Contribution ID: **4** Type: **not specified**

LFAA LMC Overview Use-Case

Introduction

This section gives a brief list some of the features for an LMC system, and what kind of control the system should provide to the TM interface layer for generic control of all elements. Primarily, LMC will be concerned with some or all of the following, to different extents:

1. Monitoring and control of digital devices and any hardware/software components
2. The detection of any specified alarms/evens, to inform appropriate recipients
3. The running of computational/logical jobs/tasks with specified schedules
4. Routine checks and diagnostics
5. Exposing functionality in a logical fashion, or in detailed form for debugging and fault mitigation
6. Interaction with any operators/users
7. Routing, processing, saving of data streams etc.
8. Management of the available computational resources

For the purposes of the LFAA LMC, the above requirements could be serviced by various software blocks as follows:

1. Monitoring/control of devices/hardware –> TANGO Server/Drivers + Hardware interfaces/Access Layer
2. Alarms/Events –> TANGO Drivers + more generic core
3. Resource management –> Apache Mesos
4. Scheduled jobs –> Apache Aurora (on top of Mesos) and native services
5. Routine checks –> Applications running via Aurora
6. Exposing functonality –> LFAA Interface API
7. Interaction with users/operators –> TANGO wrapper over LFAA Interface API
8. Routing/processing/saving data –> GlusterFS logical volumes + low-level DAQ services + real-time processing services

TANGO Device Control - TANGO States

All LFAA LMC requirements must fall within the Telescope Operating States, which are assigned manually by the telescope operator, while the operating status can be set by the LMC or by TM. An example of a typical state flow could be as follows:

- The system starts in the “off”state, where all hardware/software is powered down. Once powered up the state transitions to “standby”, the default operating status. A number of transitions are possible (a) Debug/Maintenance: occurs when telescope is set to maintenance mode, (b) Low-power: occurs when a command from TM sets the telescope to low-power mode, (c) Safe-state: an optional state (similar to low-power), (d) Init: occurs when an observation schedule is received by TM and the telescope needs to be initialized (forming stations, programming boards, setting up workflows etc.), (e) Faulty: occurs in cases where a serious or critical error is detected.
- “Debug”, “low-power”and “safe”states will remain in those states until TM instructs a change.
- “Faulty”state can go to “standby”if the fault is mitigated/fixed, or to the “off”state.
- Errors can occur in all operating states, so every state can transition to the “faulty”state.

It is apparent during our prototyping that Tango (v8/v9) does not allow for custom states to be added to the set of states in the framework. There are some workarounds e.g. modifying the source code to allow for more states, but this breaks Tango-related packages like Pogo (the device designer). Tango 9 has introduced ENUM attributes, but one can’t assign the Tango state machine system to attributes out of the box.

In our current prototyping, we provide a design pattern to write a new state system within a Tango device driver that can replicate the same functionality for Tango states, but with any list of custom states.

The Python TANGO driver will include:

1. A dictionary called **state_list**, which contains a list of key/value pairs, the key is the driver function name, and the value is a list of states (represented by a numeric value) of allowed states under which that function is runnable.
2. (a) An inbuilt function called "**check_state_flow**"which, given a function name, checks if the current state of the device is in one of the allowed states. This function returns True/False accordingly.

Furthermore, with this re-implementation that mirrors the Tango state-flow control, the predefined Tango state/status information is redundant, though it could still be used for some specific cases if needed. We anticipate replacing the "numeric"value in "state_list", with an ENUM value.

TANGO Device Control - TANGO Device Hierarchy

In LFAA, a number of antennas are connected to a single tile (via a Tile Processing Module, or TPM). Based on the Telescope Model information, tiles are configure into logical stations, and some of the operations will be transmitted to stations rather than to individual TPMs.

We can define a TANGO station with the following non-exhaustive capabilities:

1. Add/Remove TPM to/from Station
2. Connect/Disconnect TPM in a Station
3. Set/Get station state (which is an aggregate of TPM states)
4. Run station command - a command across all TPMs in the Station, and return all results for each TPM

This hierarchy is therefore very straightforward. We shall simply go over the implementation model to send commands to all TPMs within a station, waiting and gathering all results. A number of criteria must be catered for:

1. The command to be executed must exist on all connected devices.
2. Each TPM must be in a state where the command in question can be run.
3. Arguments for the command have to be passed in to the various TPMs.

It is trivial to check if the command exists on all TPMs. In order to pass in parameters to the various TPMs, a pickled list of dictionaries can be utilized. If the list contains only one dictionary, then the same parameters are applied to all TPMs. If there is more than one item in the list, then the number of items in the list must be equal to the number of TPMs in the station.

When passing a command to all TPMs, the Station device maintains an index, called "**command_indexes**". Each command per TPM is run asynchronously in TANGO via the "**command_inout_asynch**"TANGO call. This returns a command ID which can be polled for completion, and we store this in "command_indexes".

A while-loop is then able to use the command IDs stored to poll for a result from each individual TPM by utilising the "**command_inout_reply**"TANGO call, which by default returns an exception when polling a command that has not yet returned. Once all results are accounted for, stored in an array, the Station call can return these replies.

TANGO Parameter Limitations

Commands in TANGO can, at most, pass in/out a single parameter. The easiest way to circumvent this issue when multiple parameters are required is to use a string representation of a set of key/value pairs. In the case of pyTango, this can be easily achieved by using a pickled representation (giving a string) of a dictionary. For example, from within the TANGO driver, one can "un-pickle"the argin input argument as follows:

```
self.debug\_stream("Unpacking arguments...")
arguments = pickle.loads(argin)
commandName = arguments['commandName']
inDesc = arguments['inDesc']
outDesc = arguments['outDesc']
allowedStates = arguments['states']
```

TANGO Parameter Limitations - Other Devices

The LFAA LMC will require the control of other devices such as Switches, PDUs. TANGO drivers for these devices will also be written in a similar fashion to those for TPMs, with a state-handling mechanism as described.

Moving Upwards

The LFAA LMC use-case requires that additional components aside from a TANGO-based control system to be available. In the arcitecture currently being developed, some of these components are Apache Mesos, Apache Aurora and GlusterFS. We give a brief overview of each.

Moving Upwards - Apache Stack for Resource/Workflow Management

The LFAA will require a set of nodes, the "Monitoring, Control and Calibration Servers", which is a compute cluster. On this MCCS cluster a number of applications and services will be run (including the TANGO service). Cluster management technologies can be employed to manage the resources of each node as well as submit applications and services as jobs. Traditionally, the resources of a cluster are handled separately for each node, such that the job submission system as well as the administrators need to know how the cluster is configured. An abstraction over this is to aggregate all the available resources as a single entity, such that the entire cluster is viewed as a single large node. These tools implement a level of abstraction on top of operating systems, and can be viewed as a cluster operating system. Apache Mesos is such a tool, a cluster manager on which frameworks can run. These frameworks provide the scheduling and executing logic required to launch jobs and workflows. In this respect, Apache Aurora is being considered. Additionally, large clusters require a level of redundancy and reliability, especially when master nodes fail. Apache ZooKeeper can provide this functionality.

The mix of technologies chosen here may seem daunting. In fact, each block included requires separate configuration and setting up. It is good to minimize the amount of "moving parts"in a design. The stack chosen here can adequately handle the LMC requirements for LFAA. This is not the first design that was considered for LFAA. Earlier designs and test modules included some more/different components that were eventually taken out and or replaced by better packages. The choice was based on some general and some very specific criteria. The products are all open source. They are tailor made to handle the tasks required of them in an LMC setting. They have dedicated teams actively developing the products and fixing bugs/adding features. The TANGO control system was analyzed earlier on for its utility in LMC and was found to be very suitable to device-only control and very scalable for that specific purposes. On the other hand, packages like Apache Mesos, Apache Aurora, GlusterFS etc. are already deployed in very large systems and guarantee scalability and industry standards for their specific purpose as well.

The various units of this Apache stack provide an API/command line tools that can used to run, monitor and control the stack. We anticipate that the required functionality will be wrapped as part of the entire LFAA LMC API. But we do not anticipate that this will be done via a TANGO driver. The reasons are that most of the interaction with some of these units are based on their own UI (for internal use) or through scripts. TANGO does not follow the scripting model, since it is basically a reply-request mechanism meant for simple commands to devices.

Moving Upwards - Logical Volumes

GlusterFS is an open source, distributed file system capable of scaling to several exabytes and handling thousands of clients. GlusterFS clusters together storage building blocks over Infiniband RDMA or TCP/IP interconnect, aggregating disk and memory resources and managing data in a single global namespace. It is based on a stackable user space design and can deliver exceptional performance for diverse workloads. Most existing cluster file systems are not mature enough for the enterprise market. They are too complex to deploy and maintain, although they are extremely scalable and cheap since they can be entirely built out of commodity OS and hardware. GlusterFS solves this problem, by offering stability and low-maintenance setup for some speed trade-offs. GlusterFS is an easy to use clustered file system that meets enterprise-level requirements:

1. GlusterFS can be deployed with the help of commodity hardware servers.
2. No metadata server is required.
3. Any number of servers can access a storage that can be scaled up to several exabytes.
4. Aggregates on top of existing file systems. A user can recover the files and folders even without GlusterFS.
5. GlusterFS has no single point of failure. It is completely distributed, thanks to not having a centralized meta-data server like Lustre.
6. It is not tightly coupled with the OS kernel (like Lustre) and therefore any updates to the system as a whole have no effect on GlusterFS.

As a software component of the LFAA LMC system, GlusterFS will manage one logical cluster wide partition to store raw data files and logs. Each node will have a dedicated partition which will make up part of this logical volume. GlusterFS will automatically:

1. Handle failures in case of drive or node failure

2. Use the fastest available cluster interconnect (in this case, 40GbE network) to transfer data between nodes
3. Recover lost data (depending on the type of volume deployed)

The raw data files will use a custom defined file format, which typically takes the form of an HDF5 format. A directory structure within the logical format will be used to organize data generated by the data acquisition application and logs. All application access to the distributed file system will occur through appropriate calls in the API.

Moving Upwards - Functional Applications

Functional applications refer to processes and application which are not related to monitoring and control, but are required for the correct execution of an observation schedule. These include calibration, pointing, diagnostic routines, and data acquisition. They are referred to as 'functional applications'since they are not part of the software infrastructure, but rather use it to execute upon. They will generally be executed by running a workflow submitted to the workflow manager. These applications need to specify how much resources they require to run, and whether they'll be run as services or one-off. Depending on the availability of these resources, as managed by the resource manager, they will be scheduled to run on the cluster.

A non-exhaustive list of possible functional applications is as follows:

1. Data Acquisition –streaming control data from the TPI to the MCCS cluster, to be processed and stored in HDF5 files.
2. Pointing –beamforming coefficients are calculated every few seconds by the pointing algorithm. The coefficients are downloaded to each TPM (via the TANGO control system).
3. Calibration –Antenna signals need routine calibration.
4. Diagnostics –Integrity checks are performed routinely, to make sure antennas, signal path, firmwares are running within acceptable parameters. Some of the information required by the diagnostic routines can be requested via the TANGO control system.

TM-LFAA LMC Interface

As one can guess, a "core component"is required to manage all the LMC subcomponents themselves, as well as expose the available functionality to external users. The primary user of the LMC is TM. The interface between these two entities is defined in the TM-LFAA ICD. The interface between these two entites is defined in the TM-LFAA ICD. This document only lists a number of high-level requirements which this interface should meet, with no reference to technology preferences (for example, REST, RPC, SOAP). Nevertheless, the interfacing protocol should be unaware of any technology choices adopted within the LMC. It should also be architecturally agnostic, meaning that whatever the underlying architecture of the LMC is, it should be able to satisfy the interface requirements through a standardised API. To accomplish this task, the interface on the LMC side should "flatten-out"the functions provided by all system components and provide an API which is de-coupled from the underlying deployment. Broadly, we can list the generic aims of the API as follows:

1. Implement the TM-LFAA interface
2. Manage the telescope configuration
3. Report errors/alarms through an appropriate notificaton system
4. Manage role-based priviliges and audits
5. Monitor and control the diverse set of underlying components (TANGO-based or not)
6. Logging

One realises quickly that there needs to be a way to amalgamate in an effective way, the method by which each different component is managed and communicated with.

TM-LFAA LMC Interface - Components and Capabilities

LFAA LMC extends the concept of components and capabilities defined in the TM-LFAA ICD to include any internal hardware, software and logical entities. The primary components type defined in the system are:

- Hardware (racks, servers, switches, PDUs, $\cdots$)
- Software (logger, core, device controller, file manager, cluster manager, $\cdots$)
- Tiles (and stations)
- Workflows (observation modes and related jobs)

- Running tasks (beamformer, calibrator, DAQ, ⋯)

The list of components present in the systems is updated dynamically during the execution lifetime of the LMC. For example, when a new job is launched, it is represented as a new component which exposes a new list of capabilities which can be queried and/or called. Capabilities can be categorized into four types:

- **Properties** which represent values that can be set or retrieved (for example "state", which is available for all components). Several property types are defined, such as metrics, registers, values.
- **Commands** re functions defined in the component which can be called. A command may require a number of arguments, and can return several results. Typical commands include *initialise*, *check_status* and *configure*.
- **Events** generated by a component can be used to notify other components of changes. Components may register to receive events from specific components types or instances, and when the event is fired, it will be forwarded to all components registered to receive it. Several event types can be defined, for example "Component configured"and "Component status changed"
- **Data** epresents output data generated by a component which can be accessed or viewed by other components. This can be used, for example, to access raw antenna data generated by a data acquisition process

Additionally, **alarms** can be set on any property. A permissible value range can be set, and if the property' s value exceeds this range, the component itself is set in alarm state. Alternatively, an alarm value can be specified on the property, and if the current property value matches the alarm value, then it is set in alarm state. The latter behaviour can be used to automatically set a component in alarm state when its internal state matches a particular value.

Unifying Communication

The LFAA LMC core is built as a hierarchy of components, with the core itself being represented as the root of the component tree. The state of each component is an aggregate of the internal states of the underlying components. Communications between components are handled by a communication channels network with a central broker. Messages are sent to the broker which forwards them to any interested components. For example, if component A wants to read a property value from component B, it will send a request message to the broker specifying component B as the destination. The broker will decide on which channels the message will travel through and the message will eventually arrive at its destination. The same procedure is performed to send the reply from B to A. The communication system is currently based on RabbitMQ, with message formatted using JSON.

When a new component is introduced to the system (for example, during system startup) it exposes all the capabilities which can be used by other components. A software component interface is plugged into the core which is capable of communicating with this instance. The core communicates with this interface, which in turn forwards the JSON-formatted request to the component. For example, if the device controller needs to be initialised, the following steps are performed (all communication between the core and interface are direct function calls, whilst communication between the interface and the component instance happens through JSON-formatted messages over RabbitMQ on a dedicated channel):

1. The core loads the device controller component interface
2. The core calls the component initialise method
3. The interface starts the device controller component (depending on which controller the core is configured to use, for example, the TANGO server or custom scripts). The controller can be started on the same server or another server (say, by using the job scheduling interface).
4. Once the device controller is initialised it will send a notification to the interface
5. When the core receives this notification (through the interface), it will call the configure method (configure and initialise are different processes, since a component's configuration can change during its lifetime, however it does not need to be initialised each time)
6. After configuration, the core will call the get_component_capability method on the interface. This will send the JSON-formatted request to the component instance, which will generate a list of internal components, each having their own capabilities. This list is sent as a reply to the core, which places this list in the database so that other components and external entities (such as TM) can query it through the core's API.

Notes on the above:
- The list of components (and associated interfaces) which the core should load will be provided through a configuration file

- The list of hardware devices which the core should monitor and control (through the device controller) will be provided through a configuration file
- All components have a pre-defined set of capabilities, which they inherit through the generic component interface (for example, the state property and the initialise, configure and get_component_capability commands). Upon initialisation and configuration, the component can expose any other internal components and capabilities is desires.

The LMC core uses a NoSQL database (MongoDB) for internal housekeeping, such as:
- Keeping track of loaded components and their capabilities
- Storing (and possibly updating) a local copy of the Telescope Model
- Keeping track of which components are registered to which events
- Keeping track of alarms
- Temporarily queue events and command runs which need to be forwarded to external parties such as TM

Unifying Communication - Interfacing with LMC

The dynamicity of the core must be reflected in the interfacing layer, which exposes all the available functionality to third party clients, most notable of which is Telescope Manager. It is known that each LMC element will communicate with TM via a Tango interface. Beneath this Tango interface will reside a RESTful API. The list of possible URLs is generated dynamically when the core is started and throughout its lifetime, depending on the entries in the database. The URLs are designed in such a way as to make it easy to drill down, or filter, components and capabilities by specifying IDs, types and other filtering options.

REST stands for Representational State Transfer. It relies on a stateless, client-server, cacheable communication protocol (primarily HTTP). It is an architecture style for designing network applications. The primary aim is that instead of using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls instead. RESTful applications use HTTP requests to post data (create and/or update) read data (for example, to make queries) and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Write) operations. These operations are performed through the following HTTP requests:

- **GET** –Query an entity for information or data
- **POST** –Issue a command which changes the state of an entity (for example, to create an observation or write a property value)
- **PATCH** –Update the state of a created entity (for example, to stop an observation)
- **DELETE** –Delete an entity (for example, unsubscribe from receiving an event, which will delete the appropriate entry)

Notes:
- The API allows for multiple observations to be running concurrently, however currently the design of the core can only support one. When the core is extended, the API does not need to change.
- A single component instance can be specified by its component type, component name and component ID. For example Switch 5 would have component type “hardware”, component name “switch”and ID “5”.
- The client can subscribe to any event being exposed by the core and internal components. When an event is generated it will be placed in a queue for a specified time window. The client can poll for generated events using the URL above. Filtering on event type and even source is also possible.
- Transient entities like observations, command runs and events are only available for a specified time window after they are completed, after which they will be purged from the database. This timeout will be referred to as the PostTimeout.
- All API calls returning more than a pre-defined number of items will be paginated so that there is not risk of hogging the core (and client) if a request generated a large reply. The reply will include a URL which points to the next result collection.

Some usage examples:

Check component state, check the current state of all tiles:

```
GET  /components/tile/properties/state
```

Check the current state of tile 10:

```
GET  /components/tile/10/properties/state
```

Check the state of the beamforming job:

```
GET  /components/beamformer/properties/state
```

Check the temperate of all tiles in station 2:

`GET /components/station/2/properties/temperature`

Create an alarm which provides a maximum value for tile temperature (all tiles)

`POST /components/tiles/alarms`

POST will contains the property on which to set the alarm (in this case temperature) and the maximum value (JSON form).

All events and alarms alert are reported on /events. To get all alarm triggered from tiles, the following request can be used:

`GET /events/alarm\_triggered?source\_type=tiles`

To get all events for tiles:

`GET /events?source\_type=tiles`

An observation is defined by a telescope model. The external user must generate a valid model which can then be sent to the LMC by using:

`POST /observations`

POST contains the telescope model, including any software and firmware binaries which would be required. The reply will include an observation ID, which can then be used to start the observation. If an ID of 1 is received, the following can be used to start the observation:

`PATCH /observations/1/run`

PATCH includes the command to start the observation. A GET request to this URL will return information relating to the current status of the observation.

Unifying Communication - TANGO Wrapper

Given that the LMC has to interact with TM via Tango, the RESTful API has to be wrapped in a TANGO device. The TANGO community is already proposing ways for the TANGO ecosystem to include a RESTful API in future releases. So the wrapper has to be custom-developed for the time being. We currently have defined an API for LFAA, but we feel that there are ways to have this formalised across many LMCs if required. The real work is in mapping the REST API do the different sub-elements, including, but not limited to Tango. This requirement is taken care of by the LMC Core element. To follow the current devlopment on REST APIs for Tango, go to: link

Summary

This use case is a description of the current prototype of our development effort for LFAA LMC. The use case is based on our use of Tango, as well as other software packages that are involved in the LMC infrastructure. Primarily, we discuss the following:

1. Monitoring and control of digital devices and any hardware components
2. The detection of any specified alarms/events, to inform appropriate recipients
3. The running of computational/logical jobs/tasks with specified schedules
4. Routine checks and diagnostics
5. Exposing functionality in a logical fashion, or in detailed form for debugging and fault mitigation
6. Interaction with any operators/users
7. Routing, processing, saving of data streams etc.
8. Management of the available computational resources

This use case will focus on the use of TANGO within the LFAA LMC architecture, and then move outwards to show how TANGO fits within LFAA LMC, to satisfy the LMC features above.

Primary author: Dr DEMARCO, Andrea (University of Malta)

Co-author: Dr ZARB ADAMI, Kristian (University of Oxford)

Presenter: Dr DEMARCO, Andrea (University of Malta)