file from the database content, change/add/modify file contents using his favorite tool and then reload file content into the database.

Jive[21] is the tool provided to generate and load a property file. To generate a device server process properties file, select your device server process in the "Server" tab, right click and select "Save Server Data". A file selection window pops up allowing you to choose your file name and path. To reload a file in the Tango database, click on "File" then "Load Property File".

F.2 Property file syntax

```
#---------------------------------------------------------
# SERVER TimeoutTest/manu, TimeoutTest device declaration
#---------------------------------------------------------

TimeoutTest/manu/DEVICE/TimeoutTest: "et/to/01",\
                                     "et/to/02",\
                                     "et/to/03"


```

```
10 # --- et/to/01 properties
11
12 et/to/01->StringProp: Property
13 et/to/01->ArrayProp: 1,\
14                      2,\
15                      3
16 et/to/01->attr_min_poll_period: TheAttr,\
17                                 1000
18 et/to/01->AnotherStringProp: "A long string"
19 et/to/01->ArrayStringProp: "the first prop",\
20                            "the second prop"
21
22 # --- et/to/01 attribute properties
23
24 et/to/01/TheAttr->display_unit: 1.0
25 et/to/01/TheAttr->event_period: 1000
26 et/to/01/TheAttr->format: %4d
27 et/to/01/TheAttr->min_alarm: -2.0
28 et/to/01/TheAttr->min_value: -5.0
29 et/to/01/TheAttr->standard_unit: 1.0
30 et/to/01/TheAttr->__value: 111
31 et/to/01/BooAttr->event_period: 1000doc_url
32 et/to/01/TestAttr->display_unit: 1.0
33 et/to/01/TestAttr->event_period: 1000
34 et/to/01/TestAttr->format: %4d
35 et/to/01/TestAttr->standard_unit: 1.0
36 et/to/01/DbAttr->abs_change: 1.1
37 et/to/01/DbAttr->event_period: 1000
38
39 CLASS/TimeoutTest->InheritedFrom:   Device_4Impl
40 CLASS/TimeoutTest->doc_url:   "http://www.esrf.fr/some/path"
```

Line 1 - 3: Comments. Comment starts with the '#' character

Line 4: Blank line

Line 5 - 7: Devices definition. "DEVICE" is the keyword to declare a device(s) definition sequence. The general syntax is:

<DS name>/<inst name>/DEVICE/<Class name>: dev1,dev2,dev3

Device(s) name can follow on next line if the last line character is '\' (see line 5,6). The '"' characters around device name are generated by the Jive tool and are not mandatory.

Line 12: Device property definition. The general device property syntax is

<device name>**->**<property name>: <property value>

In case of array, the array element delimiter is the character ','. Array definition can be splitted on several lines if the last line character is '\'. Allowed characters after the ':' delimiter are space, tabulation or nothing.

Line 13 - 15 and 16 - 17: Device property (array)

Line 18: A device string property with special characters (spaces). The '"' character is used to delimit the string

Line 24 - 37: Device attribute property definition. The general device attribute property syntax is

<device name>/<attribute name>**->**<property name>: <property value>

Allowed characters after the ':' delimiter are space, tabulation or nothing.

Line 39 - 40: Class property definition. The general class property syntax is

CLASS/<class name>**->**<property name>: <property value>

"CLASS" is the keyword to declare a class property definition. Allowed characters after the ':' delimiter are space, tabulation or nothing. On line 40, the '"' characters around the property value are mandatory due to the '/' character contains in the property value.

List of pictures

Bibliography

[1] OMG home page

[2] "Advanced CORBA programming with C++" by M.Henning and S.Vinosky (Addison-Wesley 1999)

[3] TANGO home page

[4] ALBA home page

[5] Soleil home page

[6] MySQL home page

[7] "MySQL and mSQL" by Randy Jay Yarger, George Reese and Tim King (O'Reilly 1999)

[8] Tango classes on-line documentation

[9] "C++ programming language" third edition by Stroustrup (Addison-Wesley)

[10] "Design Patterns" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley 1995)

[11] omniORB home page

[12] The Common Object Request Broker: Architecture and Specification Revision 2.3 available from OMG home page

[13] Java Pro - June 1999 : Plugging memory leak by Tony Leung

[14] CVS WEB page - http://www.cyclic.com

[15] POGO home page

[16] JacORB home page

[17] Tango ATK reference on-line documentation

[18] The Notification Service specification available from OMG home page - http://www.omg.org

[19] ASTOR home page

[20] Elettra home page

[21] JIVE home page

[22] Tango ATK Tutorials

[23] ZMQ home page

[24] Tango class development reference documentation

Index