Tango 9 developments

September 17, 2012

1 Introduction

Following the two previous Tango executive committee meetings (May 2011 at Elettra and April 2012 at MaxIV), the list of new main features for Tango 9 is:

1. Tango pipe(s)
2. Enumeration as attribute data type
3. Forwarded attribute

Point 1 (Tango pipe) is the evolution of what was previously named "Structure as attribute data type".
This paper is a proposal of how these features could be implemented from the user point of view. Nevertheless, the work is not done yet and it is highly possible that the real implementation will not be exactly what is reported in this paper but the general idea is there.

Tango 9 will also implement (at last) some of Feature Request registered in the Tango-cs SourceForge project. From the user point of view, the most noticeable one is the introduction on the server side of a MyTangoClass::write_attr_hardware() method.

All the code snippets given in this paper are pseudo-code and are not intended to be compiled as is.

2 Tango pipe(s)

2.1 Motivation

Since several Tango meetings, we are talking of structure supported as attribute data type. It appears that the static definition of a structure prevents some key usage of this new feature. The requirement is to be able to send/receive a set of data to/from a Tango device but with variable data type. Classical use case is the camera interfaces on synchrotron beam-line where you want to transfer images coded in different data format (8, 16 or 32 bits per pixel). For beam line X ray position monitor, it is also convenient to be able to transfer the data coming from the hardware sometimes as float numbers, sometimes as double numbers. The case of scan data is also to be considered. You need to transfer data from a variable set of actuators (from different data type) and a variable set of sensors with also a different data type set. Variable data type does not fit into the Tango attribute model. Therefore, the proposal is to add new entities to Tango devices: Tango device pipe(s).

2.2 Definition

On top of commands and attributes, a Tango device can also have **pipes**. A Tango pipe is a third way to exchange data between a Tango client and a Tango device. The pipe transports a **blob** of data. Each blob is a set of **data elements**. Each data element has:

1. A name
2. Is a Tango basic type (or array thereof)

Using a pipe, you can send and/or receive data to/from a Tango device. From the device point of view, a Tango pipe

- received some data from the client. The definition of what the pipe receives is *static*. It's a fixed number of data element with fixed names and data type.
- send data to the client. The definition of what the pipe returns is *dynamic*. It's a variable set of data elements (variable names and data type)

Each pipe has:

1. A name unique for the device
2. A label and a description statically defined
3. A description which allows a client to retrieve the pipe input data definition (How many data elements, their names and data type) plus its label and description

Accessing a pipe is protected by a state machine (is_xxx_allowed() method). Compared to command or attributes, a pipe has a limited set of features:

- No polling (therefore no local history)
- No alarm
- No quality factor
- No change/periodic/archive event on a pipe
- Not accessible through a Tango group
- There is no dynamic pipe (in a sense like dynamic attribute)

For the TAC and device locking point of view, a device pipe is handled the same way command is. By default all device pipes are considered as WRITE except those defined as READ in the TAC configuration (definition at Tango device class level). Accessing a pipe is:

- Synchronous. You write data to a device pipe and you wait for its answer
- Event based. The application registers to the pipe. One application callback is executed when the device push data into the pipe

2.3 Server side usage

On the server side, in the Tango class (ie PowerSupply), there is one method for each device pipe (in the PowerSupply class). The name of this method is the pipe name. The library creates one instance of a Tango kernel **Pipe** class for each device pipe. The right instance is passed to the method when the client uses the pipe

```
class PowerSupply:public Tango::Device_5Impl
{
   ...
   void MyFirstPipe(Tango::Pipe &); // For the device pipe named MyFirstPipe
   void MySecondPipe(Tango::Pipe &); // For the device pipe named MySecondPipe

   bool is_MyFirstPipe_allowed(Tango::Pipe &); // For dev pipe named MyFirstPipe
   bool is_MySecondPipe_allowed(Tango::Pipe &); // For dev pipe named MySecondPipe
   ...
};
```

2.3.1 Sending data to the pipe

In our example, the Tango class send two kind of data using the same pipe. The first time the pipe is read by the client, the pipe blob is a set of 2 elements. Following client read return a pipe blob with a set of five elements. Data returned during the first read are arrays and therefore uses some Tango Pipe memory management issue (similar to what we already use for attribute). In the device instance, we have following data members

```
class PowerSupply:public Tango::Device_5Impl
{
   ...
   void MyFirstPipe(Tango::Pipe &);
   ...
   vector<short>      v_s;
   int                *i_ptr;
   short              s_val_1;
   int                i_val;
   double             db_val_1;
   double             db_val_2;
   short              s_val_2;
   ...
};
```

The implementation of the PowerSupply::MyFirstPipe method looks like:

```
void Powersupply::MyFirstPipe(Tango::Pipe &pi)
{
    vector<string> pipe_names;

    v_s.push_back(111);   // Init sent data
    v_s.push_back(222);   // Init sent data

    i_ptr = new int[2];   // Allocate memory for the sent buffer
    i_ptr[0] = 66;        // Init sent data
    i_ptr[1] = 77;        // Init sent data

    s_val_1 = 10;         // Init sent data
    i_val = 22;           // Init sent data
    db_val_1 = 2.2;       // Init sent data
    db_val_2 = 3.3;       // Init sent data
    s_val_2 = 33;         // Init sent data

    if (first_call == true)
    {
        pipe_names.push_back(string("Short_data_elt"));
        pipe_names.push_back(string("Int_allocated_elt"));

        pi.set_pipe_blob_elt_length(0,v_s.size());
        pi.set_pipe_blob_elt_length(1,2);
        pi.set_pipe_blob_elt_release(1,true);    // Ask Tango to free allocated mem

        pi.set_value(pipe_names,&(v_s[0]),i_ptr); // Send blob with 2 elts
    }
    else
    {
            pipe_names = {"one","two","three","four","five"};
            pi.set_value(pipe_names,&s_val_1,&i_val,&db_val_1,&db_val_2,&s_val_2);
        }
    }
```

User data are associated with the Tango pipe using the *Pipe::set_value()* method used 2 times in the example above. The same method is used whatever the number of data element in the blob which has to be sent using the pipe. The Tango kernel class Pipe has two methods allowing the user to send arrays as blob data element. Those methods are:

- *void Pipe::set_pipe_blob_elt_length(int elt_nb, int size)*
- *void Pipe::set_pipe_blob_elt_release(int elt_nb, bool release)*

If required, these methods has to be called before the blob is sent to the pipe.

2.3.2 Retrieving data from the pipe

To retrieve data from the pipe, the user method looks like

```
void Powersupply::MyFirstPipe(Tango::Pipe &pi)
{
    float fl_w;
    short sh_w;
    double db_w;

    vector<string>  pipe_names;

    pi.get_write_value(pipe_names,fl_w,sh_w,db_w);
    ....
}
```

User retrieve data sent to a pipe with the method *Pipe::get_write_value()* which is used once in our example to retrieve three data elements from the pipe: A float followed by a short and finally one double. This method accept any number of arguments on top of the reference to a string vector to retrieve the data elements name.

2.3.3 Pushing data to the pipe

In a similar way of the Pipe::set_value(), a *DeviceImpl::push_pipe_event()* method will be available.

```
DeviceImpl::push_pipe_event(string &pipe_name,vector<string> &blob_elt_names,T *val, Args ...ar
```

2.4 Client side usage

2.4.1 Using a pipe synchronously

Basic call Sending/Retrieving data to/from a pipe is done with the DeviceProxy method pipe_inout()

```
DevicePipe DeviceProxy::pipe_inout(string &pipe_name,DevicePipe &sent_blob);
```

To create a blob in order to send it to a pipe, use the Tango kernel **DevicePipe** class. Extracting data from a received blob is also possible using the Tango kernel DevicePipe class.

Creating a device pipe blob The following code snippet creates a pipe blob before send it to the device

```
double db_arr[10];
short s;
long l;

Tango::DevicePipe sent_dp;

sent_dp.insert_data_elt("FirstElt",db_arr,10);
sent_dp.insert_data_elt("SecondElt",s);
sent_dp.insert_data_elt("ThirdElt",l);

dev.pipe_inout("ThePipe",sent_dp);
```

Extracting data from a device pipe blob Within the Tango kernel DevicePipe class, there are methods to:

- Retrieve the number of data elements in the blob
- Retrieve data element name(s)
- Retrieve data type for one element in a blob (by element name or number)
- Retrieve data for one element in a blob (by element name or number)

The following is some code to extract data coming from a pipe

```
Tango::DevicePipe received_dp,sent_dp;

Tango::DeviceProxy dev(...);

sent_dp.insert_data_elt(...);
...

received_dp = dev.pipe_inout("ThePipe",sent_dp);

size_t nb_elt = received_dp.get_data_elt_nb();
vector<string> elt_names = received_dp.get_data_elt_names();

for (size_t loop = 0;loop < nb_elt;loop++)
{
    int elt_type = received_dp.get_data_elt_type(loop);

    short s_val;
    double db_val;

    switch (elt_type)
    {
    case DEV_SHORT:
       received_dp.extract_data_elt(loop,s_val);
       break;

    case DEV_DOUBLE:
       received_dp.extract_data_elt(loop,db_val);
       break;
```

```
        }
    }
```

Methods *DevicePipe::get_data_elt_type()* and *DevicePipe::extract_data_elt()* are also available with the blob data element name as argument instead of the blob data element number.

2.4.2 Using a device pipe through event

To receive data sent to a pipe using event, one application has to subscribe to the pipe event. This is done with the already existing DeviceProxy::subscribe_event() call.

```
    int DeviceProxy::subscribe_event(const string &p_name,EventType event,CallBack *cb);
```

To register to a pipe event, the new value **Tango::PIPE_EVENT** has to be used for the call event parameter. Event queue will also be supported for this kind of event attached to Tango device pipe(s). In the Tango::CallBack class, a new push_event() method will be added. This new method receives one instance of a PipeBlobData class.

```
    void Callback::push_event(PipeBlobData *event);

    struct Tango::PipeBlobData
    {
        Tango::DeviceProxy    *dev;
        string                pipe_name;
        Tango::DevicePipe     pipe_blob;
        bool                  err;
        Tango::DevErrorList   errors;
    };
```

Extraction of the pipe blob data element is similar to the synchronous call.

2.4.3 Getting device pipe description

The DeviceProxy class *DeviceProxy::pipe_list_query()* method allows one application to retrieve device pipe description.

```
    struct BlobDataEltInfo
    {
        string          name;
        int             data_type;
        vector<string>  extensions;
    };

    typedef BlobInfo vector<BlobDataEltInfo>;

    struct PipeInfo
    {
        string          pipe_name;
        string          description;
        string          label;
        BlobInfo        blob_info;
        string          in_blob_desc;
        string          out_blob_desc;
        vector<string>  extensions;
        DispLevel       disp_level;
```

```
};

typedef PipeInfoList vector<PipeInfo>;

PipeInfoList *DeviceProxy::pipe_list_query();
```

2.5 Implementation

Some more details on a possible implementation are available at Tango kernel wiki. It appears that in many ways the implementation is similar to what we already have for commands. Therefore, at a glance, implementing Tango device pipes is less work for the Tango kernel team than implementing structure as attribute data type.

3 Enumeration as attribute data type

3.1 Motivation

Many parameters in the hardware we have to control have a limited set of value with a string describing each of the possible value (Hardware mode, instrument scale,...). It's so common that several client layers have developed their own way to deal with this kind of parameters. Computing enumeration fits well with kind of data. Up to this proposal, it was not possible to define an enumeration as data type for a Tango attribute.

3.2 Definition

A Tango attribute enumeration has consecutive values **always starting with 0**. Enumeration labels are transferred within the attribute configuration. It's possible to change the enumeration labels (not the value) through the classical way of changing attribute configuration. Thus enumeration labels are re-definable at device or class level through the Tango database. It's not supported to add or delete new enumeration value (and label) at run time.

3.3 Server side usage

Let's suppose that we have a Fruit and Beverage enumerations

```
enum class _Beverage
{
    BEER = 0,
    VINE,
    WATER,
    NB_VALUE
};
typedef _Beverage Beverage;

enum class _Fruit
{
    BANANA = 0,
    APPLE,
    PEAR,
    NB_VALUE
};
typedef _Fruit Fruit;
```

In a Tango class named Meal , we have a beverage (data type Beverage) and fruit (data type Fruit) attribute. In the Meal class, we have data members for each attribute

```
class Meal:Tango::Device_5Impl
{
...
    Beverage bev;
    Fruit fr;
...
};
```

The Tango class developer code executed when a client read an attribute of enumeration data type look like

```
Meal::read_beverage(Tango::Attribute &att)
{
    bev = Beverage::BEER;
    att.set_value(&bev);
}

Meal::read_fruit(Tango::Attribute &att)
{
    fr = Fruit::PEAR;
    att.set_value(&fr);
}
```

The Tango class developer code executed when a client write the enumerated attributes look like

```
Meal::write_beverage(Tango::WAttribute &att)
{
    Beverage tmp_bev;
    att.get_write_value(tmp_bev);
    ...
}

Meal::write_fruit(Tango::WAttribute &att)
{
    Fruit tmp_fr;
    att.get_write_value(tmp_fr);
    ...
}
```

3.4 Client side usage

3.4.1 Reading an enumerated attribute

With enumeration compile time knowledge In case of a specific application, the developer is able to declare (with the help of the Tango class documentation) the enumeration in the application code. Then, reading the attribute is similar to any other attribute reading

```
enum class _Beverage
{
    BEER = 0,
    VINE,
    WATER,
    NB_VALUE
};
typedef _Beverage Beverage;
```

```
Tango::DeviceAttribute da = the_dinner.read_attribute("beverage");

Beverage bev;
da >> bev;
```

Note that in both server and client, the enumeration are C++ 11 "Strongly typed enum". This allows to have a NB_VALUE for each enumeration. This is used by the library to do some basic check on the enumeration value received from the device.

Generic case The Tango::DeviceAttribute class will be modified to add two methods to extract enumerated attribute as a number or as a string

- *DeviceAttribute::extract_enum_value(int &val_nb);*
- *DeviceAttribute::extract_enum_value(string &val_str);*

The application code looks like

```
DeviceAttribute da = the_dinner.read_attribute("beverage");

string enum_val;
da.extract_enum_value(enum_val);

cout << "My favorite beverage for dinner is " << enum_val << endl;
```

3.4.2 Writing an enumerated attribute

With enumeration compile time knowledge Once the enumeration is declared in the application code, the code is similar to all other attribute data type

```
Fruit f = Fruit::APPLE;

DeviceAttribute da;
da.set_name("fruit");

da << f;
the_dinner.write_attribute(da);
```

Generic case In a way similar to the reading case, the Tango::DeviceAttribute class will be modified to add two methods to insert enumerated attribute from a number or from a string

- *DeviceAttribute::insert_enum_value(int &val_nb);*
- *DeviceAttribute::insert_enum_value(string &val_str);*

The application code looks like

```
int enum_val = ...;

DeviceAttribute da;
da.set_name("fruit");

da.insert_enum_value(enum_val);
the_dinner.write_attribute(da);
```

3.5 Implementation

On the client side, it is based on a library maintained cache of enumerated attribute name - attribute configuration (including enumeration labels). This cache will be filled in at the first enumerated attribute read/write request and re-initialized after device re-connection. This cache also need to be updated if a user changes the enumeration labels (using set_attribute_config). This will be done on the server side by de-registering the Tango device from the CORBA POA and then re-registering it. This will force a client re-connection and then a cache update. On the server side, it is based on template method usage with a template definition of the data type used by the attribute (something already used in Tango since its release 8). Some more details on a possible implementation are available at Tango kernel wiki.

4 Forwarded attribute

4.1 Motivation

When writing Tango class for a high level equipment (Linac, RF transmitter,...), it is common to report on the high level device some attribute(s) coming from low level device(s). This force the high level class developer to code the attribute in a way it forwards all its read (and write) request to the low level device attribute. This also means that the high level device attribute has to be configured in a nearly similar way than the low level device attribute is configured. The aim of the new feature proposed here is to automate these actions in order to provide "forwarded attribute".

4.2 Definition

A forwarded attribute is one attribute which forwards:

- its read/write requests
- its configuration
- its polling
- its events subscription

to another attribute. This root attribute can be in the same Tango class, in the same device server process or elsewhere in the control system or even in another Tango control system. Because they share the same configuration (nearly), a forwarded attribute has the same data type, data format, R/W type than its root attribute. In the Tango class, **no code is required** for a forwarded attribute. Everything is done in the library. A forwarded attribute is either a static or a dynamic attribute.

4.2.1 Forwarded attribute association

There are two ways to define the association between the forwarded and the root attribute:

- Using a device property (something like local_att_name: root_att_name) when the forwarded attribute is a dynamic attribute. In this case the root attribute name has to be the FQAN (Fully Qualified Attribute Name).
- In code if it's possible to have at compile time knowledge of the root attribute device name

One of the two ways has to be used. If the property is used, it has the highest priority.

4.3 Forwarded attribute configuration

- A subset of the attribute configuration stays local to the attribute (**label - description**). All the other parameters are forwarded to the root attribute.
- Because the get_attribute_config() and set_attribute_config() calls are forwarded to the root attribute, if a user change the root attribute config, the forwarded attribute configuration will also be seen modified. For attribute change event, see chapter on forwarded attribute events.
- A new field has to be added into the attribute configuration for all attribute:

  string root_attribute_name

  It's an empty string for classical attribute and it is set to the root attribute fully qualified name in case of forwarded attribute

4.4 Forwarded attribute polling

Like all other features, it is forwarded to the root attribute. Start polling the forwarded attributes means start polling the root attribute. The DS admin device (used for all polling related commands) knows that the attribute is a forwarded one and forwards the request to the root attribute. Same idea for polling status, the DS admin device requires polling status for the root attributes of all device forwarded attributes and change their name accordingly before returning the info to the caller.

4.5 Forwarded attribute events

Like attribute configuration and polling, everything is forwarded to the root attribute. Changing the event definition on the forwarded attribute means changing it in the root attribute. When a user register to an event from a forwarded attribute, the library subscribes to the event on the root attribute. The root attribute name is returned to the client part of the library by the DS admin device event subscription command. When the event is received, the library simply change the attribute name before calling the user callback.

5 MyTangoClass::write_attr_hardware()

5.1 Motivation/Definition

On the server side, the user method execution sequencing will be modified to add a "write_attr_hardware" method. This is allows a better management of hardware which support writing several of their parameters in one go. This also makes the Sequencing symetric between reading and writing attributes

5.2 Reminder on read_attributes sequencing

On the server side, when the user calls read_attributes() on a Tango device, user methods sequencing is

```
/CALL/ always_executed_hook()
/CALL/ read_attr_hardware()

/FOR/ Each attribute to be read
    /CALL/ is_xxx_allowed()
    /IF/ previous call returns true
```

```
        /CALL/ read_xxx()
    /ENDIF/
/ENDFOR/
```

5.3 New write_attributes sequencing

On the server side, when the user calls write_attributes() on a Tango device, user methods sequencing will be

```
/CALL/ always_executed_hook()

/FOR/ Each attribute to be written
   /CALL/ is_xxx_allowed()
   /IF/ previous call returns true
        /CALL/ write_xxx()
   /ENDIF/
/ENDFOR/

/CALL/ write_attr_hardware()
```

The precise declaration of the new method MyTangoClass::write_attr_hardware() is:

- *virtual void MyTangoClass::write_attr_hardware(vector<long> &attr_list)*

This is a virtual method and the kernel provides a default implementation doing nothing.

5.4 Error management

Two possible cases:

1. The user code throws a *Tango::DevFailed* exception during the execution of this method. In this case, the Tango kernel assumes that the call fails for all attributes involved
2. The user code throws a *Tango::NamedDevFailed* exception during the execution of this method. Now, only the attribute(s) reported by the exception will be considered by the kernel as faulty. The already existing *NamedDevFailed* class will be modified to be easily generated and thrown by the user code (in a way similar to DevFailed).