Tango training

client API - C++

Elettra Sincrotrone Trieste

Claudio Scafuri

claudio.scafuri@elettra.eu

outline

remarks

fundamental operations with examples

C++ client side API

Each Tango device is an instance of a ***DeviceProxy***

- Easy connection building between clients and devices
- Manage re-connection
- Hide some IDL call details
- Hide some memory management issues

```
Tango::DeviceProxy mydev("sys/tg_test/1");

Tango::DeviceProxy *ptrdev;
ptrdev = new DevicepProxy("sys/tg_test/1");
delete ptrdev;
```

Errors

Errors throw exceptions (clean, standard O.O. style)

Tango exceptions inherit from ***DevFailed*** class:

- one catch is enough for Tango errors
- 10 exceptions classes derived from DevFailed allow detailed analysis:

ConnectionFailed, CommunicationFailed, WrongNameSyntax,

NonDbDevice, WrongData, NonSupportedFeature,

AsynCall, AsynReplyNotArrived, EventSystemFailed,

NamedDevFailedList

see documentation

reading an attribute

- *DeviceProxy read_attribute* method
- class DeviceAttribute is used for the received data
- overloaded operators for extracting data

```
DeviceAttribute DeviceProxy::read_attribute(string&);
```

see example: reading01.cpp

Elettra Sincrotrone Trieste

reading an attribute

- DeviceAttribute contains **lots** of informations:
  - name of attribute
  - timestamp
  - dim_x, dim_y
  - type
  - quality factor
  - setpoint of a r_w variable IMPORTANT !!

see example: reading02.cpp

reading vector (spectrum)

- oveloaded operators for extracting data into:
- a DevVar*Array structure:

```
DevVarDoubleDouble *outdvarray;
da >> outdvarray;
```

  - data is consumed, user get ownership, must free data
- std::vector<> of element:

```
vector<double> outvect; da >> outvect;
```

  - easier to handle, data not consumed

see example: reading03.cpp , manual

writing an attribute

DeviceProxy write_*attribute* method

```
void DeviceProxy::write_attribute(DeviceAttribute&);
```

object of class **`DeviceAttribute`** used to name the attribute and send the data. Constructors and insertion operators are defined for all the Tango types.

see example: writing01.cpp

writing vector (spectrum)

DeviceAttribute must be filled with a std::vector<*> using overloaded operators (easier, preferred)

It is also possible to use a DevVar*Array structure.

In this case the data ownership will be transferred to the ORB

no need to call delete.

see example: writing02.cpp

Elettra Sincrotrone Trieste

executing commands

DeviceProxy command_inout method sends a command to the device

class DeviceData is used for sending and receiving command data to and from the device.

```
DeviceData DeviceProxy::command_inout(const char*, DeviceData&);
```

data is inserted / extracted in/from `DeviceData` by means of overloaded operators.

Pay attention to types!

see example: command01.cpp

Elettra Sincrotrone Trieste

asynch calls

reads, writes, and commands can be executed *asynchronously* by the client.

The call is split in two phases:

- 1) invocation
- 2) result reading

May be useful when the server response is slow, the client can do other things while the server is working.

We present examples of asynch calls for read_attribute only,

but the same concepts apply to write_attribute and command_inout.

asynch call models

The asynch calls follow three distinct models:

- polling model
- callback pull model
- callback push model

polling model

The client calls read_read_attribute_asynch(), actively polls the server to detect when the response arrives and eventualy extratcs the data.

```
long request_id = dev->read_attribute_asynch(attname);

....

DeviceAttribute* da;

da = dev->read_attribute_reply(request_id,wait_time_ms);
```

Simple model , easy to manage. If `wait_time_ms` is 0, the call blocks untill the response arrives.

The key for getting the reply is `request_id`

See example asynread01.cpp for more details and exception handling.

callback poll model

The client prepares a *callback* which will be invoked for extracting and reading the attribute value. The callback is a class derived from `Tango::CallBack`; as a minimum the virtual method `read_attr(AttrReadEvent*)` must be implemented. The `read_attr` method is in charge of extracting the attribute value and passing it to client.

```
ReadCallBack rd_short_cb;

dev->read_attribute_asynch(attname,rd_short_cb);

....

dev->get_asynch_replies(wait_time_ms);
```

When get_asynch_replies is called, if data is ready, ReadCallBack::read_attr() method is called.

More complicated data handling !

The clients mantains some control on the timing of the callaback execution.

See example asynread02.cpp for more details and exception handling.

We will use Tango::CallBack when dealing with events.

callback push model

The client prepares a *callback* which will be invoked for extracting and reading the attribute value. The callback is a class derived from `Tango::CallBack`; as a minimum the virtual method `read_attr(AttrReadEvent*)` must be implemented. The `read_attr` method is in charge of extracting the attribute value and passing it to client. The client must also set globally the callback model

```
ReadCallBack rd_short_cb;

ApiUtil::instance()->set_asynch_cb_sub_model(PUSH_CALLBACK);

dev->read_attribute_asynch(attname,rd_short_cb);
```

When data is ready, ReadCallBack::read_attr() method is called, in a completely asynchronous mode.

The client **has no control** on the timing of the callaback execution!

See example asynread03.cpp for more details and exception handling.

callback push model

Last example of asymch feature is the execution of a command.

In this case the CallBack class must implement the `cmd_ended()` method.

```
CmdCallBack cmd_short_cb;
ApiUtil::instance()->set_asynch_cb_sub_model(PUSH_CALLBACK);
dev->command_inout_asynch(attname,cmd_short_cb);
```

When data is ready, `CmdCallBack::cmd_ended()` method is called, in a completely asynchronous mode.

The client **has no control** on the timing of the callaback execution!

See example asyncmd01.cpp for more details and exception handling.

asynch pros/cons

pros:

- client is not bound by slow server response
- client can interact with several devices “in parallel”

cons:

- more complex logic flow
- time and order of arrival undetermined
- exception handling is duplicated
  - when calling the asynch method
  - when extracting data

event subscription

Clients do not poll: polling is done in Server : polling thread.

Server must be configured/written to generate events.

Server sends data to interested client when “something” changes.

Clients must *subscribe* to events and handle *asynchronous* data.

event subscription

Client creates an object derived from `Tango::Callback` implementig the relevant `push_event(...)` method.

`push_event(...)` is exeucted upon arrival of the event.

Upon successful subscrition at least one event is sent by server.

Client calls `unsubscribe_event()` when no more events are needed.

event subscription

by default Event subscrition fails if Server does not support the requested event.

- if *stateless* is set to true subscrition succeeds anyway, and events will be recived when the server is up and correctly configured!

Heartbeat : client is notified if server is not reachable via event (exception is notified to callback)

Hertbeat automatically re-subcribes event again when server becomes reachable again.

see: event01.cpp

Group

Provides a single point of control for a group of devices: client performs a ***read/write_attribute*** or ***command_inout*** to all the devices with a single call.

Group calls are internally executed asynchronously: speed!

Client creates a Tango::Group, then adds devices

It is also possible to add a Group object to a Group to create a hierarchical object (advanced feature).

Elettra Sincrotrone Trieste

Group

Devices are added by *name*.

Wildcard * can be used in any part of the name (*/*/* : whole control system).

Groups are really useful when:

- Devices have uniform interfaces: use AbstractClasses
- there is a consistent and rational naming convention for device names

-----> Design the control system as a <u>whole,</u> bottom-up AND top-down

Deployment and configuration of Devices is an important and delicate step in the building and maintenace of the control system.

Group

no exceptions during Group calls

error are reported by GroupReplyList object:

- check has_failed() globally
- check has_failed() for each list element
    - can enable excpetions and get an exception while exctrating a datum:
    - `GroupReply::enable_exceptions(true);`

Elettra Sincrotrone Trieste

Group

Group can contain also other Groups

- hierarchical organization

The reply is not heriarchical!

Actions performed on a Group can be forwarded or not to subgroups

- default: forward
- turn off/on globally with statyc method on Group
- additional parameter for actions

Elettra Sincrotrone Trieste

Group

with a Group a client can:

- excute command
  - without argument
  - with same iput argument to all members
  - with different input arguments for different members
- read one attribute
- write one attribute
  - with same value for all members
  - with different values for different members

see: group01.cpp

comprehensive examples on Tango manual

device locking

A client can lock a device server to gain "exclusive" write acces to the device:

```
dev->lock(); .... dev->unlock();
```

A client can verify/check the locking of a device:

```
dev->is_locked(); dev->is_locked_by_me();
dev->get_locker(LockerInfo&);
```

- the lock is reset if no actions is done by locker for an amount of time, if server is restarted, etc. Destroying DeviceProxy unlocks the device.
- **a client can force the unlocking of a device :**

```
dev->unlock(true);
```

- use locking with utmost restraint and care ...

pipes

Pipe: new in tango-9 , can contain any combination of Tango types values, *including other Pipe objects*.

The structure of a Pipe may change at each reading/writing.

Eeach pipe value is named with a string.

Eeach pipe value has its type identifier.

- dynamically discovery of the structure

available only in C++ (jan. 2016)

reading pipes

`DevicePipe DeviceProxy::read_pipe(string&)`

reads the named pipe, then clients must extract data from `DevicePipe` object.

In case the Pipe structure is known beforehand you can use directly the extract operator >>

see : pipe01.cpp

reading pipes

If the DevicePipe structure is unknown or changes call-to-call, you have to analyze it in order to extract the values. Methods `get_data_elt_nb()`, `get_data_elt_name(i)`, `get_data_elt_type(i)` together with `>>` operator are the available tools.

see: pipe02.cpp first part

Elettra Sincrotrone Trieste

writing pipes

Use write_pipe() method :

`DeviceProxy::write_pipe(DevicePipe&)`

You must first build and fill the `DevicePipe` object. You must set:

- pipe name
- number of elements
- names of elements
- element values

DevicePipe has methods on operators to do this

see pipe02.cpp second part

special Tango types

`DevVarDoubleStringArray:`

holds together an array of strings and an array of *double.* Each array has its own length. N.B.: strings are of type `TangoDevString` .

`DevVarLongStringArray:`

holds together an array of strings and an array of *long.* Each array has its own length. N.B.: strings are of type `TangoDevString` .

see tangotypes01.cpp

TangoDevString

`Tango::DevString` is a `char*`.

Use auxiliary functions from `CORBA` namespace: `string_alloc()`, `string_dup()`, `string_free()`. Use C `strcpy()` for copying.

```
DevString mystring = CORBA::string_alloc(5);
strcpy(mystring,"tango");
DevString tangostring = CORBA::string_dup("pluto");
CORBA::string_free(tangostring);
CORBA::string_free(mystring);
```

Sequences

`Tango::DevVar*Array` are CORBA IDL sequences mapped to C++ classes;

They behave like vectors of variable length

Defined for Tango basic types.

Can allocate and manage their own buffer or use a client supplied buffer for the array values.

There are operators for filling `std::vector<*>` to/from `DevVar*Array` Make life easier at the price of some memory copies.

see devvar01.cpp