TANGO has many advanced features

This talk describes some of the features for

Advanced Data Transfer

Chapter 7 - Advanced Features

7.1 Attribute Alarms
7.2 Enumerated attributes
7.3 Device Polling
7.4 Threading
7.5 User events
7.6 Multicast protocol
7.7 Memorized attributes
7.8 Forwarded attributes
7.9 Transferring images
7.11 No database, multiple databases
7.14 Controlled access

- OmniORB default
  - In IOR: First IP address which is not the loopback one
- Since Tango 7.X.Y
  - By default no cmd line option required because
    - All possible addresses in IOR
    - Client tries to connect to each of them until one succeed
  - Warning: ORBendPoint cmd line option disable this feature

- **Database server case**

  - Has to listen on a specific port → use the ORBendPoint option :

    ORBendPoint giop:tcp::1234

    - Host not specified → omniORB default (first one which is not the loopback)
    - Could be the private network IP address
    - Does not prevent DB connection (from the TANGO_HOST)
    - Prevent the event system to work correctly (Java client)

C++TANGO CLIENT: LARGE DATA TRANSFER

- Extracting data from DeviceAttribute into std C++ vector copy the data
- In case of large data transfer
- Extract data to Tango array type by pointer
- Use Tango array type method get_buffer() to retrieve data pointer
- Warning: Extraction done that way consumes memory

→ Don't forget to release it after usage

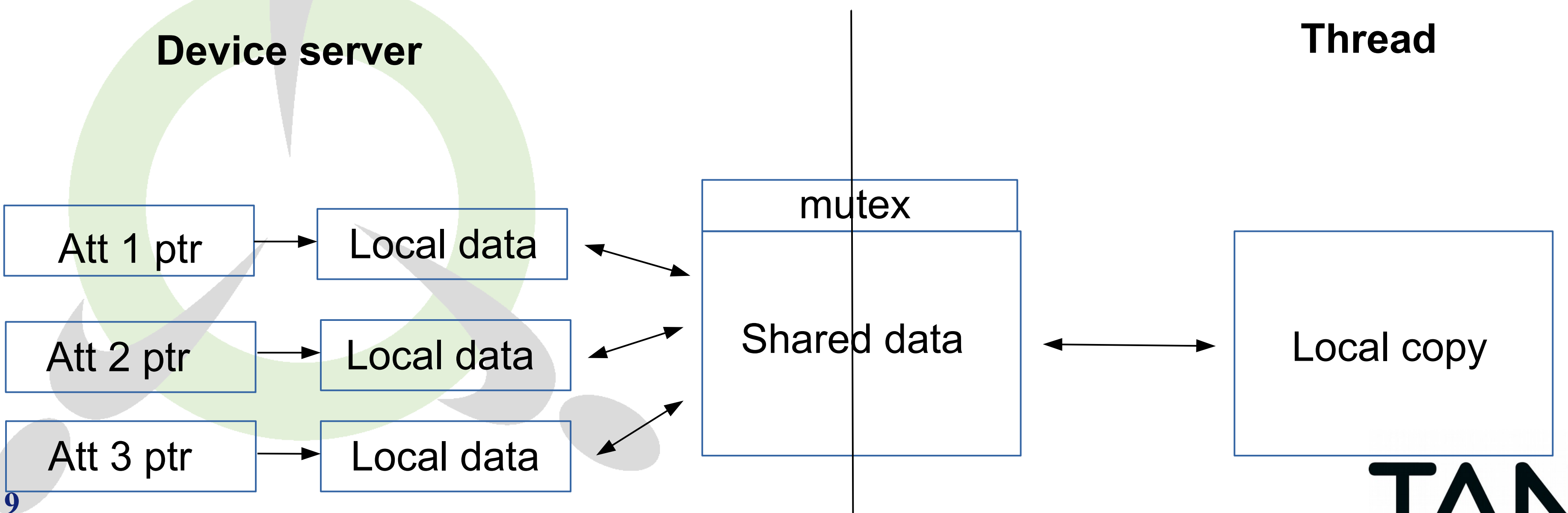
```
Tango::DeviceAttribute da = mydev.read_attribute("LargeData");
Tango::DevVarShortArray *dvsa;
da >> dvsa;
const short *ptr = dvsa→get_buffer();
// use memory
delete dvsa;
```

- Tango DS are multi-threaded process
  - DS without any client: **8 threads**
    - Main thread in the ORB loop
    - 3 ORB threads (2 + scavenger)
    - Signal thread
    - Heartbeat thread
    - 2 ZMQ threads
  - ORB : One thread per client (k clients → k threads up to a max of 50 threads)
  - Polling threads (m threads)
  - Thread number: 8 + k + m

- By default, each device has a **Tango monitor**
  - Monitor “locked” when the thread associated to client A execute a request
  - All other threads associated to other client(s) have to wait until the monitor is “unlocked”
  - Retrieving data from the polling buffer (for polled attributes/commands) does not take the device monitor
    - Polling system has its own monitor system

- Four kinds of serialisation
  - **BY_DEVICE** : Each device has its own monitor
    - Default behavior
  - **BY_CLASS** : Only one monitor for all devices belonging to a Tango class
  - **BY_PROCESS** : One monitor for the whole process
  - **NO** serialisation:
    - Warning: Does your device and your code support this?
    - Database server is using this mode
- Serialisation model choice using **Util::set_serial_model**() in **main.cpp** file (See doc chapter 7.4.1.1)

- If 1 thread per device then use
  init_device() / **delete_device**()
- If 1 thread per class then use
  xxxClass ctor / **xxxClass dtor**



- Convenient to have the device ptr in thread but
  - Take care with **Device_Impl::set_state**() and **Device_Impl::set_status**()
  - Take care of deadlock:
    - Client ask cmdA → thread clnt1 has the device monitor and ask thread to do cmdA
    - Thread execute cmdA and after execution and due to error try to set device state!

```
try
{

dev→get_dev_monitor().get_monitor();
   dev→set_state(Tango::FAULT);
   dev→set_status("....");
   dev→get_dev_monitor().rel_monitor();
}
catch (Tango::DevFailed &e)
{
....
}
```

THREAD CODING

Thread framework

- Initialisation

- /IF/ Init OK
    - /WHILE/ exit is false
        - Sleep a while
        - Get shared data (including cmd)
        - Do work
        - Set result in shared data
    - /END WHILE/
- /ENDIF/

Safest way
(Paranoid code)

```
try
{
        Thread code
}
catch (std::exception &e)
{
…
}
catch (…)
{
...
}
```

Thread creation: Thread code is one method of the device class

```
class MyDevice: public TANGO_BASE_CLASS
{
….
     void th_main(InitData);
….
    std::thread   th;
}
```

```
void MyDevice::init_device()
{
…
    InitData id;
    id.xxx = yyyy;
    th = std::thread(&MyDevice::th_main,this,id);
…
}
```

```
void MyDevice::delete_device()
{
….
   /* Send exit cmd to thread */
   if (th.joinable() == true)
       th.join();
…
}
```

Thread init phase:
Using promise/future

```
void MyDevice::init_device()
{
…
   the_promise.reset(new std::promise<void>());
   std::future<void> fut = the_promise->get_future();

  /* Start thread */

   try
   {
      fut.get();
   }
   catch (Tango::DevFailed &e)
   {
       set_state(Tango::FAULT);
       set_status(….);
   }
…
}
void
MyDevice::delete_device()
{
…
  the_promise.reset();
…
}
```

```
class MyDevice: public TANGO_BASE_CLASS
{
….
    void th_main(InitData);
    void init_thread(InitData);
….
   std::thread                                  th;
   std::unique_ptr<std::promise<void>>   the_promise;
}
```

```
void MyDevice::th_main(InitData _id)
{
  bool init_failed = false;
  try
  {
    init_thread(_id);
    the_promise->set_value();
  }
  catch (Tango::DevFailed &e)
  {
    the_promise->set_exception(std::current_exception());
    init_failed = true;
  }

  If (init_failed == false)
  {
   ….
  }
}
```

Thread paranoid code: using atomic data type

```
class MyDevice: public TANGO_BASE_CLASS
{
….
    void th_main(InitData);
    void init_thread(InitData);
….
   std::thread             th;
…
   std::atomic_bool       unk_except;
}
```

```
void MyDevice::init_device()
{
…
   unk_except = false;

   /* start thread...*/
…
}

void MyDevice::always_executed_hook()
{
   If (unk_except == true)
   {
      set_state(Tango::FAULT);
      set_status(….);
      unk_except = false;
   }
}
```

```
void MyDevice::th_main(InitData
_id)
{
   try
   {
      /* Thread code */
   }
   catch (...)
   {
      unk_except = true;
   }
}
```

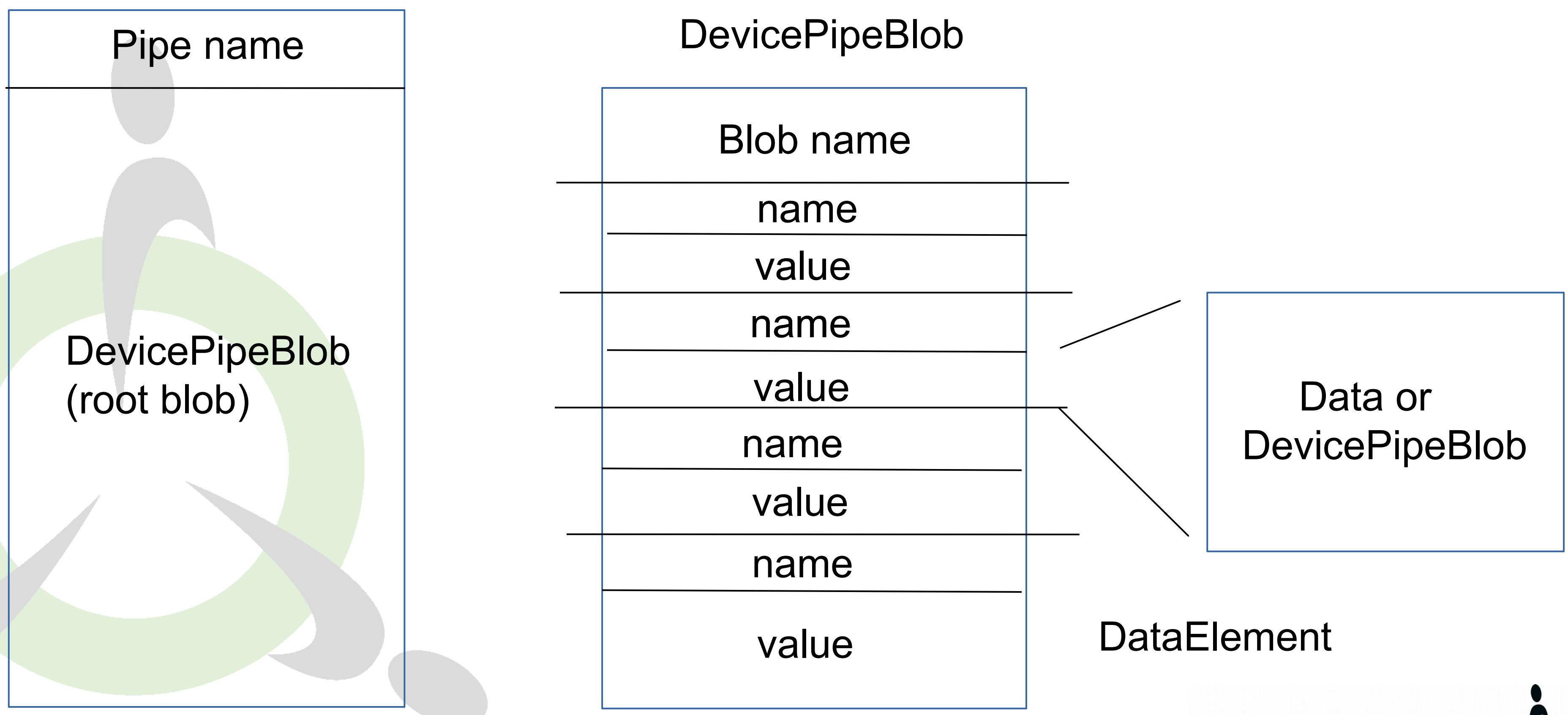
- Shared data
  - Include one enum for thread command
    - With a exit thread command
  - Add a std::mutex instance to protect them
    - Eventually a std::condition_variable instance to trigger cmd execution while thread is in its sleeping time.

- Each Tango device may have pipe(s) on top of command / attribute
- Data transferred through pipes do not have fixed type. Type is dynamically defined at each read/write
- Each pipe has
  - A name
  - A reduced configuration with
    - R/W type (read or read-write)
    - Display level
    - Description (settable at run-time)
    - Label (settable at run-time)

Data transferred through pipe are :

- The pipe name
- A blob which is
  - A name
  - A set of **DataElement**
    - Each DataElement is a name / value pair.
    - The value part may be
      - Data of any Tango type (scalar or array)
      - A blob (Recursive data type)

Using Pipes (Tango 9)



- Using Pipe (Tango 9)
  - On the **client** side :
    - **DeviceProxy::read_pipe**() to read a pipe
    - **DeviceProxy::write_pipe**() to write a pipe
    - **DeviceProxy::write_read_pipe**() to write then read a pipe
    - **DeviceProxy::get_pipe_config**() and **DeviceProxy::set_pipe_config**() to get/set pipe config
  - On the **server** push Pipe events :
    - **DeviceImpl::push_pipe_event**()

- Using Pipes (Tango 9)
  - Reading a pipe with prior knowledge of pipe content(C++)
    - Example: a long, array of double, array of unsigned short

```
DevicePipe dp = mydev.read_pipe("ThePipe");

DevLong dl;
vector<double> v_db;
DevVarUShortArray *dvush = new DevVarUShortArray();

dp >> dl >> v_db >> dvush;

delete dvush;
```

- Name lost!
  Use template class DataElement<T> instead of basic data type

Using Pipes (Tango 9)

Reading a pipe **without** prior knowledge of pipe content(C++)

```
DevicePipe db = dev.read_pipe(“ThePipe”);

size_t nb_elt = dp.get_data_elt_nb();
for (size_t loop = 0;loop < nb_elt;loop++)
{
   int data_type = dp.get_data_elt_type(loop);
   string de_name = db.get_data_elt_name(loop);

   switch (data_type)
   {
     case DEV_LONG:
     {
       DevLong  lg;
       dp >> lg;
     }
     break;
     ….
   }
}
```

Using pipe (Tango 9)

Writing a pipe

Similar to reading a pipe but nb of data element in blob has to be defined before inserting data (with << operator)

```
DevicePipe dp("MyPipe");

vector<string> de_names {"first","second","third"};
dp.set_data_elt_names(de_names);
….
dp << dl << v_db << dvush;

mydev.write_pipe(dp);
```

Using pipes (Tango 9)

On the **server** side

When the client read a pipe:

- always_executed_hook()
- is_<pipe_name>_allowed()
- read_<pipe_name>()

The read_<pipe_name>() method received a "Pipe" instance reference. The use of Pipe class is very similar to the DevicePipe class on the client side (but in a read_<pipe_name> method, you insert data into the pipe).

Using pipes (Tango 9)

On the **server** side

- When the client write a pipe:
  - always_executed_hook()
  - is_<pipe_name>_allowed()
  - write_<pipe_name>()
- The write_<pipe_name> method receives a WPipe object. Using it, pipe extraction is possible
  - Similar to reading a pipe in the client side

Using Pipes (Tango 9)

Blob inside blob?

- Classes DevicePipe, Pipe or WPipe contains a DevicePipeBlob instance.
  - Methods previously described used this hidden instance (root blob)
- User may
  - create DevicePipeBlob instance
  - insert / extract data in / from it in a similar way
  - insert / extract from instance in / from the root blob (or another DevicePipeBlob) as the data part of a DataElement.

TANGO controls

EVENTS INTERACTION WITH TANGO_HOST + IP ADDRESS