

Writing a Subsystem Manager Device Server using YAT

*some feedback about the design and
implementation of a system supervisor
involving hundreds of heterogeneous
devices...*

Writing a Subsystem Manager Device Server using YAT

some feedback about the design and implementation of a system supervisor involving hundreds of heterogeneous devices...

Writing a Subsystem Manager Device Server using YAT

*just want to let you know that there's
"something" on tango-ds that could be
useful in some of your projects...*

Use Case: FOFB Manager

- need an example to illustrate this talk!
 - a typical YAT use case
-

Use Case: FOFB Manager

- 240 Tango devices involved
 - 48 X-corrector power supplies
 - 48 Z-corrector power supplies
 - 120 BPMs (Libera)
 - 48 "associated with x-corrector" bpms
 - 48 "associated with z-corrector" bpms
 - 24 "no corrector" bpms
 - 17 timing boards
 - 2 fiber network sniffers (Libera intercon.)
 - 1 DCCT (stored current)
 - 3 misc. (SOLEIL internal cooking)
-

Use Case: FOFB Manager

- a device in order to...
 - configure ,
 - monitor,
 - control the FOFB system ...

 - ... as a whole!
 - one button FOFB start/stop
-

Use Case: Constraints

- critical application for machine operation, so...
 - ... should be reliable, efficient, scalable and... maintainable!
-

Use Case: Design

- adopt the simplest possible design with a clear separation of concern between subsystems management
 - concentrate supervision logic into a dedicated entity
-

Use Case: Design

- use “active objects”
 - split global activity into separate “tasks” running in parallel and exchanging data via messages
 - one orchestrating the global activity and running the core logic
 - actual manager - knows how FOFB system works!
 - one per subsystem to manage/ctrl
 - knows how its associated FOFB subsystem works
 - hi-level subsystem interface for the manager
 - get (and pre-compute) data ...
 - ... then forward it to the manager (or report errors)
-

Use Case: Impl.

- a task = a thread + a msgQ
 - task activity triggered by messages
 - passively waits for msgs
 - no activity means no CPU consumption
 - external vs internal notifications
 - “self generated” msgs
 - PERIODIC: best effort on-time delivery
 - TMO: “no activity” notification
-

Use Case: Impl.

- handle one message at a time
 - a task can't be interrupted while processing a msg
 - msgs serialization
 - limitation? might be in very specific contexts
 - FIFO msgQ with msg priority is supported
 - support the lo/hi watermark paradigm
 - provides a way to detect system overload

 - make the message a generic container
 - msg = type/ID + optional data
 - can post "any" data to a task
-

The Use Case: Impl.

- might be a recurrent need so...
 - externalize these sw tools into a library
 - make this lib a general purpose C++ toolbox
 - offer more than the simple task impl.
 - so... the we are about to write **yet another toolbox?**
 - yes, so... let's call it **YAT!**
-

YAT: Overview

- C++ toolbox offering misc. services
 - threading: task and sync. objects
 - memory management: [circular] buffer<>, ...
 - network com.: client & server sockets
 - misc: timer, "a la Tango" exception, bits-stream, ...
 - Supported platforms
 - Windows, Linux & MacOS X
 - Does it have some Tango dependency?
 - no dependencies
 - but... we have Yat4Tango
 - YAT specialization for Tango
 - `yat::Task` vs `yat4tango::DeviceTask`
-

YAT: Overview

- C++ toolbox offering misc. impl. tools
 - threading: thread, task and sync. objects
 - memory management: [circular] buffer<>, ...
 - network com.: client & server sockets
 - misc: timer, "a la Tango" exception, bits-stream, ...
 - Supported platforms
 - Windows, Linux & MacOS X
 - Any dependency?
 - no – can be used in "any" project – not only devices
 - native impl. - use target platform API
-

YAT: Overview

- What is Yat4Tango?
 - a YAT specialization for Tango
 - `yat4tango::DeviceTask`
 - `yat4tango::ThreadSafeDeviceProxy`
-

YAT: Using the DeviceTask

yat4tango::DeviceTask

```
public:  
    virtual void go (size_t tmo);  
    virtual void exit ();  
    virtual void post (yat::Message* msg, size_t tmo);  
    virtual void wait_msg_handled (yat::Message* msg, size_t tmo);  
  
protected:  
    virtual void process_message (yat::Message& msg) = 0;
```

YAT: Using the DeviceTask

yat4tango::DeviceTask

```
virtual void process_message (yat::Message& msg) = 0;
```



MyTask

```
virtual void process_message (yat::Message& msg);
```

YAT: Using the DeviceTask

MyTask

```
virtual void MyTask::process_message (yat::Message& msg)
{
    switch ( msg.type() )
    {
        case yat::TASK_INIT:
            break;
        case yat::TASK_EXIT:
            break;
        case yat::TASK_PERIODIC:
            break;
        case yat::TASK_TIMEOUT:
            break;
        case MY_DATA_MSG:
            break;
    }
}
```

YAT: Using the DeviceTask

yat4tango::DeviceTask

```
public:  
    virtual void go (size_t tmo);
```

- DeviceTask::go
 - synchronous impl.
 - waits up to *tmo* ms for this task to start
 - msg. id: yat::TASK_INIT
-

YAT: Using the DeviceTask

yat4tango::DeviceTask

```
public:  
    virtual void exit ();
```

- DeviceTask::exit
 - synchronous impl.
 - asks the task to quit, joins with the underlying thread then returns
 - msg. id: yat::TASK_EXIT
-

YAT: Using the DeviceTask

yat4tango::DeviceTask

```
public:  
virtual void post (yat::Message* msg, size_t tmo);
```

■ DeviceTask::post

- posts the specified *msg* to the task – i.e. inserts it into its *msgQ* according to *msg* priority - then returns
 - in case the *msgQ* is full - i.e. hi-watermark reached - waits up to *tmo* ms for the *msgQ* to reach its low watermark - throws a `Tango::DevFailed` in case of failure
 - system overload, too slow processing ...
-

YAT: Using the DeviceTask

yat4tango::DeviceTask

```
public:  
virtual void wait_msg_handled (yat::Message* msg, size_t tmo);
```

- DeviceTask::wait_msg_handled
 - same as *post* but also waits up to *tmo* ms for the *msg* to be handled
 - synchronous msg. handling might be required for some specific msgs
 - what's about using a MyTask public method instead?
 - you can do that but... no race condition with *wait_msg_handled* – msg processing is serialized
-

YAT: Using the DeviceTask

```
#define MY_DATA_MSG yat::FIRST_USER_MSG + 1

MyTask * t = ...;

MyDataStruct * d = ...;

yat::Message * m = new yat::Message(MY_DATA_MSG);

m->attach_data(d);

t->post(m, 1000);
```

YAT: Using the DeviceTask

MyTask

```
virtual void MyTask::process_message (yat::Message& msg)
{
    switch ( msg.type() )
    {
        case MY_DATA_MSG:
            this->process_data ( msg.get_data() );
            or
            this->process_data ( msg.detach_data() );
            break;
    }
}
```

(back to) Use Case: Impl.

- we said...
 - one task orchestrating the global activity and running the core logic
 - actual manager - knows how FOFB system works!
 - one per subsystem to manage/ctrl
 - knows how its associated FOFB subsystem works
 - hi-level subsystem interface for the manager
 - get (and pre-compute) data ...
 - ... then forward it to the manager (or report errors)
-

Conclusion

- simple & efficient design/impl
 - we obtain very good results in terms of performances, stability and maintainability
 - if you want to give YAT a try...
 - [sourceforge/tango-ds/libraries](https://sourceforge.net/projects/tango-ds/libraries)
 - contributions are welcome!
-