Prototype Development: Experiences, Progresses and Lessons Learned

C.Baffa, E.Giani

LMC harmonization through Telescopes
Step2: LMC Peer Review
Meeting 2

Madrid, 11-13 April 2016

Reference framework

The reference framework is established by:

- SKA1_MID Telescope Interface Control Document CSP to TM
- Interface Control Document LMC to CSP Sub-elements
- SKA CSP Local Monitor and Control Sub-element Detailed Design Description
- LMC Interface Guidelines Document
- Tango Interface Guidelines

The development has been performed in the MID mental framework, but as functionalities are in common with LOW, the prototype structure would differ in minor details.

The Prototype Aim

The prototype is intended to:

1. Test the Tango ability and find the best approaches to implement the main CSP.LMC functionalities:
   - The conversion of TM command to sub-element level commands
   - The sub-elements communication and coordination
   - Handling of alarms, events and messages
   - Handling of timed commands
   - Monitoring points and report general/detailed status
2. Verify the compliance of requirements about timings in critical operations (re-configuration, alarms notifications, initialization etc.) and/or get a better estimate of these timings
3. Test, if possible, a small subset of design alternatives

First Experience

The first device was intended as 'test-bed' for Sys Class

- Reuse of a portion of a Community Tango Class
- Customization for our specific needs
- Name 'TemperatureMonitor', but not only temperatures!
- First experiences of handling arrays of attributes.
- Experience on Tango standard tools

Lessons learned:

Power and limits of Pogo (limit on inheritance)

Inheritance & abstract classes

SKA Concepts as Tango Entities (1)

Mapping SKA concepts in terms of Tango ones:

- Standard SKA status variables to Tango 'enum'
- Problems with State, two alternatives:
  - Use of Tango-State as SKA-OpState: we miss some states
  - Use of a custom variable as SKA-OpState: we miss state machine
- Drill down in engineer mode: *use of real tango device names*
- Hierarchy of servers and of Tango facilities as SKA elements (➦ Lize's presentation in Trieste)

SKA Concepts as Tango Entities (2)

Lessons learned:

State problem: we do not want a Tango fork!
Some weakness in Tango enum
Use of a central 'devices & proprieties' repository

- *Mapping SKA ICDs to Tango Control System,* C.Baffa, E.Giani, Arcetri Technical Report 3/2015

Vertical Simulator

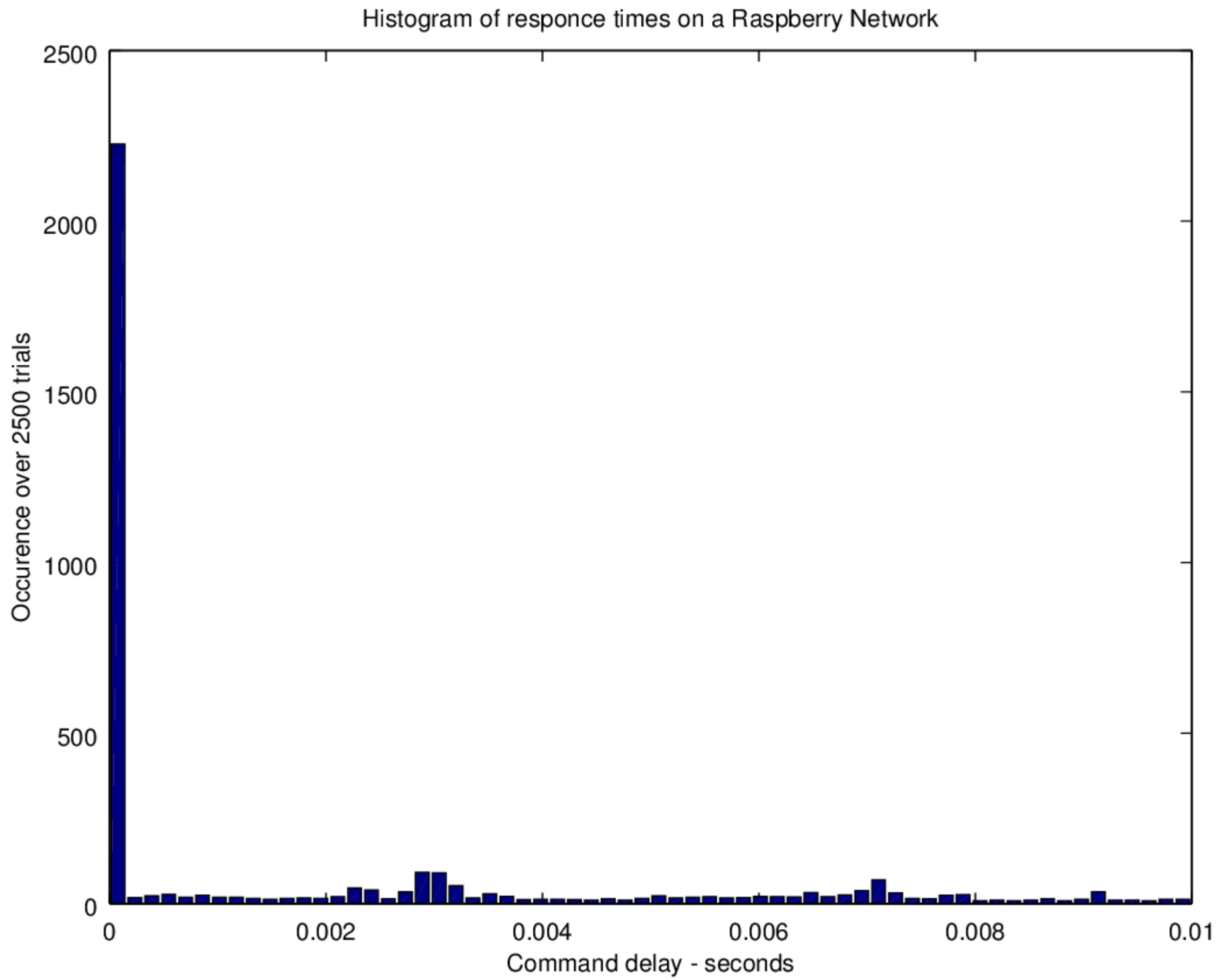
A Vertical simulator in order to test Connectivity

- Three level communication: from 'TM' to a 'Master' to the 'Devices'
- Test connections and commands/message exchange
- Possibility to handle a large number of *device* nodes
- Experience on responsiveness and timings

Lessons learned:

High speed of communications
Normally low latency, occasional larger one

Vertical Simulator Response time



The first bin is 2 microseconds wide.

The Prototype Functions

The CSP.LMC Prototype shall implement overall CSP monitor and control.

- Maintain and control the overall CSP status
- Implement the interface with TM and SubElements.
- Receive and execute TM commands
- Perform mapping of TM commands to command for individual CSP SubElements
- Handle timed commands
- Configure SubArrays and allocate the Capabilities to them
- Handle CSP alarms, events and other messages received from the CSP SubElements.
- Maintain a log of all the activities

Basic Assumptions & Requirements

1. Use of Tango as control framework
2. Most, if not all, TM interactions flow through CSP.LMC
3. TM is agnostic about the detailed hardware structure
4. TM sends coherent and complete commands to CSP.LMC
   - CSP.LMC performs syntactic and minimum safety checks
5. TM sends detailed configurations for scan programming (EICD) as compounded settings for parameters or compounded commands:
6. TM sends immediate and timed commands.
   - the implementation of command queues on all Master nodes
   - Each command identified by an ID
   - A structured system of acknowledge for immediate and delayed commands

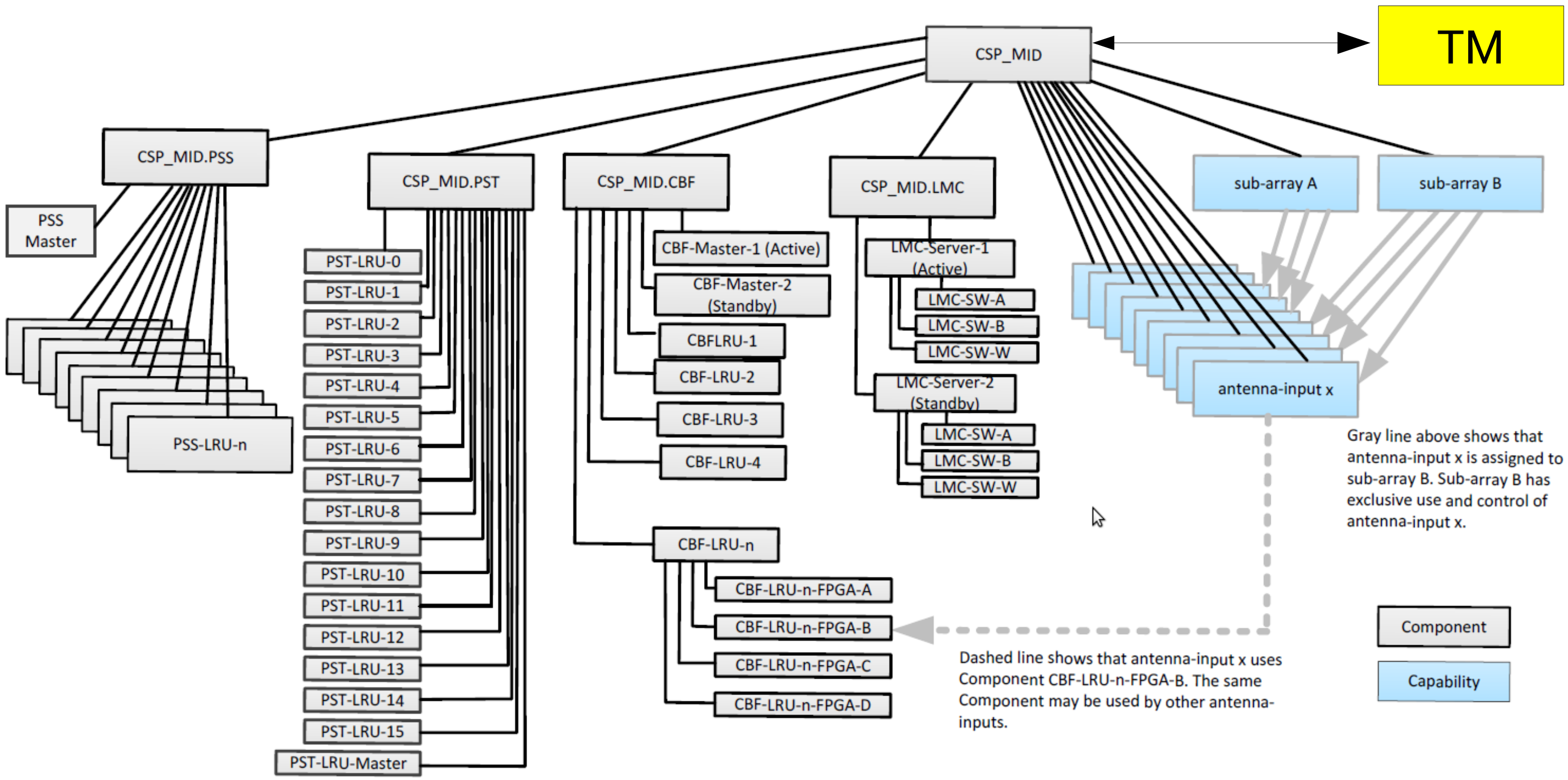
Inside our Tango Classes

To define the Tango Classes of the CSP.LMC prototype we started from the two ICD documents:

- from the *EICD* we have derived the attributes and methods common to all elements, sub-elements and capabilities → we have defined few *abstract* classes
- from the *IICD* we have derived the attributes and methods specific to each sub-element and capability

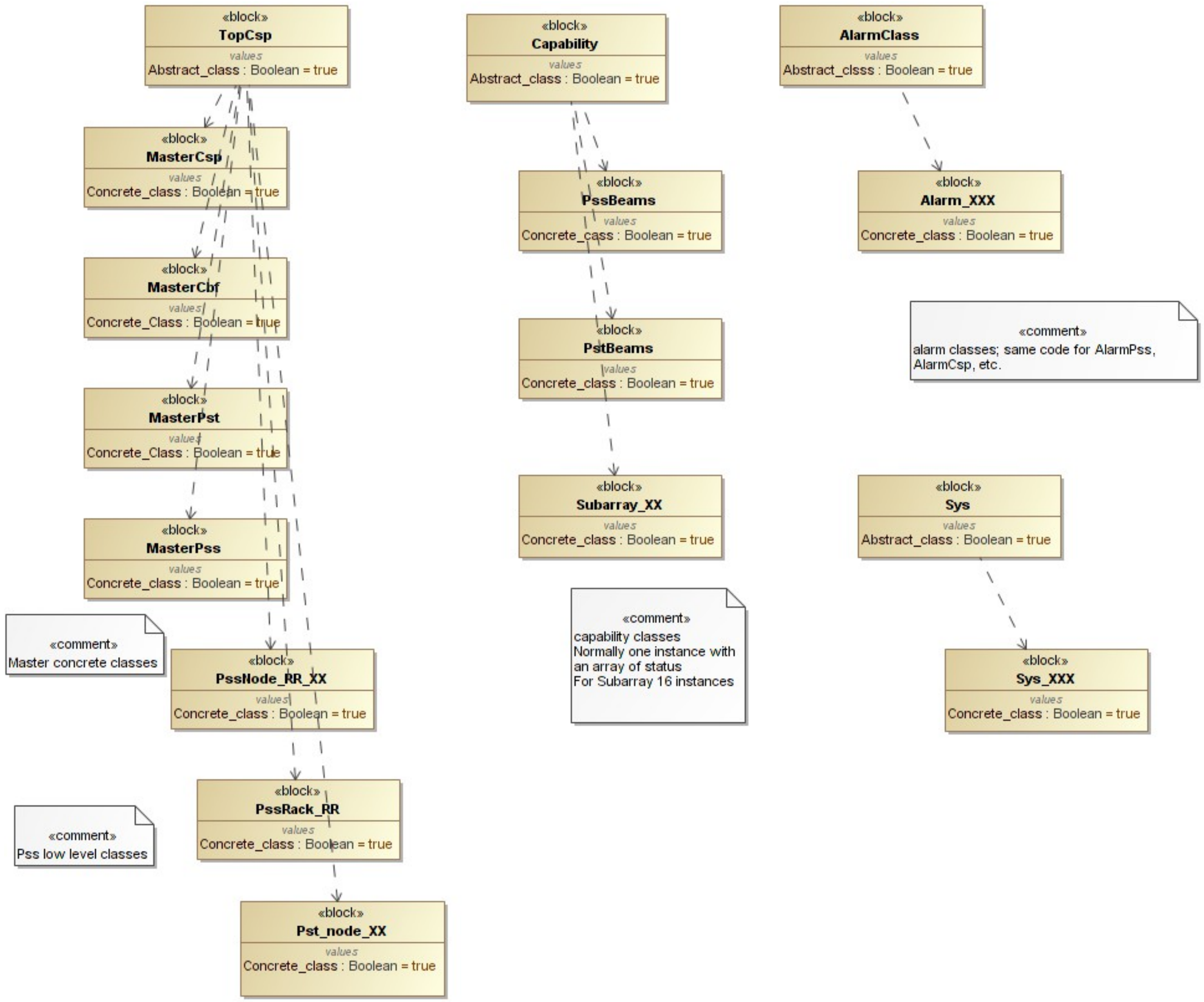
CSP Detailed Structure

Figure 7-1 Monitor and control hierarchy for CSP Mid– this diagramhas been provided as an example of the monitor and control hierarchy, it does not show the accurate number of Components.



From: *S.Vrcjc SKA ICD SKA Document*

Taxonomy of the prototype classes



The prototype structure (1)

- The prototype structure is modeled on the CSP architecture: Each M&C entity is implemented as a Tango Device Servers running one or more Tango Devices

↓

- ✓ One Tango Device Server for CSP Element
- ✓ One Tango Device Server for each SubElement (CBF, PSS and PST)
- ✓ Each Device Server runs on a separate PC (Master Node)

- The prototype will implement some M&C functionalities as Tango Devices.
- The prototype uses the Tango System Logging for logging (open!)

The prototype structure (2)

We see two alternative approaches to sub-array implementation:

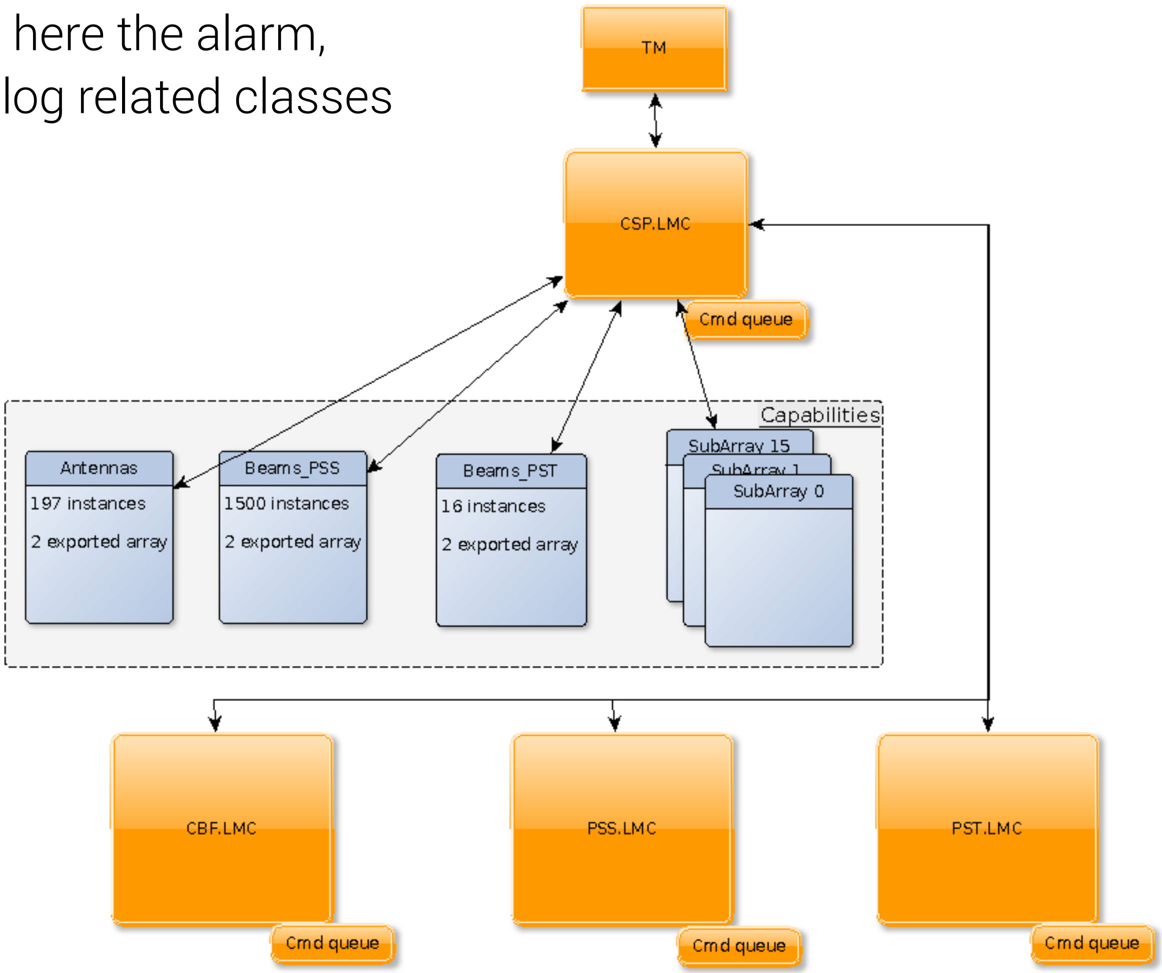
a) Implementation of subarrays as 16 separate telescopes which share a pool of hardware resources

b) A single hardware pool which can be organized in up to 16 subarrays.

→ **In our prototype we have implemented model b).**

The prototype structure (3)

We omit here the alarm, sys and log related classes



The prototype structure (4)

- There will be one top-level CSP and three sub-element Master Nodes: 4 pc based on COTS hardware and SO (Linux)
- In each master node will run one or more Tango Devices servers

On the CSP Master node:

- The CSP.LMC Tango Device
- The CSP.SYS Tango Device
- A command handler device (scheduler)
- An Alarm Handler Device
- The Capability Device(s)
- The SubArray Device(s)
- A logging system

The prototype structure (5)

- On the three SubElement Master nodes:
  - The SubElement LMC (CBF.LMC, PSS.LMC, PST.LMC)
  - The SubElement SYS
  - The Alarm Handler
  - A command handler device (scheduler)
  - A logging system
  - These devices can run in a single Tango Device Server (as a multi-class device) or can run in separate Tango Device Servers.

→ Alternative: single server (Box) or independent servers?

- Single server might be affected by Thread-Safety issues

Prototype main points

- Tentative Naming Schema
- State/Mode Variables
- Parameter setting, *setParam*
- Capability/SubArray strategy
- Scenarios execution analysis
- Alarms implementation
- Initialization strategy

Prototype main points

- **Tentative Naming Schema**
- State/Mode Variables
- Parameter setting, *setParam*
- Capability/SubArray strategy
- Scenarios execution analysis
- Alarms implementation
- Initialization strategy

Tentative Naming Schema for CSP (1)

Object	JsonID	Class	Device in Database	ObjectID
Master CSP	csp	MasterCsp	proto/master/csp	MasterCsp
		SubArray	Proto/capability/subarray	SubArray
		LmcMaster	proto/lmcmaster/csp	CspSys
		Alarm	proto/alarm/csp	CspAlarm
Master PSS	pss	MasterPss	proto/master/pss	MasterPss
		LmcMaster	proto/lmcmaster/pss	PssSys
		Alarm	proto/alarm/pss	PssAlarm

Tentative Naming Schema for CSP (2)

Object	JsonID	Class	Device in Database	ObjectID
Master PST	pst	MasterPst	proto/master/pst	MasterPst
		LmcMaster	proto/lmcmaster/pst	PstSys
		Alarm	proto/alarm/pst	PstAlarm
Master CBF	cbf	MasterCbf	proto/master/cbf	MasterCbf
		LmcMaster	proto/lmcmaster/cbf	CbfSys
		Alarm	proto/alarm/cbf	CbfAlarm

Prototype main points

- Tentative Naming Schema
- **State/Mode Variables**
- Parameter setting, *setParam*
- Capability/SubArray strategy
- Scenarios execution analysis
- Alarms implementation
- Initialization strategy

SKA Operational State (1)

SKA		Tango
OFF	This is a Powered off state.	OFF
READY	This suggests that the Element is ready to operate	ON
SHUTTING-DOWN	This is a transient state in which the Element is shutting down.	MOVING
HYBERNATE	Special non-operational state in which Entity has been placed after intialization. From this state it can transit to OFF or SLEEP.	DISABLE
SLEEP	Special non-operational state in which Entity has been placed to reduce power consumption. From this state it can transit to HYBERNATE or READY.	STANDBY
FAILED	An Element reports an 'Error' state when it detects a problem that affects its ability to accept certain commands or execute certain processes/operations.	FAULT
UNKNOWN	TM is not aware of actual state of Element.	UNKNOWN
INITIALIZING	This is a transient state in which the Element exists when it is starting up its processes.	INIT

SKA Mode Variables

From: LMC Interface Guidelines Document

SKA			Tango name
Element Type	*Real*	a Real Element is connected	Element_Type
	Simulated	a simulator is connected in place of a real Element.	
	Standby	a Element/Simulator is connected and is used as a backup device for providing redundancy	
	Not-Fitted	the Element/sub-element is not fitted.	
Control Mode	*Central*	Element is under TM control.	Control_Mode
	Local	The element is under manual/local control of the LMC.	
Operating Mode (alternative to Administrative Mode)	*Enabled*	The Element is allowed to perform activities.	Operating_Mode
	Disabled	The Element is intentionally excluded from performing activities or participating in an operation	
	Maintenance	The Element is reserved for maintenance activities like diagnostics, configuration changes, commissioning	
	Test	The Element is reserved for setup / testing activities.	
	Safe	This mode imposes functional restrictions that increase resilience to equipment damage	
Health Status	*Normal*	Element is in normal working condition	Health_Status
	Degraded	Element is functioning in degraded condition when subset of its functionality is compromised or unavailable.	
	Failed	Implies when there is major failure that prevents Element to perform its function.	
	NotOperable	Element is not available for observations due to missing dependencies.	
Usage Status	*Idle*	Element is not in use but it is available for use.	Usage_Status
	Active	Element is performing in observations, but it still has operating capacity to provide for more observations.	
	Busy (?)	Element is performing in one or more observations and cannot participate in any other observations	

Table 3: Synopsis of proposed mapping of SKA Status and Mode variable to Tango Attributes.

States Use Cases (1)

Tango State very useful for a simple physical device.

but

For a complex physical device or for a logical one can be not enough. The associated state machine cannot cope with its complexity.

States Use Cases (2)

Some examples:

1. We have two subArrays busy on 1000 and 500 PSS beams, PST in fault, CSP is ready. But I can still make image observations
2. All is working, we have two subArrays busy on 1000 and 500 PSS beams, we will put PST in low-power mode
3. We have two subArrays busy on 800 and 500 PSS beams, and 200 PSS beams are in Fault. How TM will know it cannot allocate further beams?

→ **We need to use all SKA defined Status Variables!**

Arrays of State/Mode Attributes

We need to use, report and summarize many SKA Status variables

For a complex physical device as CBF, PSS and PST we need to report the logical states of a large number of devices, from tenths to thousands.

Most efficient handling by means of arrays

Tango still do not implement arrays of enum,

↓

SKA Status Variables
will be implemented as array of shorts

Prototype main points

- Tentative Naming Schema
- State/Mode Variables
- **Parameter setting, *setParam***
- Capability/SubArray strategy
- Scenarios execution analysis
- Alarms implementation
- Initialization strategy

Parameters Setting

At startup:

Succession of defaults:

- Tango library default
- Class default
- Tango database
- Hardwired code

At set-up:

- Use of setParam command
- SetParam('json object');

For special/engineering purpose a remote setAttribute(s) command

SetParam Command (1)

setParam accepts attribute settings and general commands

```
Command:  setParam     From: TM  Destination: CSP.LMC
(cspMaster).
Argument: Json  String {
"activationTime": "10:30:00", // should be a Unix time
"sourceId": "TM",
"commandId" : "123456",// identifies this execution
"subArray0": { // init of subArray
    "antennasList": "0,1,2,3,4,10,100",
    "creationDate": "20160310 10:30:00",
    "administrativeMode": "enabled",
    "observingMode": "0",   // idle
}
```

setParam can have a complex command structure inside

Json argument versus structured Pipe:
efficiency, flexibility, easier to maintain

SetParam Command (2)

setParam for a complete 1500 beams PSS parameter set-up has a 100K json string argument. We have a sintetic Json generator.

```
Command:  setParam            From: TM  Destination: CSP.LMC (cspMaster).
Argument: Json  String {
"activationTime": "10:31:00", // should be a Unix time
"sourceId": "TM",
"commandId" : "123456",       // identifies this execution
"GlobalValues": { // init of internal variables common to all subsystems
          "subArrayId": "4",
          "ObservingMode": "2", // PSS
          "scanId":  "AB45-34",          // We store scanId for subArray 4
          "numberOfBeams": "500"
}
"CSP" : {
          // CSP specific parameters
          "PSSBeamID" : ["AB45-34/1", "AB45-34/2", … "AB45-34/500"] // 500 values
          "PSSPointingCoord" : [ … ] // 500 values
          "PSSDestinationAddress" : ["10.1.1.1:4000", … "10.1.50.10:4000"] // 500 values
}
"CBF.Master": {
           "setSubArray":{    // specialized command
                    "scanTime":  "34.12" ,         // scan (integration) time
                    "subArrayObsMode":  "2", // PSS
                    "numberOfChannels":  "4096", // PSS
                    "beamBw":  "2", // PSS
                    "bitPerSample":  "8", // PSS
                    "Filter Banks Parameters" : { … ,}      // many hardware related parameters
                    "Delay Model Parameters" : { … },
                    "commandId" : "123456/2",      // identifies this execution
          }
          "setBeams":{        // specialized command
                    "numberOfChannels":  "4096",
                    "PSSBeamID" : ["AB45-34/1", "AB45-34/2", … "AB45-34/500"] // 500 values
                    "Beam Ponting Parameters" : { … },      // many hardware related parameters
                    "commandId" : "123456/3",      // identifies this execution
          }
}
"PSS.Master": {
           "setSubArray":{    // specialized command
                    "subArrayId":  "4", // in the  fourth slot we host subArray 4
                    "scanTime":  "34.12" ,         // scan (integration) time
                    "subArrayObsMode":  "2", // PSS
                    "beamBw":  "2",
                    "accelerationRange" : "0",
                    "DispersionMeasure":  "300",
                    "programming Parameters" : { … }        // many hardware related parameters
                    "commandId" : "123456/4",      // identifies this execution
          }
           "setBeams":{       // specialized command
                    "beamBw":  "2",
                    "accelerationRange" : "0",
                    "DispersionMeasure":  "300",
                    "PSSBeamID" : ["AB45-34/1", "AB45-34/2", … "AB45-34/500"] // 500 values
                    "programming Parameters" : { … }        // many hardware related parameters
                    "commandId" : "123456/5",      // identifies this execution
          }

}
}
```

TM Json simulator

Command line generator for observation programming command

Usage: ./json_generator <option(s)> <mode>Mode:

- CBF Generate a CBF only command string
- PSS Generate a PSS only command string
- CSP Generate a combo CBF & PSS string (default)

General Options:

- -h Show this help message
- -o Output file (default out.json)

PSS Options:

- -b Specify the number od PSS beam to generate (default 50)

```
Command:  setParam            From: TM  Destination: CSP.LMC (cspMaster).
Argument: Json  String {
“activationTime”: “10:31:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”,       // identifies this execution
“GlobalValues”: { // init of internal variables common to all subsystems
          “subArrayId”: “4”,
          “ObservingMode”: “2”, // PSS
          “scanId”:  “AB45-34”,         // We store scanId for subArray 4
          “numberOfBeams”: “5”
}
   "CBF": {
        "setBeams": {
            "numberOfChannels": 4096,
            "subArrayId": 4,
            "subCommandId": "123456/3"
        },
        "setSubArray": {
            "bemBw": 2,
            "bitPerSample": 8,
            "delayModel": [
                [
                    "3.62",
                    "0.27",
                    "6.90",
                    "0.59",
                    "7.63",
                    "9.26"
                ], ….
            "numberOfChannels": 4096,
            "subArrayObsMode": 2,
            "subCommandId": "123456/2"
        }
    },
```

```
    "CSP": {
        "setBeams": {
            "pssDestinationAddress": [
                "10.1.1.0:4000",
                "10.1.1.1:4000",
                "10.1.1.2:4000",
                "10.1.1.3:4000",
                "10.1.1.4:4000"
            ],
            "pssPointingCoord": [
                "05:34:31.83 +22:00:52.86",
                "05:34:31.77 +22:00:52.15",
                "05:34:31.93 +22:00:52.35",
                "05:34:31.86 +22:00:52.92",
                "05:34:31.49 +22:00:52.21"
            ]
        }
    },
    "PSS": {
        "setBeams": {
            "accelerationRange": 1,
            "bemBw": 2,
            "dispersionMeasure": 300,
            "subArrayId": 4,
            "subArrayObsMode": 2,
            "subCommandId": "123456/4"
        },
    "activationTime": 1459870550,
    "commandId": 123456,
    "globalValues": {
        "numberOfBeams": 5,
        "observingMode": 2,
        "scanId": "AB45-34",
        "scanTime": "34.12",
        "setBeams": {
            "pssBeamId": [
                "AB45-34/34",
                "AB45-34/35",
                "AB45-34/36",
                "AB45-34/37",
                "AB45-34/38"
            ]
        },
    },
    "sourceId": "TM"
}
```

Json parameter
for 5 PSS beams

Commands

A large set of pre-defined commands from Tango for engineering use. For normal operation we use setParam and a small set of specific ones

Command name	Input data type	Output data type
State	void	Tango::DevState
Status	void	Tango::DevString
Init	void	void
DevRestart	Tango::DevString	void
RestartServer	void	void
QueryClass	void	Tango::DevVarStringArray
QueryDevice	void	Tango::DevVarStringArray
Kill	void	void
QueryWizardClassProperty	Tango::DevString	Tango::DevVarStringArray
QueryWizardDevProperty	Tango::DevString	Tango::DevVarStringArray
QuerySubDevice	void	Tango::DevVarStringArray
AddLoggingTarget	Tango::DevVarStringArray	void
RemoveLoggingTarget	Tango::DevVarStringArray	void
GetLoggingTarget	Tango::DevString	Tango::DevVarStringArray
GetLoggingLevel	Tango::DevVarStringArray	Tango::DevVarLongStringArray
SetLoggingLevel	Tango::DevVarLongStringArray	void
StopLogging	void	void
StartLogging	void	void

Prototype main points

- Tentative Naming Schema
- State/Mode Variables
- Parameter setting, *setParam*
- **Capability/SubArray strategy**
- Scenarios execution analysis
- Alarms implementation
- Initialization strategy

Capability Strategy

Proposal

- Most of processing intelligence inside the MasterCsp
- Capabilities as information and configuration container
- We consider capabilities as a different view to the real hardware, more like a mental organization tool.

Consequence

This separates hardware handling from logical entities handling, as per Tango approach.

Alternatives:

an array of data structures inside MasterCsp.

Capabilities Implementation

- We will implement capabilities as container of configuration and status
- We have therefore the necessity to handle a list or array of complex data structure inside a Tango class.
- Tango attribute can only be arrays of simple types.
- We need to access the structures as a whole, but also to access arrays of specific attributes (es. Health Status)

The solution we devise:

- inside of the Tango device a list or array of the appropriate structures
- to make visible, as Tango attributes, only arrays of needed attributes

Alternative: synchronized arrays of simple types

Example of Capability Data: PSS case

Field Name	Example Data	Source of data
BeamId	1234	MasterCsp
HealthStatus	Normal	MasterPss
Available	No	MasterCsp
UsageStatus	Active	MasterCsp
SubArray	1	TM
CbfBeam	233	MasterCbf
PssResource	node_01_13_B	MasterPss
PssDevice	proto/node_01_13_A	MasterPss
PssInputAddr	10.0.10.54:4000	MasterPss
SdpInputAddr	10.1.12.11:3500	TM
TimeStart	2147483647	TM
TimeEnd	2147484577	TM
CreationTime	2147483600	MasterCsp

1500 instances

Capabilities view

PstBeams

AntennaInputs

PssBeams

Public (Tango attributes):
- HealthStatus[1… 1500]
- Usage[1… 1500]
- BeamId[1… 1500]
- ...

Private:

```
Struct {
  HealtStatus,
  Usage,
  BeamId,
  CbfBeam,
  PssResource,
  …
  SdpAddress} [1500];
```

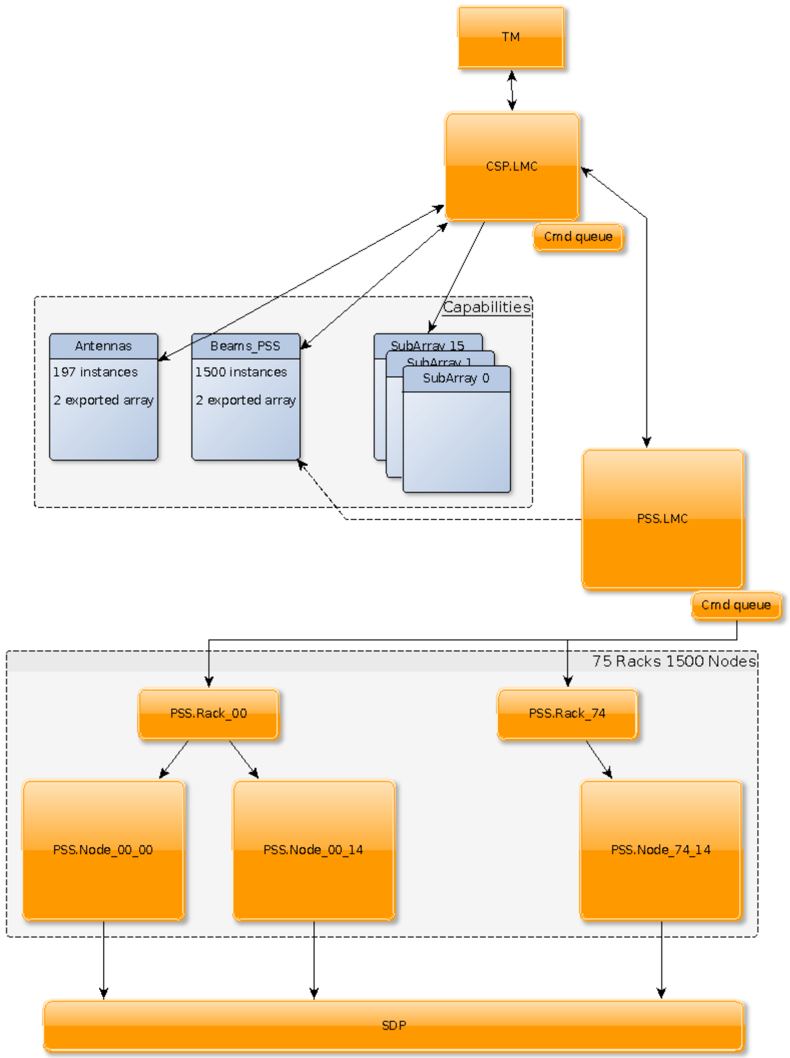
Sub-Arrays

- Sub-Arrays implemented as 16 instance of a simple device driver class
- We will implement Sub-Arrays as a container of configuration and status

Alternative:

- Inside MasterCsp Class device a list or array of the appropriate complex structures.
- A single SubArray Tango Device with an array of complex structures as the others capabilities.

Pulsar Search M&C



A glance to Pulsar Search (1)

One PSS node can process from 2 up to 12 CBF-beams.
Each CBF-beam data is processed by a single data pipeline

We will have N nodes, each with M parallel data processing pipelines,
Gives the N * M = 1500 PSS.MID-Resources. Now N=750, M=2.
256 FPGA boards of the CBF.MID can form up to 1536 CBFBeams used by the PSS.MID.

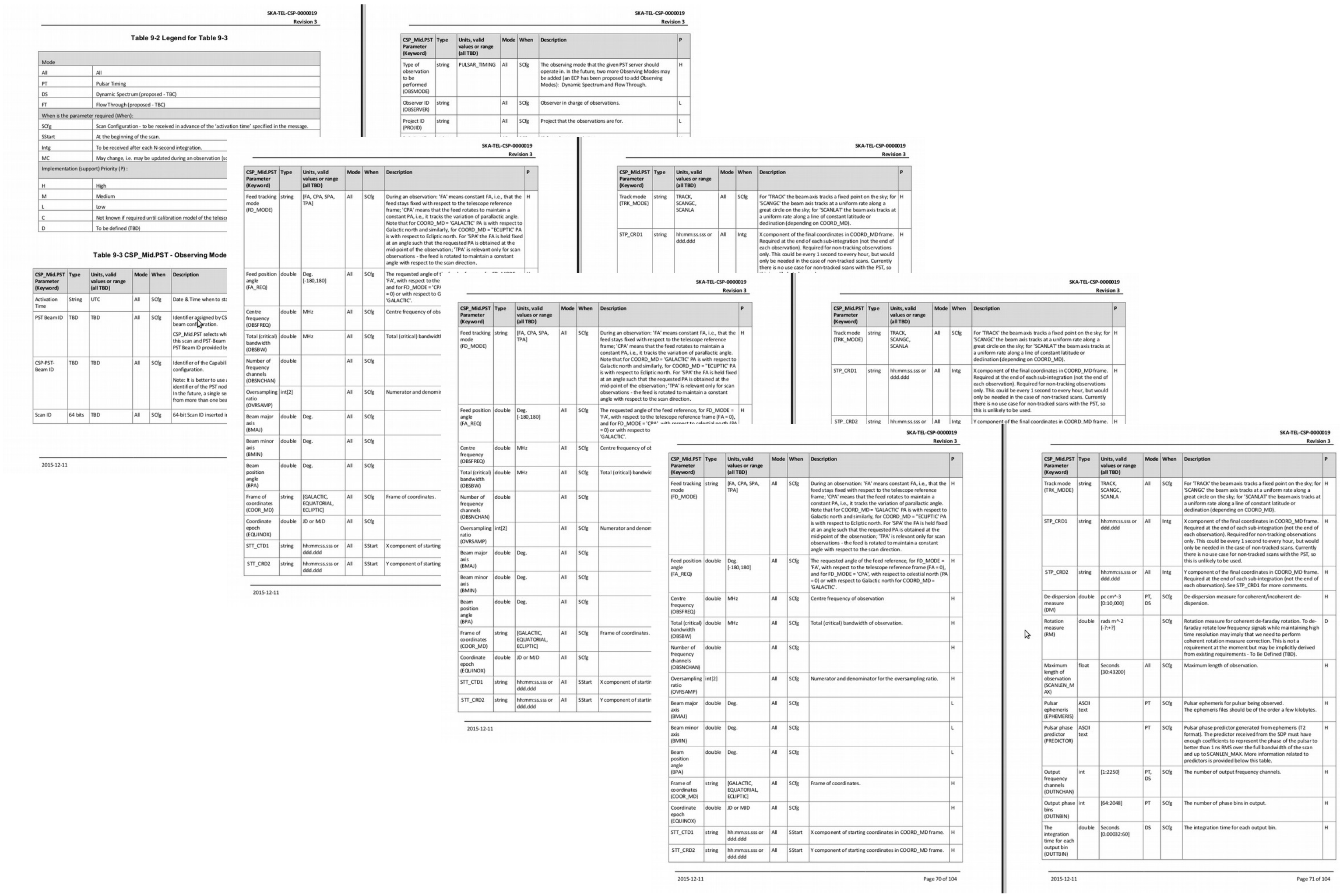
A glance to Pulsar Search (2)

A resource at PSS level corresponds to a single software data pipeline, which process data coming from the associated CBF-beam.

Each PSS-Resource is characterized by (Slide 35):

- a name corresponding to the Tango device name (running on a PSS node).
- an IP Address-port combination for data input
- a SDP IP Address-port combination for output products.
- A symbolic name, for instance: Node_RR_PCX where RR is the rack, PC is the PC sequential number, and X is the pipeline identifier
- Few ancillary data (creation and expiration times, etc).

EICD Parameters



Prototype main points

- Tentative Naming Schema
- State/Mode Variables
- Parameter setting, *setParam*
- Capability/SubArray strategy
- **Scenarios execution analysis**
- Alarms implementation
- Initialization strategy

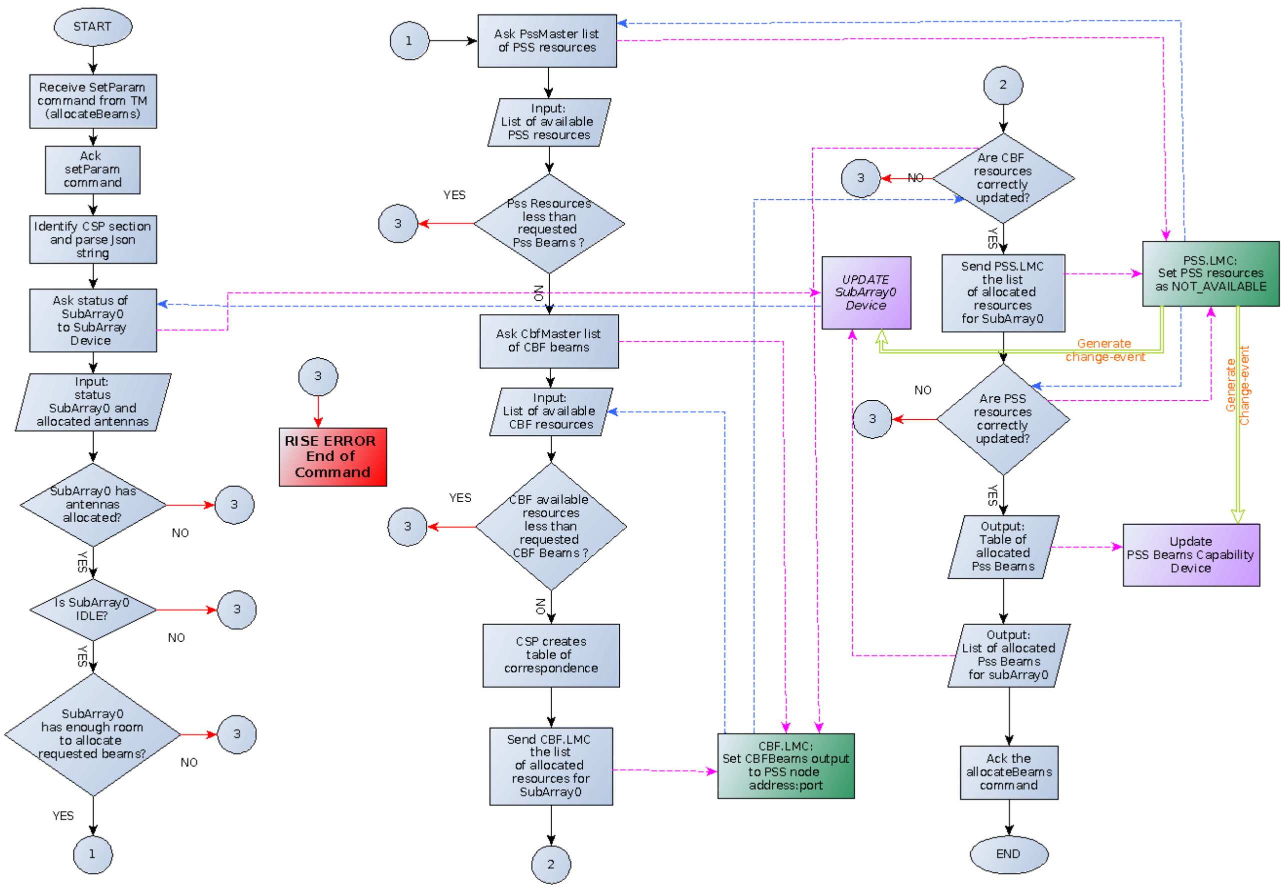
Scenario Example – 1
Allocation of 500 PSS beams

setParam accepts attribute settings and general commands

```
Command:  setParam  From: TM  Destination: CSP.LMC (cspMaster).
Argument: Json  String {
“activationTime”: “10:31:00”, // should be a Unix time
“sourceId”: “TM”,
“commandId” : “123456”,  // identifies this execution
“CSP” : {
     “allocateBeams”: { // init
          “beamsType”: “PSS”, // it can be PSS, PST and VLBI
          “subArrayId”: “0”,
          “beamsCount”: “500”,
          “creationDate”: “20160310 10:31:00”,
          “commandId” : “123456/1”,// identifies this execution
     }
}
}
```

setParam can have a complex command structure inside

Scenarios Example (2)



Graphic flow of beam allocation operations. Error handling in red, Capabilities in purple and SubDevices in green

Scenarios Example (3/1)

CSP.LMC	subArray{0-15}	PssBeams Capability
Receive the command from TM		
Acknowledge command 123456		
Identify the CSP section and parse it		
Spawn the execution of command (123456/1)		
Verify if subArray0 is already initialized and has Antennas allocated	Report status of subArray0 and allocated Antennas	

Scenarios Example (3/2)

CSP.LMC	subArray{0-15}	PssBeams Capability
If subArray 0 has not any Antennas allocated Rise error → End of Command		
If subArray 0 is not IDLE Rise error →End of Command		
Ask PssMaster list of available PSS Resources		
If available PSS Resource are less than requested beams Rise error →End of Command		

Scenarios Example (3/3)

CSP.LMC	subArray{0-15}	PssBeams Capability
Ask CbfMaster list of available CBF Beams Resources		
If available CBF Beams Resource are less than requested beams Rise error →End of Command		
CSP.LMC creates a correspondence table between PSS Resource and CBF Beams		

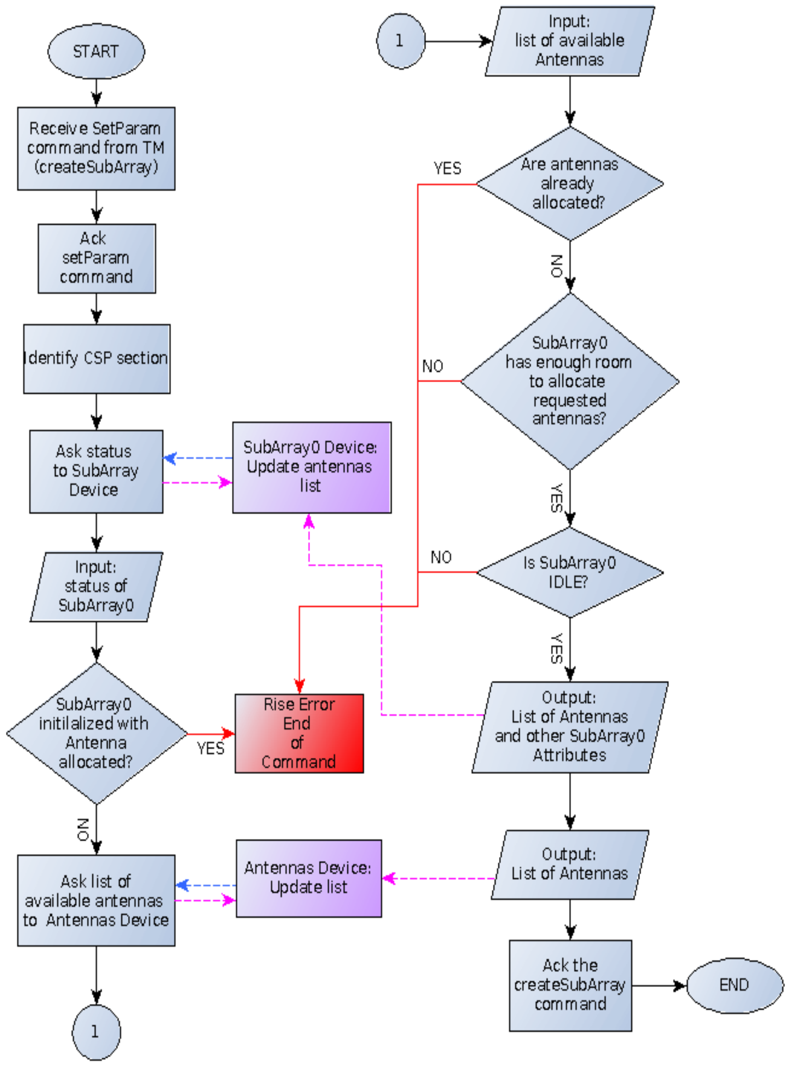
Scenarios Example (3/4)

CSP.LMC	subArray{0-15}	PssBeams Capability
CSP.LMC update PssBeam Capability		Update the PssBeam correspondence table
CSP.LMC update SubArray Capability	Update the implemented PssBeam table	
If available subArray PssBeam table is full Rise error →End of Command		

Scenarios Example (3/5)

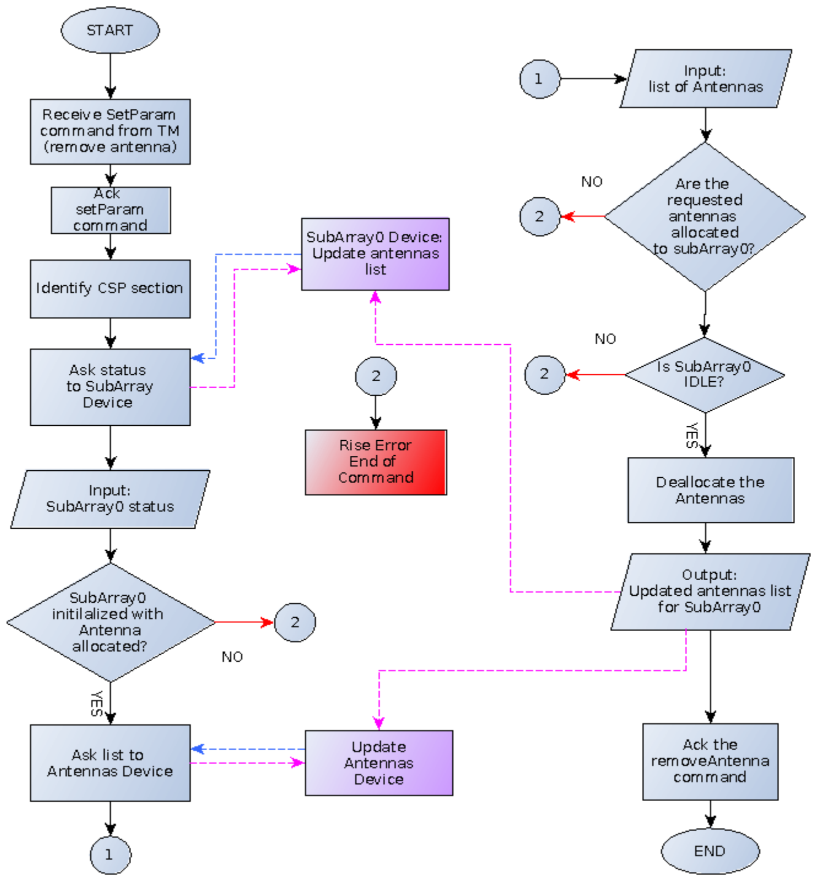
CSP.LMC	subArray{0-15}	PssBeams Capability
Write other attributes to SubArray0 (creationDate, administrativeMode, observingMode, ...)	Update other attributes to SubArray0	
Acknowledge successful execution of command 123456/1		
Acknowledge successful execution of command 123456		
End of execution		

More simple examples (1)



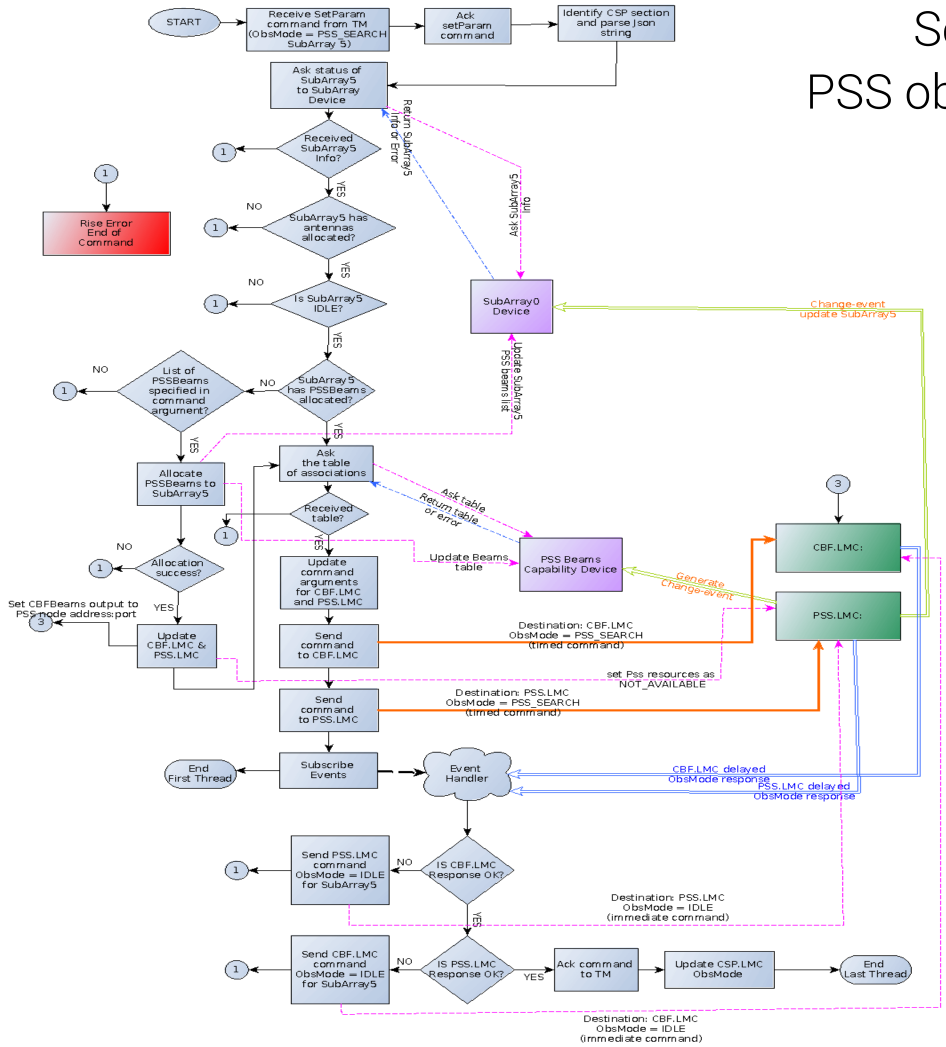
Initialization of a subArray

More simple examples (2)



Antennas removal
from a subArray

A More Complex Example



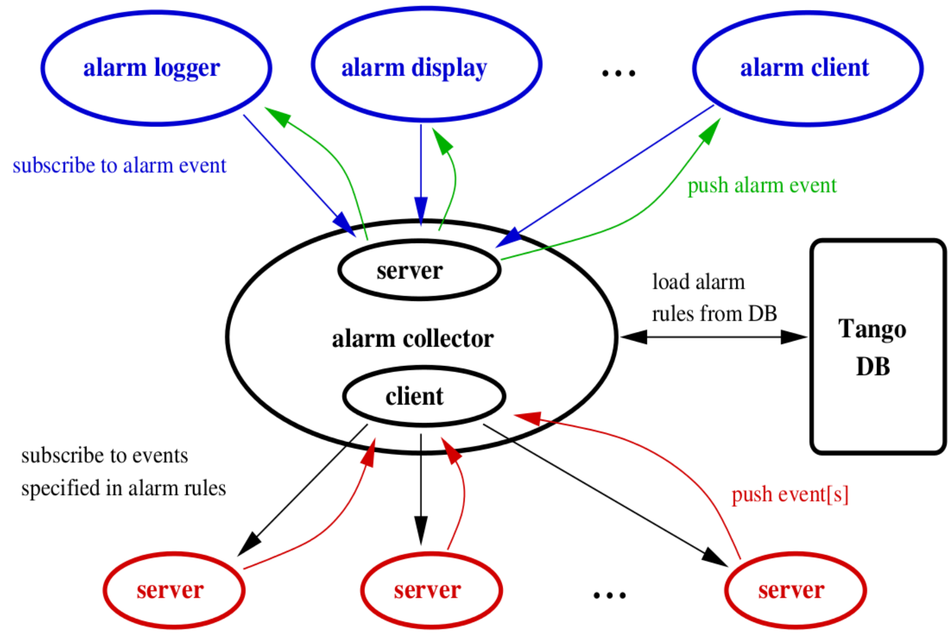
Set-up of a PSS observation

Prototype main points

- Tentative Naming Schema
- State/Mode Variables
- Parameter setting, *setParam*
- Capability/SubArray strategy
- Scenarios execution analysis
- **Alarms implementation**
- Initialization strategy

Alarms Handling (1)

We plan to use Tango C++ Alarm System



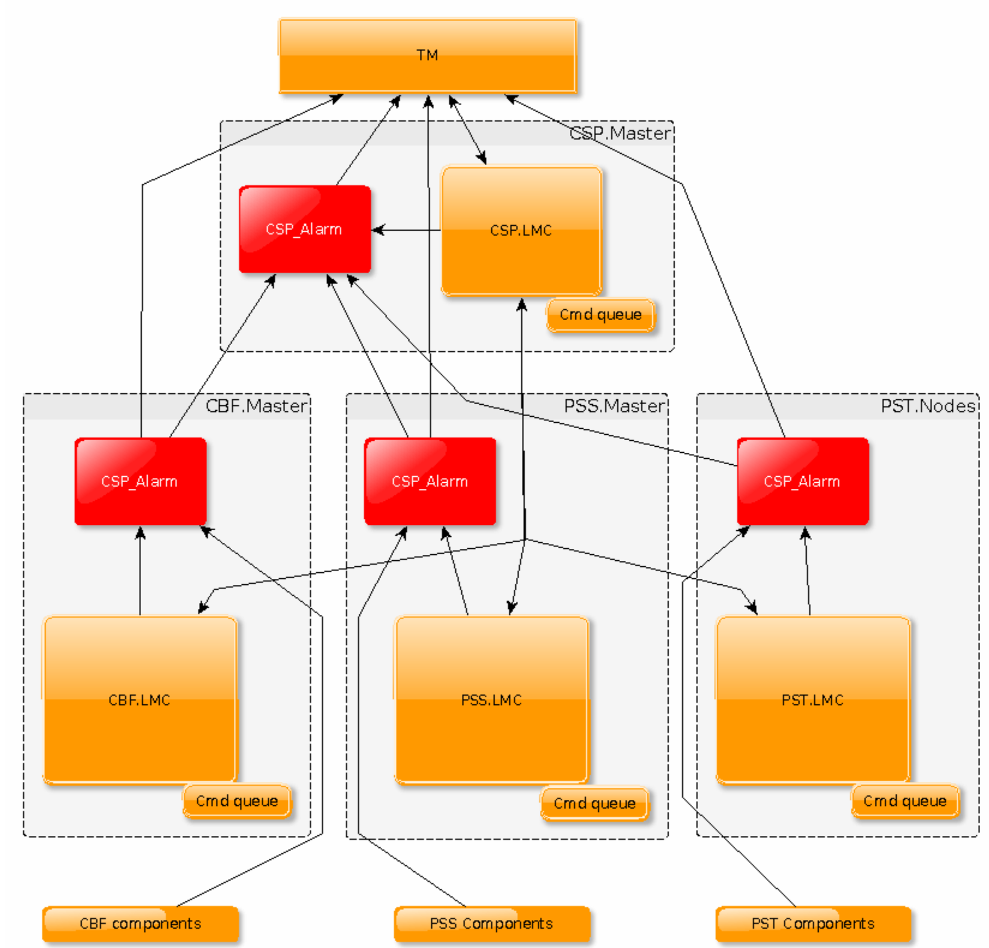
Alarms Handling (2)

From thousands to millions of attributes: We definitively need both a **fast Implementation** and a **hierarchical approach.**

Alarms in Tango are out of limits exception.

The Alarm device driver can convert this scenario to a complete Alarm System.

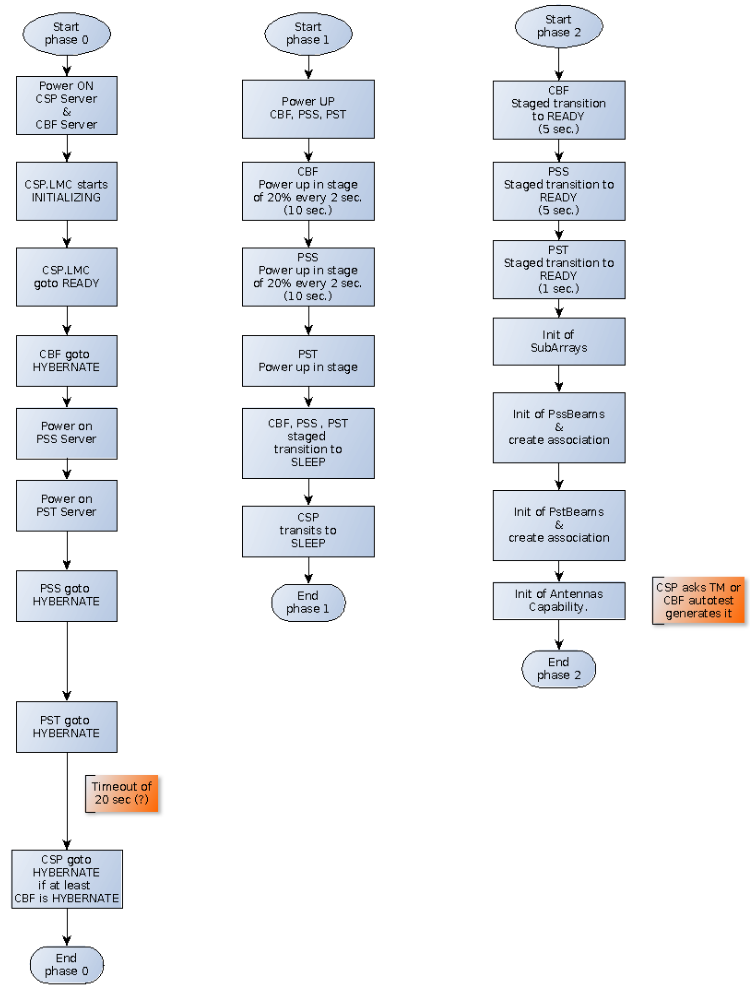
Alarms hierarchy



Prototype main points

- Tentative Naming Schema
- State/Mode Variables
- Parameter setting, *setParam*
- Capability/SubArray strategy
- Scenarios execution analysis
- Alarms implementation
- **Initialization strategy**

Initialization Schema



Initialization Strategy

Long and complex task.

Tree implementation alternatives:

1. Sequential initialization inside main Tango Driver
2. Sequential initialization inside a yat4tango thread
3. Parallel initialization using a yat4tango thread for each sub-element.

We have implemented already 3, but we have some interprocess communication issues.

Comments and Suggestions?

Thank you!

Special thanks:
Marina Vela Nuñez for the presentation review
Luca B. for the presentation layout