Software design for CSP.LMC in SKA

Marotta G.[a], Giani E.[a], Novak I.[b], Soderqvist A.[b], and Baffa C.[a]

[a]INAF Osservatorio Astrofisico di Arcetri, Florence, Italy
[b]Cosylab Switzerland, Brugg, Switzerland

ABSTRACT

The Square Kilometer Array (SKA) is devoted to the construction of a two body giant Radio-telescope. The design and implementation of the SKA Monitor and Control software involves about 100 people referring to eight Agile Teams developing different software elements of the telescope. Each of these elements is implemented as a "device" within the TANGO Control System framework, written in Python code. This paper analyzes the implemented design of the Local Monitoring and Control (LMC) of the Central Signal Processor (CSP), from now on called CSP.LMC.

CSP is the SKA element that will make the data that comes from the antennas available for scientific analysis. It is composed of different data processing components, i.e. the Correlator and Beam Former, the Pulsar Search and the Pulsar Timing. In this larger system, CSP.LMC has the role to communicate with the Telescope Manager (TM), i. e. the software front-end for operations, as if the CSP was an unique entity.

The paper shows the detailed structure of the software, implemented with an Object Oriented approach, with a design largely inspired by some standard design patterns, such as the Observer, the Command and the Aggregator. Another essential feature is the separation of the business logic from the TANGO communication layer, improving the testability and the maintainability of the code.

1. INTRODUCTION

The Square Kilometer Array (SKA) is an international effort to construct the world's biggest radio telescope. It will be composed of two body giant interferometers: the first one will observe the sky in the mid-range radio frequencies (350MHz - 15.3GHz) and will be located in South Africa, the other will observe at low-range frequencies (50MHz - 350 MHz) and will be located in Australia.[1]

SKA is one of the greatest technological challenges in contemporary astronomy and its construction requires the effort of hundreds of scientists and engineers around the world. The construction phase, started on July 1st 2021, marks the last and the most important period of its development, setting the project towards the first observations, scheduled for 2027.

During the operations, hundreds of receptors will receive a very large quantity of data, around 7.3 TB/s for Low and 8.8 Tb/s for Mid.[2] In order to obtain sensible information, the data needs to be processed before being available for scientific analysis. Figure 1 shows a simplified view of the structure of the SKA facilities. The data coming from the receptors are processed at first by a facility called the Central Signal Processor (CSP). The purpose of CSP is to correlate, filter and make a preliminary analysis. On the other side, the astronomical prescriptions for the observations come from the Telescope Manager (TM). TM is the subsystem with responsibilities for managing the observations life-cycle and for the monitoring and control of the telescope.[3] After the CSP initial data handling, data are processed by the Science Data Processor (SDP) and then are sent to the Science Regional Centers, where they will be stored and made available for the scientific analysis.

CSP is composed of three complex signal processing subsystems (Figure 2):

Further author information: Marotta G.: E-mail: gianluca.marotta@inaf.it

Software and Cyberinfrastructure for Astronomy VII, edited by Jorge Ibsen, Gianluca Chiozzi, Proc. of SPIE Vol. 12189, 121891A · © 2022 SPIE
0277-786X · doi: 10.1117/12.2630140

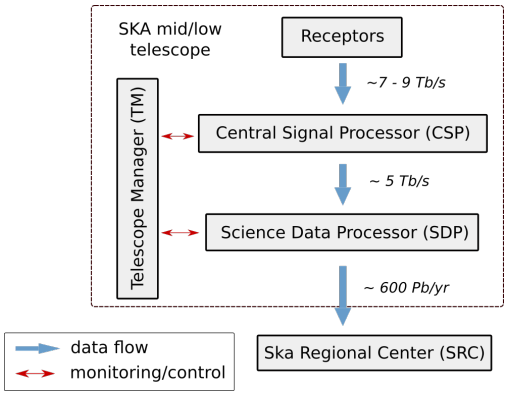

Figure 1. Simplified schema of the SKA computing facilities

- the Correlator and Beam Former (CBF), that creates the visibilities from raw data coming from the receptors and the beams to send to the other two instruments;
- the Pulsar Search (PSS), intended to perform an all-sky pulsar search survey;
- the Pulsar Timing (PST), that measures the frequency of the radiation emitted from pulsar candidates.

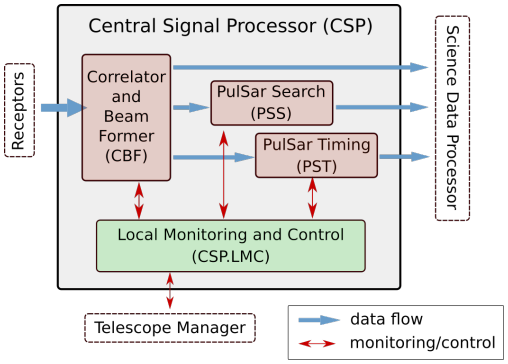

Figure 2. Simplified schema of the Central Signal Processor

2. THE CSP.LMC ROLE

The CSP Local Monitoring and Control (CSP.LMC)[4] provides to the TM the monitoring and control interface for both engineering operations and the execution of scientific observations. It makes it possible for the three CSP sub-systems to be controlled and monitored as a single instrument, without exposing the internal complexity and the implementation details. It provides two interfaces:

- an interface to monitor and control the sub-systems;
- an interface that exposes the CSP parameters and commands to TM, to execute complex functions and state transitions.

CSP.LMC coordinates the request received from TM and provides responses on behalf of the CSP. CSP.LMC also provides the TM with the availability to access information and commands implemented by the individual CSP.LMC components and sub-systems for diagnostic and debugging purposes.

To optimize the use of telescope resources, each telescope instance allows the user to divide collecting area in up to 16 *subarrays*. Each *subarray* acts as an independent instrument allowing to schedule, start and stop observations independently. The Local Monitoring and control of each CSP subsystem, including the CSP itself, provides a software interface for the subarrays, while the whole instrument is managed by a component called *controller*.

The CSP.LMC Controller provides an API for monitoring and control of the CSP instrument, including the 16 subarrays. It implements status indicators, configuration parameters and commands. It also maintains the information about the schedulable resources availability.

The CSP.LMC Subarray provides API for configuration, monitor and control of the signal processing. A CSP.LMC Subarray is directly commanded and monitored by the TM once enabled for observing purposes.

The same applies to the Controller and Subararys of the CSP sub-systems with the difference that PST, in order to achieve a simpler design, provides up to 16 instances of a PST beam that are directly accessed by the CSP.LMC. These beams are CSP schedulable resources that may be assigned to a subarray, as needed by the configured observation.

In this paper we will show the software design strategy adopted to let CSP.LMC perform its operation in a reliable and efficient way.

3. SOFTWARE ENVIRONMENT

Before explaining in more detail the CSP.LMC software structure, it's worth to spend some words about the environment on which SKA is building its software.

SKA LMC software implementation is based on the TANGO Control System Framework[5] and the SKA Control System Guidelines.[6]

An SKA "component" is every software or hardware unit that expresses a finite and well defined functionality and communicates with other components via a network interface. In TANGO CS framework, each SKA component can be modeled as a *TANGO Device* which exposes:

- a standards set of attributes and commands available via class inheritance from a set of standard TANGO classes, named *SKA Base Classes*
- a set of component specific attributes and commands, as required for each component.

Each instance of a TANGO Device runs within a *TANGO Server* a process accessible via a standard API, registered within a TANGO Facility Database.

All SKA components implemented as TANGO Devices, are containerized in order to be built in an environment that is machine-independent and standard to all of them. Templates and basic containers are provided at system level for SKA software. All of these containers are orchestrated via Kubernetes,[7] an "open source platform for managing containerized workloads and services". Kubernetes permits to develop the TANGO device servers in a clusterized way, decoupling the service by the bare metal development (when possible) and letting the system to be "high-available". In a Kubernetes system whichever failure or impossibility to reach a software component will be rescued by different clones of the software, that are managed automatically.

All components belonging to the same SKA subsystem are developed by a different Agile team. The whole structure of SKA Software is organized within the SAFe framework, a Scaled Agile solution proposed for big and complex software projects. At the base of the software development process there is the Continuous Integration/Continuous Deployment philosophy, a DevOps practice where the software is continuously deployed in a working environment and continuously tested when integrated with other components.

4. STATES AND MODES

In a large system such as SKA, a large number of different components need to be monitored and controlled. For this reason, it is important to agree on what are the States and the Modes that represent each device server. Also in this topic, a set of standards are taken inside the SKA community and all the transitions are managed by a State Machine.

A Finite *State Machine* is a computational model in which the machine can stay in a finite number of states. According to standard Design Patterns, a State is an entity on which the "object's behavior depends on [...], and it must change its behavior at run-time depending on that state".[8]

Each of the SKA States and Control Modes determines the behavior of the component to which it belongs. As we will see, the computation and update of the SKA state and control modes is one of the most important tasks that CSP.LMC needs to perform in order to let TM aware of the available operations on the CSP tool.

The base code of each SKA component provides all of those that need to be reported as TANGO attributes. Between them, the most important are:

- the *State*, representing the overall state of the system, it can be ON, OFF, STANDBY or FAULT;
- the *ObsState*, that indicates the status of an observation; typical values are EMPTY, IDLE, READY, CONFIGURING, SCANNING ecc. . . ;
- the *HealtState*, indicating the overall status of the component, OK, DEGRADED, FAILED;
- the *AdminMode*, that gives information about the operativity of the system, such as MAINTENANCE, ONLINE, OFFLINE,
- the *ObsMode*, that states what is the main purpose of the ongoing observation.

5. CSP.LMC SOFTWARE ARCHITECTURE

The use of Tango CS gives the SKA software an intrinsic overall hierarchical structure. General approach is that CSP.LMC acts as a server towards TM and as a client towards CSP sub-systems. Components belonging to the higher levels of the hierarchy control and configure components lower in the hierarchy. On the other side, the devices that are in the lower levels of the hierarchical structure, use events to report changes in their internal states or mode transitions, alarms and other events of interest to the higher devices.[9] CSP.LMC subscribes for these events to receive the updates.

Also the CSP.LMC monitor function is hierarchical: each layer in the hierarchy intelligently rolls-up the status of subordinate equipment, and software components and reports overall status. The top hierarchy layer, implemented by CSP.LMC, reports the overall CSP status indicators such as the operational and observing states, the health status, the observing mode. Error and fault conditions are reported by the lower devices in the same way.

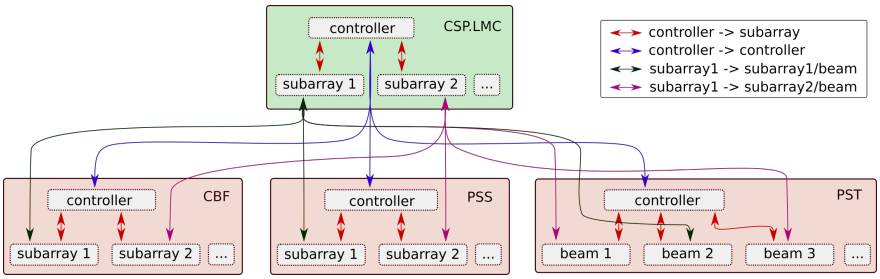

Figure 3. Schema for the monitoring and control inside the CSP

Figure 3 shows a schema that represents the complexity of the monitoring and control inside the CSP. As a first thing, each *controller* monitors and controls all the subarrays belonging to the same CSP component (red arrows) while, on the other side, CSP.LMC controller monitors also all the subsystem's controllers (blue arrows). Similar behavior for CSP.LMC subarrays: each subarray controls the corresponding subarray present in CBF and PSS. Since PST has not subarrays, each CSP.LMC subarray will also monitor the PST beam assigned to it. In the example of Figure 3, CSP.LMC subarray 1 will monitor PST beam 1 and 2, subarray 2 will monitor beam 3.

6. SOFTWARE STRUCTURE OF CSP.LMC

The CSP monitoring and control functionalities are largely inspired by standard design patterns and by a object-oriented programming approach. In fact, every object is designed to have a specific and well-defined purpose, enhancing the readability of the code, the simplicity of maintenance and the testability of the software.

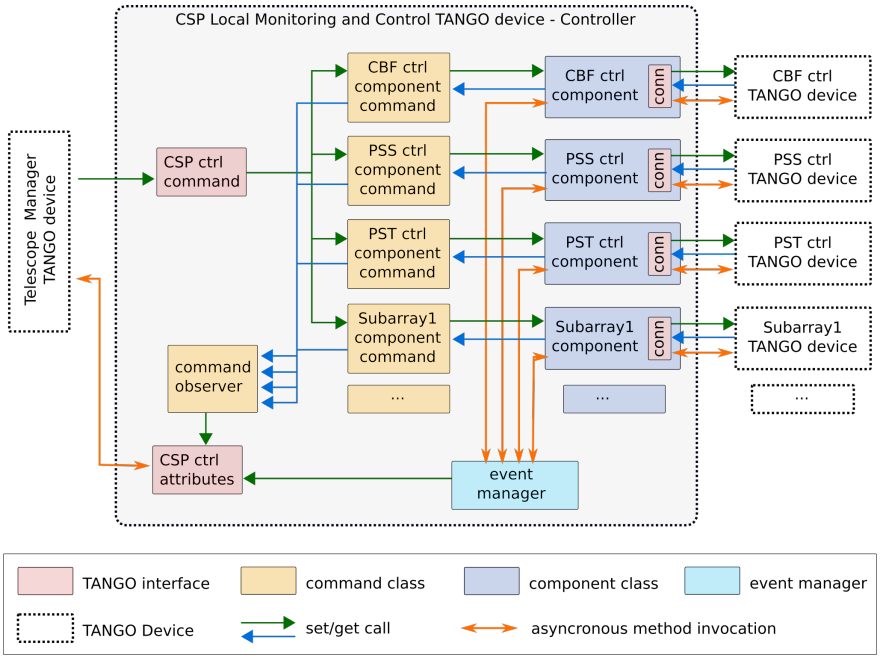

Figure 4. Simplified Component and Connector diagram for CSP.LMC

Figure 4 represents the simplified Component and Connector diagram for CSP.LMC. In the figure, the Controller structure is provided. The same structure applies also for the Subarray, with the difference of the subsystems controlled (See Figure 3).

The most external layer is represented by the Tango interface (pink boxes). From the client point of view, the TM, CSP.LMC exposes as an interface a series of *commands* and *attributes*, as defined in the previous section. From the other side, CSP.LMC acts as a client, connecting to its subsystems acting as Servers. To perform this connection, a class called *Component* was developed. It acts as an adapter and allows the execution of specific instructions on a subsystem before invoking the required command. The last layer towards the subsystems' TANGO Device is provided by the class called *Connector*. One of the main advantages to having this class is the possibility of it being easily mocked during the tests. Both CSP.LMC internal and external interfaces are implemented as TANGO API by the top level CSP.LMC software components. In general TM acts as a client, while CSP.LMC acts as a server.

Between the TANGO layers, from the TM to the subsystems, relies most of the CSP.LMC business logic. As we will see, the monitoring and control can be summarized in the issuing of commands and the

management of events. To perform these operations, dedicated objects exploit specialized duties. In the following we will explore these two functionalities and how some standard design patterns have helped the software design.

7. COMMANDS

One of the base functionalities of a TANGO Device is the possibility to issue *commands*, i.e. specific operations performed by the device. The Tango Core libraries allow the user to perform commands in both *synchronous* and *asynchronous* ways. A command is called *synchronous* when it blocks the execution of the Client that is issuing it. On the other hand, a command is called *asynchronous* when it is executed in a concurrent way. Therefore, the Device is able to perform other operations and can respond to client requests during the execution of the command.

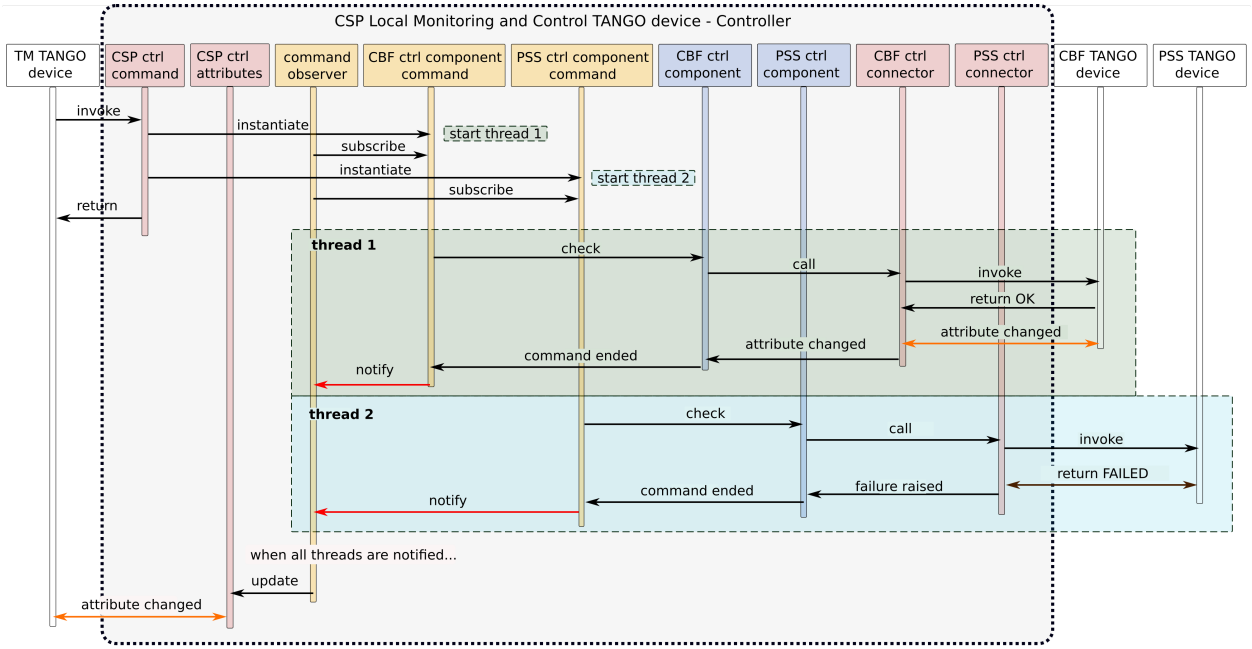

Figure 5. Example of typical sequence for CSP.LMC command

The role of CSP.LMC in the execution of the commands is mostly to forward the request arriving from the TM to its subsystems. A typical sequence of operation is shown in Figure 5. For clarity, only CBF and PSS controllers are reported. Let's consider, as an example, that an *Assign Resources* Command is requested from the TM on CSP.LMC Subarray 1. This observational command is meant to assign the subarray with the necessary resources, such as the receptors, the correlators of the CBF or the pulsar search beams. The first thing that a TANGO device does when a command is issued is to check if some conditions are reached. In the SKA case, this means that the component is in the proper set of states and modes. After the check, the thing to be done is to split the request for the resources into the sub-requests addressed to the proper subsystem. This can be done directly from the Tango Command Class, or can be done in a more specific way by an operative class called *Component Manager*. For the purpose of this paper it is not important who is in charge of doing the "splitting". After the client's invocation, the *Command* class can report immediately that the command was received (or in other words it is started). From a TANGO point of view, this means that the command can be issued also in a synchronous way, since the Device will be blocked for a limited amount of time. In fact, as soon as the command is received, CSP.LMC starts a thread for each subsystem, in order to run the requested operations, and be available for other client requests.

The threads are started from an instance of a specific class, called *Component Command*. The Component Command models a command as acting on a sub-system Component instance. It implements the logic to manage and control the command issued on a single component. It contains all the information about the request, such as the input parameters, the Component to act on and the conditions for success, failure and timeout. It accesses the corresponding Component class that sends, via the Connector class, the TANGO request to the corresponding Devices. The Component Command, accessing the Component class, it also checks that basic functionalities for the corresponding subsystem are compliant to the issuing of the command. As an example, it will check that the component's internal representation reports it as online and no failures are raised on it, before sending the command to the actual subsystem.

Since the commands are executed in a concurrent way and on different subsystems, CSP.LMC needs a mechanism to collect the information from the commands threads and update some TANGO attributes about the command's execution. This task is performed by the *Command Observer*. The idea of having such an object derives directly from the standard design patterns. In fact, its implementation is largely inspired by the *Observer* pattern, that is a behavioral design pattern whose intent is defined as "one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically".[8] In our case, the "one-to-many dependency" is defined from the CSP.LMC to its subsystem, so the changing of the dependent components is notified to the one that is controlling them.

The basic concept behind this pattern is the *publish-subscribe* mechanism. In practice, a method from the Command Observer, i.e. a "callback", is registered inside the *Component Command* at the command invocation, before returning to the client. After the end of the command execution, the callback is called inside the thread, letting the observer know that the command on that subsystem is terminated and what its status is (succeeded, failed, timed out). After all the threads are finished, the Observer has all the information about the command and can update the client about the end of the command and its status.

To understand better the mechanism of this design, it's useful to refer to the previous example with reference to Fig. 5. Let's consider that the assignment of resources is successful on CBF while the command fails on the PSS for some external reason. The thread of the CBF command (thread 1 in Fig. 5) will get a return statement from the CBF TANGO server that says that the command is STARTED. This could mean that some concurrent mechanism for the command was started and the device is available again, in a similar way to the one of the CSP.LMC. In order to be sure that the subsystem has performed the required task, the value of a TANGO attribute needs to be checked. Also in this case, a subscription mechanism, that we will discuss in the next section, allows the CSP.LMC connector to detect the change of the required attribute. This attribute could be specific to report the status of the command or it could be one of the SKA states and modes. In the example of the assignment of resources, the transition of the *Operational State* from *EMPTY* to *IDLE*, provided from a suitable *state machine*, ensures that the assignment of resources is successful. A set of calls send the information of the success back to the *component command* that "notify" the command observer. On the other hand, the command could fail or time out on a subsystem. Even in this case, the information is propagated back to the *Component Command* and then notified to the *Command Observer*.

When all the sub-commands are concluded and notified to the Command Observer, the latter will update an ad-hoc attribute to inform clients about the execution of the command. In principle, this could be done with a specific attribute or triggering the state machine to act the proper transition. In the current implementation an attribute called *Command Result* is updated, reporting the latest command executed with a "Result Code" reporting the status of the command (STARTED, OK, FAILED). To not trigger the state machine was a specific architectural decision for CSP.LMC. In fact, as we will see in the next section, the SKA state and modes are managed by a different code functionality, called *Event Monitoring*.

8. EVENT MONITORING

As told before, important information about an SKA software component is given by their states and modes. They are valuable information regarding the component readiness, the possibility of issuing commands and performing observation.

In the case of the CSP.LMC, these attributes refer to the whole CSP tool and need to represent its overall functionalities. Anyway, also CSP subsystems have their own states and modes. The purpose of the CSP.LMC is to report on the CSP as a whole instrument to the TM. This is why it also has to report the state and modes in a way that is consistent with subsystems' one. Furthermore, it has to detect quickly any change on them, due to external reasons.

We call "event" whatever changes on subsystem states and modes. *Event monitoring* refers to the CSP.LMC feature to properly detect events and change its attributes accordingly. To do this, a publish/subscribe mechanism is adopted in a similar way to the one used in command execution.

Even in this case a common software design pattern is used. The *Aggregator* pattern.[10] It is defined as a "a stateful filter [..] to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages."

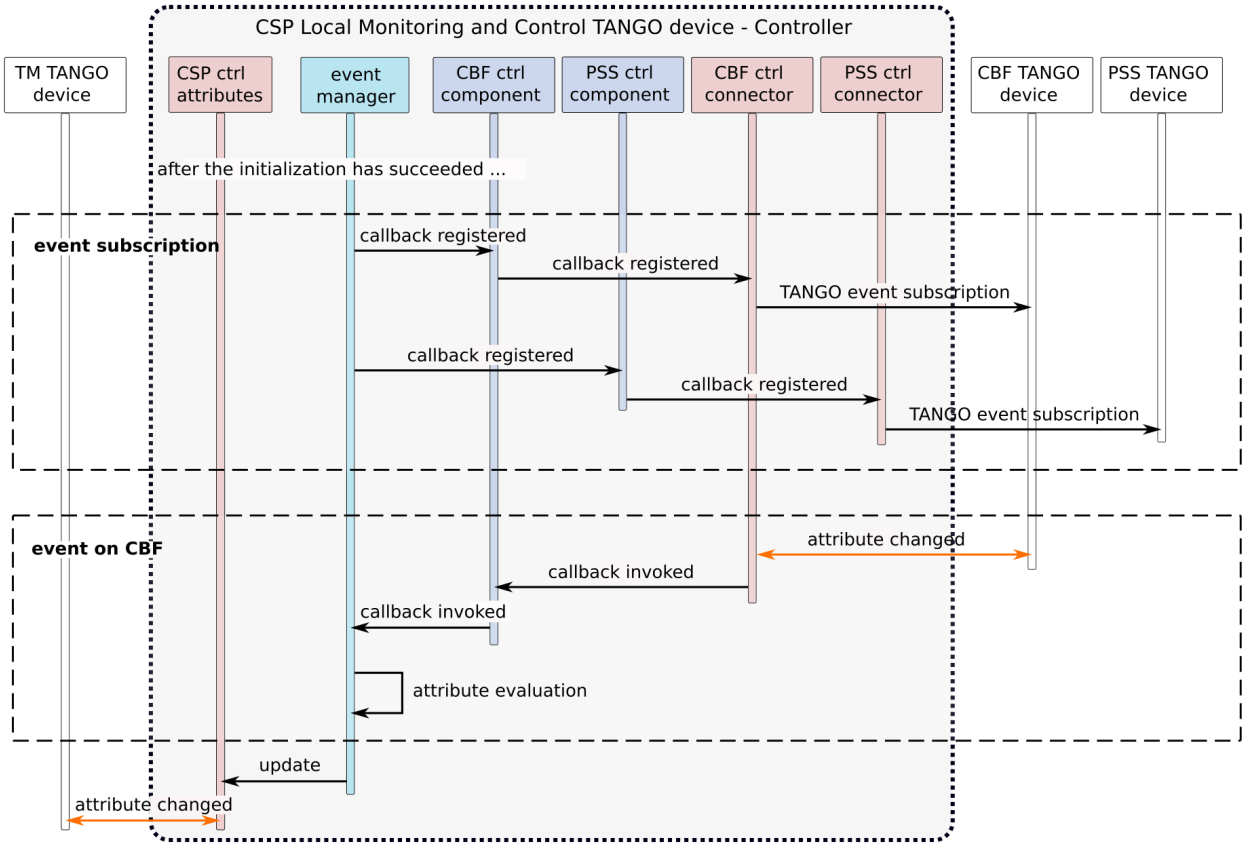

Figure 6. Example of typical sequence for the aggregation of CSP.LMC events

The class devoted to Event Monitoring is the *Event Manager*. Figure 6 shows an example of a sequence diagram for the event monitoring.

After the initialization has succeeded, i.e. all the subsystems are connected and pingable from CSP.LMC, the Event Manager registers a callback to all the Component classes representing the subsystems. From their side, the Components subscribe, via the Connector, on the corresponding TANGO device, using a TANGO mechanism, called *event subscription*. This is the same mechanism that detects the change of attributes in the case of Commands, as explained in the previous section. At this point, the series of callbacks that were previously registered are invoked, notifying the Event Manager of the change.

The arrival of this "message" triggers the evaluation of CSP.LMC state and modes inside the Event Manager. In fact, there is a set of rules inside this object that are compliant with SKA guidelines and combine the states and modes in order to create the ones of the CSP.LMC. Furthermore, the Event Manager has the memory of all the messages that have income before. It is also worth mentioning that the same subscription of events is considered an event itself, so the Event Manager will have the knowledge of all the attributes subscribed since the initialization of the device.

As an example, let's assume that the PSS is having some problems during the configuration. The expected transition for the ObsState would have been from IDLE to READY, but it went to FAULT. On the other hand, everything went well for the CBF. According to the observation's schedule, the main purpose of that observation was to perform an image of the sky, while the PulsarSearch was only an "ancillary operation" to be performed in the meantime. The telescope user could want to make the observation in any case. In this case the Event Manager will update CSP.LMC ObsState to READY, and the HealthState to DEGRADED. This allows the Scan Command to be issued, but TM will also know that the general health of the CSP has some problems. A different situation would be if CBF ObsState transitioned to FAULT. In this case no observation would be possible and CSP.LMC will report its ObsState to FAULT and its HealthState to FAILED.

9. CONCLUSIONS

In this paper we've seen how the Monitoring and Control software for the Central Signal Processor, a crucial part of both SKA Mid and Low Telescopes, has been designed. A dedicated component, the CSP.LMC handles his three subsystems to report to the Telescope Manager acting as a single instrument. The main responsibilities of this software are:

- the issuing of commands to subsystems (control);
- the monitoring of the events generated by changes in subsystems status (monitor).

The first task is implemented by a design that is inspired by the *Observer* standard design pattern.

The second task is performed by an object called the *Event Manager*. The change of the attributes subscribed on the subsystems trigger this class that calculates the corresponding states and modes for CSP.

The use of design patterns has permitted the design of the code in an object-oriented way. This structure also simplifies the testing possibilities, each object can be tested in isolation and connection with the outside can be mocked.

The future implementations of more complex functionalities, as the handling of entities comprising HW, SW and configuration, will be clearly simplified by those approaches.

REFERENCES

[1] "Skao website - telescopes infosheet." https://www.skatelescope.org/wp-content/uploads/2018/08/16231-factsheet-telescopes-v71.pdf. (Accessed: 28 June 2022).

[2] "Skao website - operational model infosheet." https://www.skatelescope.org/wp-content/uploads/2018/08/16231-Factsheets-operational-model-v4.pdf. (Accessed: 28 June 2022).

[3] Natarajana, S., Barbosa, D., Barraca, J. P., Bridger, A., Choudhury, S. R., Di Carlo, M., Dolci, M., Gupta, Y., Guzman, J., Van den Heever, L., LeRoux, G., Nicol, M., Patil, M., Smareglia, R., Swart, P., Thompson, R., Vrcic, S., and Williams, S., "Ska telescope manager (tm): status and architecture overview," in [*Proceedings of SPIE Astronomical Telescopes + Instrumentation*], International Society for Optics and Photonics, SPIE (2016).

[4] Baffa, C., Giani, E., Vrcic, S., and Vela Nuñez, M., "Ska csp controls: technological challenges," in [*Proceedings of SPIE Astronomical Telescopes + Instrumentation*], International Society for Optics and Photonics, SPIE (2016).

[5] "Tango-controls framework." https://www.tango-controls.org/. (Accessed: 28 June 2022).
[6] Van den Heever, L., De Marco, A., Pivetta, L., Riggi, S., and Vrcic, S., "Ska1 control system guidelines," (2021).
[7] "Kubernetes." https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (Accessed: 28 June 2022).
[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., [*Design Patterns. Elements of Reusable Object-Oriented Software*], Addison-Wesley (1995).
[9] Baffa, C., Giani, E., and Vela Nuñez, M., "Ska monitor and control: Harmonization challenges," in [*proceedings of Astronomical Data Analysis Software and Systems XXVI*], Astronomical Society of the Pacific (2019).
[10] Hohpe, G. and Woolf, B., [*Enterprise Integration Patterns. Designing, building and deploying messaging solutions*], Addison-Wesley Signature Series (2003).