The TANGO Control System Manual
Version 9.2

The TANGO Team

June 24, 2016

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
- Chapter 5 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 6
- Chapter 7 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 [**?**] to get all information about Soleil. In December 2003, Elettra joins the club. Elettra is an Italian synchrotron radiation facility located in Trieste. See [**?**] 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 [**?**] 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[**?**] 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 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 !

2.2 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 [**?**] 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**.

Since release 9, a Tango device also supports pipe. This is an advanced feature reserved for some specific cases. Therefore, there is no device pipe example in this "Getting started" chapter.

2.2.1 The commands and attributes code

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*

2.2.1.1 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.

2.2.1.2 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 :

```
1  Tango::DevVarLongArray *DocDs::dev_array(const Tango::DevVarLongArray *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::DevVarLongArray  *argout  = new Tango::DevVarLongArray();
 9
10          DEBUG_STREAM << "DocDs::dev_array(): entering... !" << endl;
11
12          //      Add your own code to control device here
13
14          long argin_length = argin->length();
15          argout->length(argin_length);
16          for (int i = 0;i < argin_length;i++)
17                  (*argout)[i] = (*argin)[i] * 2;
18
19          return argout;
20  }
```

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 **??** for all the details.

2.2.1.3 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 **??** for all the details.

2.2.1.4 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:

```
1  Tango::DevVarStringArray *DocDs::dev_str_array()
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::DevVarStringArray        *argout  = new Tango::DevVarStringA
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.

2.2.1.5 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:

```
1   Tango::DevVarDoubleStringArray *DocDs::dev_struct()
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::DevVarDoubleStringArray  *argout  = new Tango::DevVarDoubleS
9
10          DEBUG_STREAM << "DocDs::dev_struct(): entering... !" << endl;
11
12          //      Add your own code to control device here
13
14          argout->dvalue.length(3);
15          argout->dvalue[0] = 0.0;
16          argout->dvalue[1] = 11.11;
17          argout->dvalue[2] = 22.22;
18
19          argout->svalue.length(2);
20          argout->svalue[0] = CORBA::string_dup("Be Bop");
21          string str("Smurf");
22          argout->svalue[1] = CORBA::string_dup(str.c_str());
23
```

```
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.

2.2.1.6 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) :

```
1
2
3  protected :
4          //      Add your own data members here
5          //-----------------------------------------
6          Tango::DevString        attr_str_array[5];
7          Tango::DevLong          attr_rd;
8          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 :

```
1 void DocDs::read_attr_hardware(vector<long> &attr_list)
2 {
3     DEBUG_STREAM << "DocDs::read_attr_hardware(vector<long> &attr_list) entering.
4 // Add your own code here
5
6     string att_name;
7     for (long i = 0;i < attr_list.size();i++)
8     {
9         att_name = dev_attr->get_attr_by_ind(attr_list[i]).get_name();
10
11        if (att_name == "LongRdAttr")
12        {
13            attr_rd = 5;
14        }
15    }
16 }
17
```

```
18 void DocDs::read_LongRdAttr(Tango::Attribute &attr)
19 {
20     DEBUG_STREAM << "DocDs::read_LongRdAttr(Tango::Attribute &attr) entering...
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...
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..
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... "<<
43
44     attr_str_array[0] = const_cast<char *>("Rock");
45     attr_str_array[1] = const_cast<char *>("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* Tango function. There are several ways to code spectrum or image attribute of the DevString data type. A HowTo related to this topic is available on the Tango control system Web site. 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".

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 [**?**]). 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 [**?**]. The complete TANGO IDL file can be found in the TANGO web page[**?**] 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/Set data specific to each device belonging to a class via TANGO **pipes**
- 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 five 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 The TANGO pipes

Since release 9, in addition to commands and attributes, TANGO devices also support pipes.

In some cases, it is required to exchange data between client and device of varrying data type. This is for instance the case of data gathered during a scan on one experiment. Because the number of actuators and sensors involved in the scan may change from one scan to another, it is not possible to use a well defined data type. TANGO pipes have been designed for such cases. A TANGO pipe is basically a pipe dedicated to transfer data between client and device. A pipe has a set of two properties which are the pipe label and its description. A pipe can be read or read-write. A device can support a list of pipes. Clients can read one or more pipes from one or more devices. To read a TANGO pipe, the client uses the **read_pipe** operation. To write a TANGO pipe, a client uses the **write_pipe** operation. To write then read a TANGO pipe within the same network request, the client uses the **write_read_pipe** operation. To query a device for all the pipes it supports, a client uses the **get_pipe_config** operation. A client is also able to modify some of parameters defining a pipe with the **set_pipe_config** operation. These five operations are defined in the device CORBA interface.

In contrary of commands or attributes, a TANGO pipe does not have a pre-defined data type. Data transferred through pipes may be of any basic Tango data type (or array of) and this may change every time a pipe is read or written.

[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

3.3.4 Command, attributes or pipes ?

There are no strict rules concerning what should be returned as command result and what should be implemented as an attribute or as a pipe. 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. Finally pipe is well adapted when the kind and number of data exchanged between the client and the device change with time.

3.3.5 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
- 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 **??**. 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 **??**, the CORBA attributes are not allowed to raise exceptions whereas command (which are implemented using CORBA operations) can.

3.3.6 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

3.3.7 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.

3.3.8 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. It is not possible to poll a device pipe.

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 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[**?**] 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 [**?**]

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

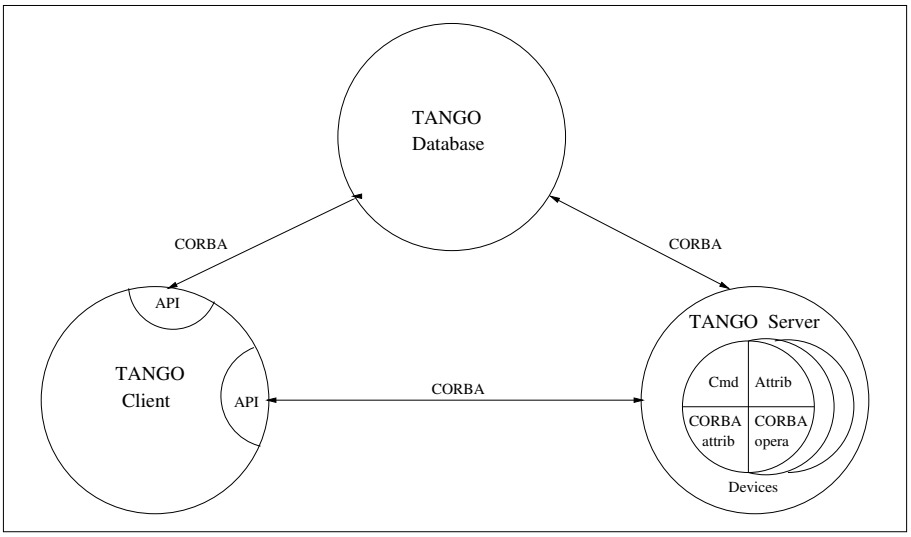
While it is true TANGO clients can be programmed using only the CORBA API, CORBA knows nothing about TANGO. This means client have to know all the details of retrieving IORs from the TANGO database,

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

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

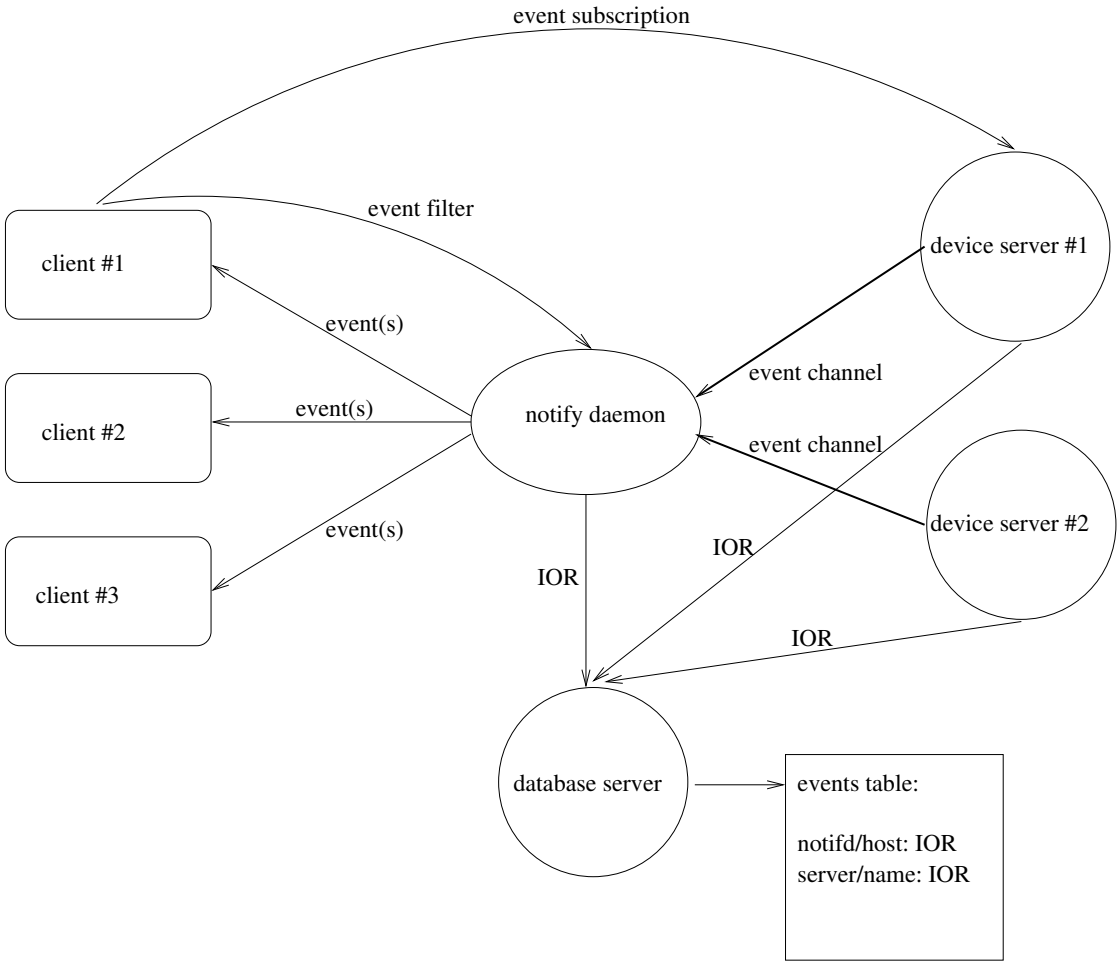
On top of the two communication model previously described, TANGO offers an "event system". The standard TANGO communication paradigm is a synchronou/asynchronous two-way call. In this paradigm the call is initiated by the client who contacts the server. The server handles the client's request and sends 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.

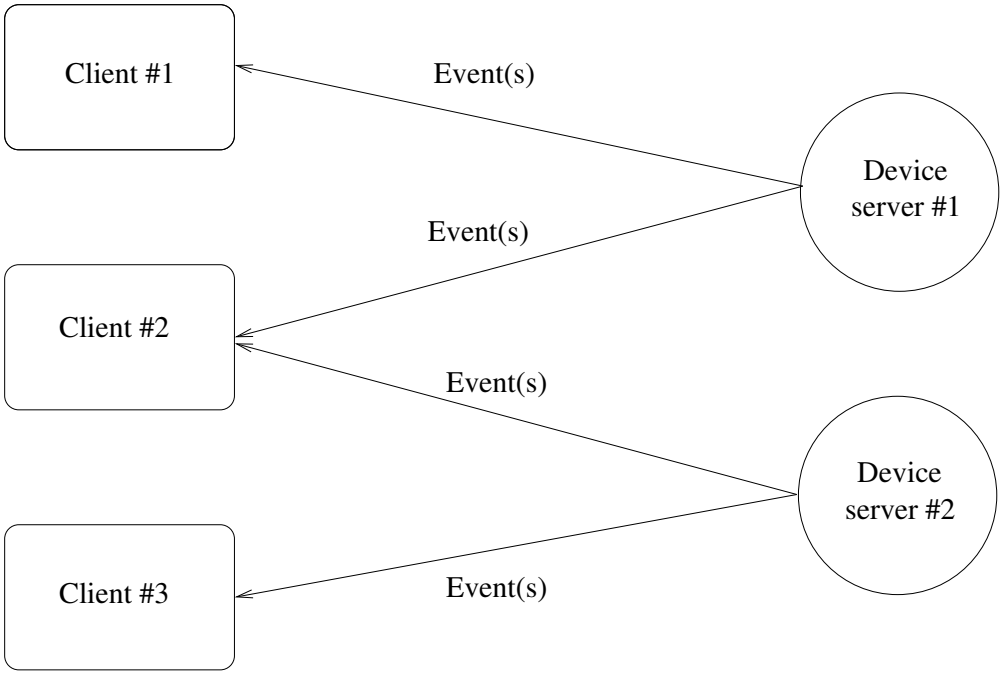
Schematic of TANGO Events system

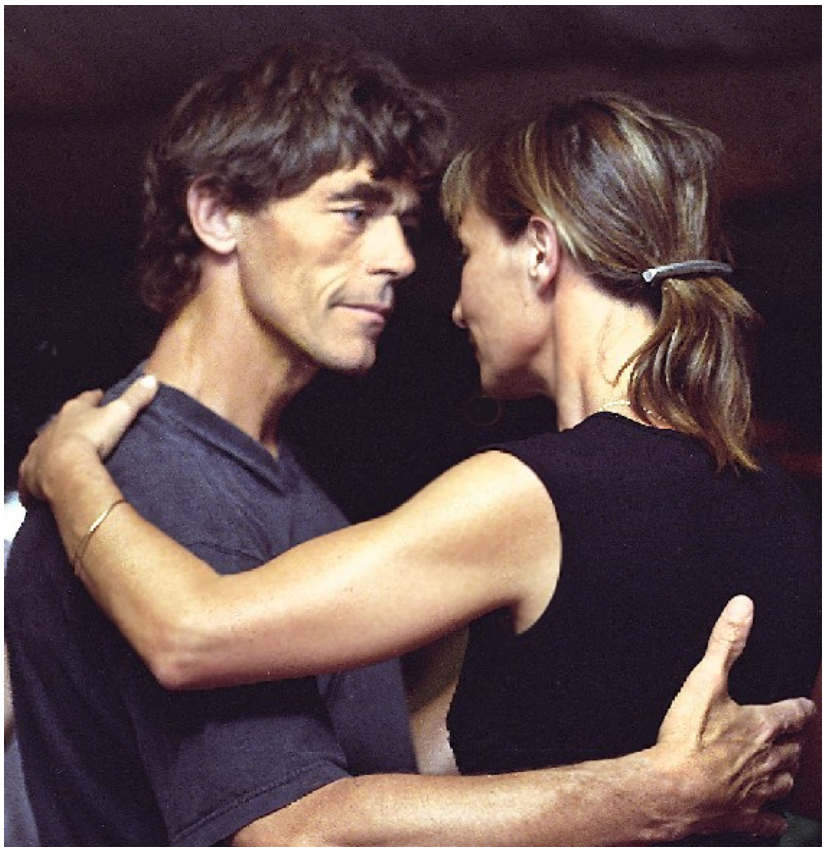


Starting with Tango 8, a new design of the event system has been implemented. This new design is based on the ZMQ library. ZMQ is a library allowing users to create communicating system. It implements several well known communication pattern including the Publish/Subscribe pattern which is the basic 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. Starting with Tango 8.1, the event propagation between devices and clients could be done using a multi-casting protocol. The aim of this is to reduce both the network bandwidth use and the CPU consumption on the device server side. See chapter on Advanced Features to get all the details on this feature.

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





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 has 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. Reference documentation is available as Web pages in the Tango Web site

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, DeviceAttribute** or **DevicePipe** 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
DevEnum	enumeration (only for attribute / See chapter on advanced features)
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 **??** 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). Device pipe supports only the synchronous model. 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(s)
- *write_read_attribute()* or *write_read_attributes()* to write then read Tango device attribute(s)
- *read_pipe()* to read a Tango device pipe
- *write_pipe()* to write a Tango device pipe
- *write_read_pipe()* to write then read Tango device pipe

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. For pipes, data are send/receive to/from device pipe using the *DevicePipe* and *DevicePipeBlob* classes.

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 eight 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()*).

7. **device interface change** - This event is sent when the device interface changes. Using Tango, it is possible to dynamically add/remove attribute/command to a device. This event is the way to inform client(s) that attribute/command has been added/removed from a device. Note that this type of event is attached to a device and not to one attribute (like all other event types). This event is triggered in the following case :

   (a) A dynamic attribute or command is added or removed. The event is sent after a small delay (50 mS) in order to eliminate the risk of events storm in case several attributes/commands are added/removed in a loop

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

(b) At the end of admin device RestartServer or DevRestart command

(c) After a re-connection due to a device server restart. Because the device interface is not memorized, the event is sent even if it is highly possible that the device interface has not changed. A flag in the data propagated with the event inform listening applications that the device interface change is not guaranteed.

(d) At event re-connection time. This case is similar to the previous one (device interface change not guaranteed)

8. **pipe** - This is the kind of event which has to be used when the user want to push data through a pipe. This kind of event is only sent by the user code by using a specific method (*DeviceImpl::push_pipe_event()*). There is no way to ask the Tango kernel to automatically push this kind of event.

The first three above events are automatically generated by the TANGO library or fired by the user code. Events number 4 and 7 are only automatically sent by the library and events 5, 6 and 8 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
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 [**?**]. 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 *);
   virtual push_event(Tango::DevIntrChangeEventData *);
   virtual push_event(Tango::PipeEventData *);
}
```

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;
}
```

DataReadyEventData is defined as follows :

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

DevIntrChangeEventData is defined as follows :

```
class DevIntrChangeEventData
{
   DeviceProxy           device;
   string                event;
   string                device_name;
   CommandInfoList       cmd_list;
   AttributeInfoListEx   att_list;
   bool                  dev_started;
   bool                  err;
   DevErrorList          errors;
}
```

and PipeEventData is defined as follows :

```
class PipeEventData
{
   DeviceProxy       *device;
   string            pipe_name;
   string            event;
   DevicePipe        *pipe_value;
   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);
```