LARGE SYNOPTIC SURVEY TELESCOPE

## Large Synoptic Survey Telescope (LSST)
## Telescope & Site

# Control Software Architecture

**Tiago Ribeiro, William O'Mullane, Tim Axelrod, Dave Mills**

**LSE-150**

**Latest Revision: 2019-02-25**



## Abstract

TSS Architecture and approach.

# Change Record

| Version | Date | Description | Owner name |
|---|---|---|---|
| 1 | 2012-12-14 | V1 | German Schumacher |
| | 2019-01-20 | Unreleased. | William O'Mullane, Tim Axelrod |
| | 2019-02-01 | Unreleased. | Tiago Ribeiro |
| | 2019-02-11 | Unreleased. Adds comments from reviewers. Add missing Figure and update AT architecture figure. Re-organize sections a bit, added OCS section. Writeup of Software Deployment Strategy. | Tiago Ribeiro |

# Contents

# Control Software Architecture

## 1 Introduction

The LSST Control Software contains the overall control aspects of the survey and the telescope including the computers, network, communication and software infrastructure. It contains all work required to design, code, test and integrate, in the lab and in the field, the high level coordination software.

## 2 System Architecture

The LSST control system is based on a reactive data-driven actor-based architecture that uses a multi cast Data Distribution Service (DDS) messaging protocol middleware. A high level view of this architecture is given in Figure 1, where each box corresponds to a component of the system (not all components are displayed here).

The LSST System Architecture is comprised mainly of;

- The Service Abstraction Layer (SAL[1]) communication middleware. Based on the DDS protocol, it provides interfaces for all the project adopted programming languages (LabView, C++, Java and Python).
- Engineering and Facility Database (EFD).
- SAL-aware reactive components, a.k.a Commandable SAL Components (CSCs).
- LSST Operators Visualization Environment (LOVE).

The SAL middleware is the backbone of the LSST system architecture. It is a high level layer on top of Data Distribution Service (DDS), a standard message passing system. LSST uses the PrismTech OpenSplice DDS library, community edition. It implements three distinct types of messages; Commands, Events and Telemetry, with distinct purposes. Commands are sent to a specific component, which must acknowledge its receipt and perform some action. In general, the receiving component will be the only entity listening for the commands it accepts[2]. Events and Telemetry are messages broadcast by components to the middleware

[1] https://docushare.lsstcorp.org/docushare/dsweb/Get/Document-21527/
[2] But note that the EFD, for instance, will also be listening for commands, though it will not acknowledge them.

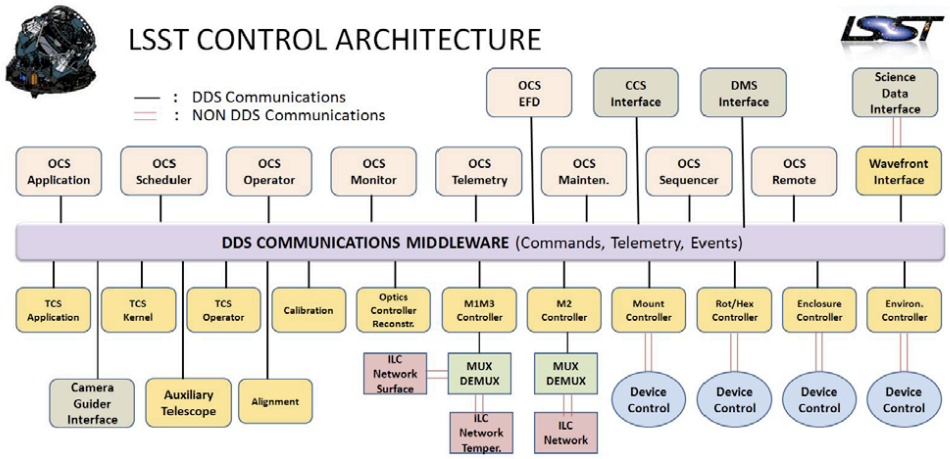


FIGURE 1: High Level Architecture Diagram. (To be replaced...)

and are available to any entity on the system to receive. The distinction between Events and Telemetry is that Events are output when conditions change, whereas Telemetry is output at semi-regular intervals. As such, it is much more important that Events be transmitted reliably than Telemetry. We cannot afford to lose any events, but we can lose occasional Telemetry. Thus Events are sent using a higher Quality of Service (QoS).

The EFD is responsible for capturing all SAL messages broadcasts to the middleware (including Commands, Events and Telemetry) and storing that information into a database.

CSCs are the main actors of the LSST system architecture. They are responsible for managing the incoming traffic of data and take appropriate actions, controlling hardware (e.g. M1M3, M2, Mount Controller, etc in Fig. 1), software (e.g. Optics Controller Reconstructor, DMCS Interface, etc in Fig. 1) or even other CSCs (e.g. ScriptQueue, TCS, ATCS, OCS, etc in Fig. 1 ).

LOVE is responsible for capturing SAL messages and displaying them in a useful way for general users, providing some basic interface to query and analyze data from the EFD, an interface to issue pre-defined commands to a set of components and user interaction with the ScriptQueue (see Sect. 3.1).

The SAL processes XML based definitions of the Commands, Events, and Telemetry for each CSC. Using this information, it creates runtime objects which support the messaging required. These take the form of shared libraries (C++, Python, LabVIEW) or Jar archives (Java) which implement consistent namespaces and API's. Other assets such as Simulated data, Sql table

definitions, and web based documentation, may also be generated. On top of these low level APIs, developers have access to two higher-level set of frameworks; Python SalObj[3] library and the LabVIEW component template. No higher level framework is supported for implementations in Java or C++.

Overall, the system architecture can be divided into three main namespaces; Observatory, Main Telescope (MT) and Auxiliary Telescope (AT). The Observatory is the highest level and encapsulates both the Main Telescope, Auxiliary Telescope and global components such as the weather station, DIMM, etc. The complete set of components that belong to each of these namespaces can be seen in Figs. 2, 3 and 4.

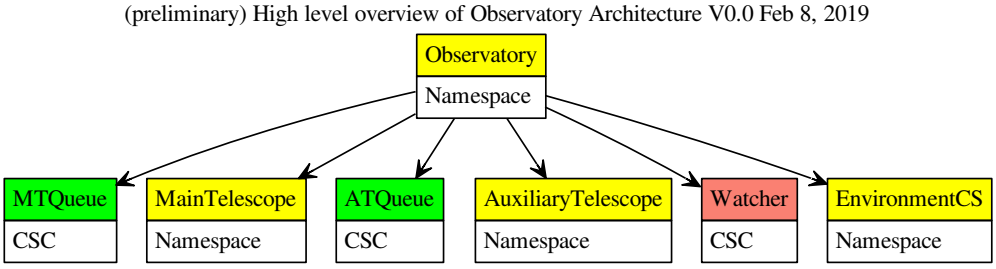


FIGURE 2: Hi level observatory architecture with namespaces and observatory-wide CSCs. (preliminary)

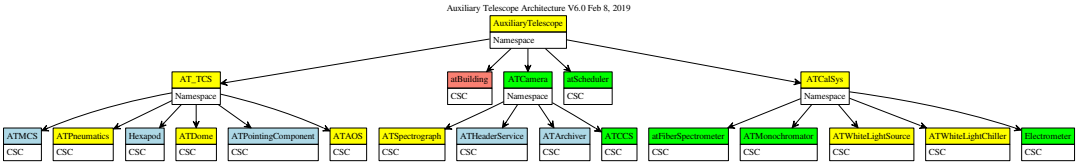


FIGURE 3: Complete set of AT CSCs (preliminary)

## 2.1 SalObj - Python and scripting

provides a high level interface

SalObj is a Python library provides a pythonic and object-oriented interface for SAL compo-

[3] https://github.com/lsst-ts/ts_salobj

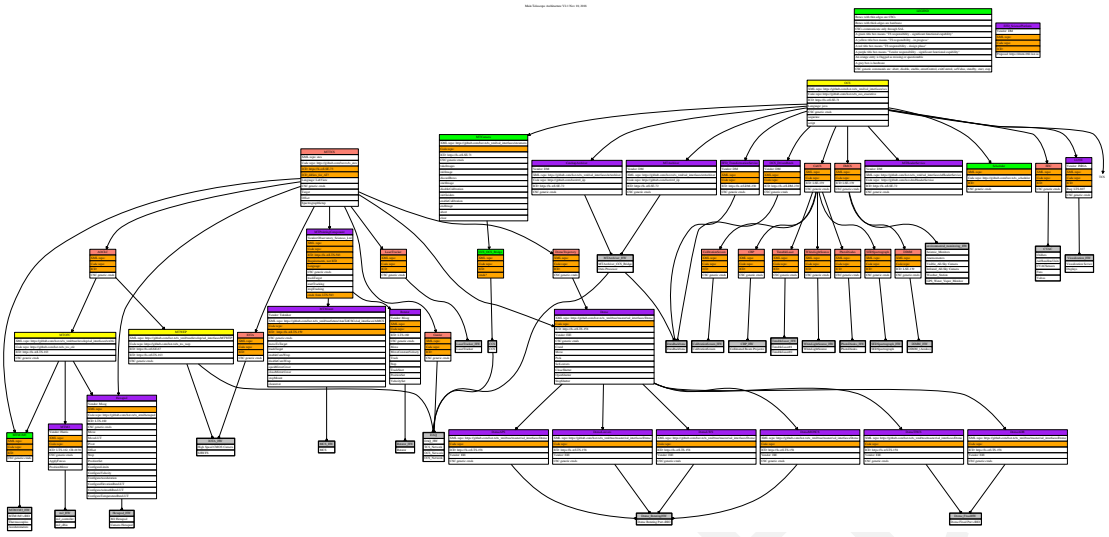

FIGURE 4: Complete set of MT CSCs (preliminary)

nents such as CSCs and SAL scripts (see Sect. 3.1). The library defines two sets of base classes that are mirror to each other, Remote and Controller. A Remote will send commands to and receive telemetry and events from a specific component whereas a Controller will receive commands and publish telemetry and events. SalObj also provides BaseCsc, a subclass of Controller that handles the standard state transitions and is intended to be used as a parent class for CSC.

Internally, SalObj uses the python library `asyncio`[4] to handle the inherently asynchronous nature of the SAL messaging system.

## 2.2 Hardware interface components

Probably the most critical or sensitive components of the LSST system architecture are those that directly control hardware. Some of these components are going to be delivered directly by external vendors, such as those that will control the main telescope mount (MTMount) and the main telescope secondary mirror (MTM2). There are also those that are developed in house, e.g. the main telescope M1M3 (MTM1M3).

In some special cases, where fast real time response is required, it is highly desirable that the control software and hardware are part of an integrated system. For those systems, the

[4] `https://docs.python.org/3/library/asyncio.html`

components are developed either using the LabView component template, which is part of the LSST infrastructure or in C++ developed using the low level SAL API.

In most other cases, the hardware comes with control software that can be easily interfaced by using standard protocols (such as TCP/IP or serial ports), and there is no special need for the component software to reside close to the low level hardware controller. In those cases, the components are written in Python using the SalObj library which is also part of the LSST infrastructure. By writing these components using a unified language and library (Python+SalObj) we allow a high level of flexibility and maintainability of the software and considerably decrease the development cycle.

## 2.3 Pure software components

In the LSST System Architecture there are a number of components that, even though they do not control hardware directly, dictate what hardware components are supposed to do. Some of these components are responsible for heavy computational routines, such as the Optical Feedback Control (MTOFC), which is responsible for applying corrections to both M1M3, M2 and hexapod components for the main telescope or even the Scheduler, which is responsible for processing an entire set of observatory telemetry information and history of observations to compute an observing queue.

These pure software components are mostly written in Python using SalObj library. There are three special cases of these components that form the basis of the LSST System Architecture; the ScriptQueue (Sect. 3.1), Control Systems (Sect. 3.3) and the Watcher (Sect. 3.2). Together, they provide the tools needed for integration, commissioning and operation of the observatory.

## 2.4 Configuration Management

During commissioning and operations, LSST will have a large number of running software components under the purview of DM, Camera, and TSS. In general, the behavior of each of these components is modifiable through configuration information which is read in during startup of the component, or possibly changed while the component is running. Careful management of this configuration information is crucial to reliable functioning of the Observatory, and to the analysis of its data products.

In this context, git is the solution adopted to store and manage different sets of configurations

and different versions of configurations. Git is already a standard in industry as a software management tool and has becoming increasingly used to manage general documents and files as well, not to mention that it is already readily available and broadly adopted by the project. Therefore, each component must be capable of handling a git-based configuration repository.

These configuration repositories will be hosted on a configuration server at the summit so that, even if communication with the base or the internet is not available, components still maintain access to their configuration repositories. Different configuration sets (or labels) are stored in separate branches, and tags can be created to specify immutable sets.

Several options for configuration file format, and their associated software tools, have been considered. Each of the available options naturally has its strengths and weaknesses, and none stand out as being particularly useful for all LSST use cases (and/or available for all the project adopted programming languages).

For components written in Python, `pex_config` is the adopted solution. As an overview of `pex_config`, here are a few snippets from the library documentation:

> The `lsst.pex.config` module provides a configuration system for the LSST Science Pipelines.... Configurations are hierarchical trees of parameters used to control the execution of code.... Configurations are stored in instances of a "config" object, which are subclasses of the `Config` class. Different configuration sets are associated with different subclasses of `Config`. For example, in the task framework each task is associated with a specific `Config` subclass.... Configuration objects have `fields` that are discrete settings. These `fields` are attributes on config classes.

In the TSS context the "task" above becomes a "CSC".

`lsst.pex.config.Config` class has methods `save()` and `load()` which persist and restore class instances from files, which just contain Python code. Note that because these files are Python code, it is easy to include documentation within the files.

The validation of a `Config` file is handled by the `__init__()` method of the `Config` subclass which can check, for example, whether parameter values comply with range limits, or whether all required parameters are specified.

In this framework, the configuration schema (or definition) will be developed and stored in the CSC codebase repository (using `pex_config`). As already stated above, a separate repository hosts the actual configuration files which, in the case of `pex_config`, only contains changes to the default set of values.

# 3 Observatory Control System

The LSST Observatory Control System consist of a collection of specialized components, namely; LOVE, the ScriptQueue, the Watcher and Control Systems. This distributed control system is designed to efficiently and safely perform astronomical observations individually or through automated scheduling. In this section we describe the role each of those components play in enabling the LSST observatory operations.

## 3.1 The ScriptQueue component

There are a number of different ways users can interact with components in the LSST system. For instance, one could easily use the SAL generated API in any of the supported languages to send commands directly to a single or multiple components. It is also possible to use SalObj Remotes to write Python scripts (e.g. SAL Scripts) that would command different components to accomplish a specified task. Not to mention that LOVE itself provides a customizable interface for users to interact with components.

During commissioning and operations the LSST system will require a high degree of coordination between different crews (different daytime and nighttime shifts, for instance), not to mention the increasing number of available components and level of complexity as the system ramps up. In order to manager those issues, the LSST control system contains a specialized script queuing component, a.k.a. the ScriptQueue[5].

The ScriptQueue defines BaseScript a Python base class which provides an interface for developing SAL scripts. As Python programs, these scripts have access to all Python functionality, both from the native Python 3 language and through imported modules (including `asyncio` to manage concurrent activities or libraries from the DM stack). In particular, a SAL Script has access to all the system components using SalObj Remotes (Section 2.1) . Although Python is the only language officially supported, scripts can be written in any SAL-supported language. As long as they follow the interface defined by the ScriptQueue component, it should be possible

[5] `https://github.com/lsst-ts/ts_scriptqueue`

to execute them.

## 3.2 The Watcher

The Watcher is a component that monitors the other SAL components and output alarms in a standard way that LOVE can present to operators. The Watcher is designed in such a way that alarm rules are easy to write and easy to understand. The rules are likely to evolve rapidly during commissioning and slowly after that.

Examples of alarms published by the Watcher are;

- Dangerous weather, such as rain or high humidity.
- A SAL component is unavailable: not enabled or heartbeat is missing.
- Actuator malfunction, such as axis motors out of closed loop, filter changer stuck, an actuator hits a limit.
- CCD temperatures or pressures out of range.

A typical life cycle of an alarm:

1. Azimuth goes out of range so the controller halts motion. The Watcher reports this as an alarm with severity=serious. LOVE displays it.
2. An operator acknowledges the alarm to the Watcher, but the axis is still out of range. The Watcher outputs a new version of the alarm that includes the information that the alarm has been acknowledged. LOVE displays the alert in a way that looks less urgent (e.g. is grayed out). The alarm has been acknowledged but the condition is still current.
3. An operator fixes the problem and the controller reports this. The Watcher reports the alarm one last time with severity ”OK”. LOVE removes the alarm from the display.

A typical life cycle of a transient alarm:

1. The azimuth drive temporarily draws too much current; the component reports this but manages to keep the axis moving (presumably with temporarily degraded accuracy). The Watcher reports this as an alarm with severity=serious. LOVE displays it.

2. The drive current is within normal range again before an operator has time to acknowledge the alarm. The Watcher outputs a new version of the alarm that says the condition is now OK but the alarm has not yet been acknowledged (a "stale alarm"). LOVE still displays the alarm, but in different way to indicate that the problem is gone.

3. An operator acknowledges the alarm to the Watcher. The Watcher outputs a new version of the alarm with severity "OK" and acknowledged=True. LOVE removes the alarm from the display.

### 3.3 High level Control Systems

Given the distributed nature of the LSST system architecture it is not immediately clear that a traditional hierarchical design, with centralized Control System, is necessary or even desirable. A completely flat architecture seems completely workable and certainly sufficient during AIV and early commissioning. For example, consider a SAL Script which commands and sequences the telescope subsystems to move to the next field to be observed, take an exposure, and read out that exposure. The SAL Script can directly control each of those subsystems and maintain control of the sequencing using `asyncio`. Furthermore, the complexity of the SAL Script can be managed through normal modular programming techniques, in which subsystem functionality is implemented through Python objects imported in modules.

As the system matures and knowledge is gathered about the intricate interdependencies of the various subsystems, it is possible to realize that high level components, constantly monitoring the state of the observatory, can be responsible for some autonomous actions to safeguard operations. It is also possible to envision that some actions involving multiple components (initially developed and conducted by SAL Scripts) can be incorporated to one of these components. Henceforth, the role high level Control Systems will play in the Observatory Control System is yet to be defined once the system has matured enough.

## 4 Software Deployment Strategy

The LSST software deployment strategy follows a continuous integration (CI) process to support development all the way to deployment, employing industry standard tools. Figure 5 shows a diagram with the process. This process applies to all software component of the system infrastructure, from SAL and CSCs (regardless of the programing language they are written in) to SAL Scripts and all the other libraries.

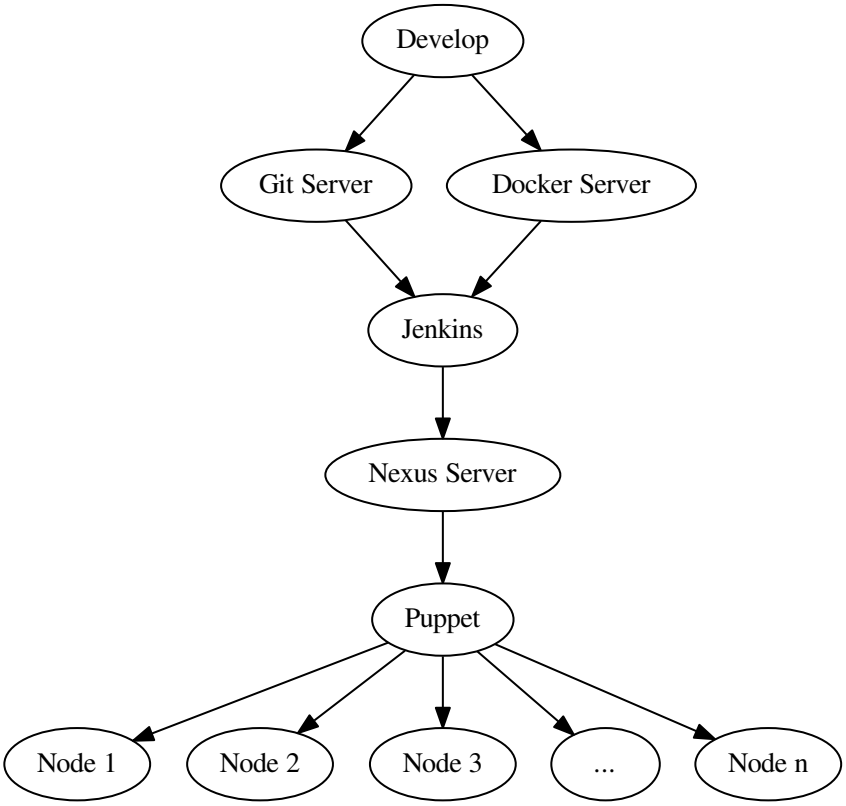


FIGURE 5: Diagram outlining the software deployment strategy.

As one can see from Fig. 5, the main end solution to the deployment strategy is the use of Puppet. Puppet is an open source systems management tool for centralized and automated configuration management. The main idea behind this service is that it is possible to describe the system architecture (e.g. how the ”system should be”) in a configuration file. Then the server is capable of configuring each node with the appropriate software.

The choice of a ”configuration management” or ”deployment system” (e.g. Puppet) still leaves open the question of how the software is packaged. One of the largest growing and broadly used industry-standard solutions in use for distributed systems like the LSST, is Docker. Docker is a container solution for software deployment, which packs code and all its dependencies into a lightweight virtual machine-like environment. It also runs quickly and reliably from one computing environment to another.

Development is the first stage of the process and is where code is either created (e.g. new CSCs, new scripts being developed, etc) or modified (e.g. bug fixes, improvements, etc). Once development is completed and the software is tested and validated it goes to the Git Server. At this stage, a Docker image may also be created and stored in the Docker Server. Once this is completed a Jenkins build is triggered. For the build, Jenkins will pull the software, along with all its dependencies, build and run unit and integration tests. If all tests passes, the software is then packed by Jenkins; which may be a new Docker image with the new version of the code, an RPM package or some other package method that can be used by Puppet. The software package is then sent to the Nexus Server.

In some cases, the resulting Docker image to contain the software may exceed Jenkins build size limit, and it is not capable of creating images for testing and deployment to the Nexus Server. In these cases, the developer creates the Docker image and place it in the Docker Server. Jenkins will pull the Docker image, start it locally and run the unit and integrations tests. If the test passes, Jenkins pushes the image to the Nexus Server.

Once the final version of the software is packed and stored in the Nexus Server, Puppet can be instructed to update the software on a specific node (or nodes).

# A References

# References

# B Acronyms used in this document

| Acronym | Description |
|---|---|
| AIV | Assembly, Integration, and Verification |
| API | Application Programming Interface |
| AT | Auxiliary Telescope |
| ATCS | Auxiliary Telescope Control System |
| C | Specific programming language (also called ANSI-C) |
| CCD | Charge-Coupled Device |
| CI | Continuous Integration |
| CSC | Controlable SAL Component |
| DIMM | Differential Image Motion Monitor |
| DM | Data Management |
| DMCS | Data Management Control System |
| EFD | Engineering Facilities Database |
| IP | Internet Protocol |
| LOVE | LSST Operations Visualization Environment |
| LSE | LSST Systems Engineering (Document Handle) |
| LSST | Large Synoptic Survey Telescope |
| M1M3 | Primary/Tertiary mirror |
| M2 | Secondary mirror |
| MT | Main Telescope |
| OCS | Observatory Control System |
| RPM | RPM Package Manager |
| SAL | Services Access Layer |
| TCS | Telescope Control System |
| TS | Test Specification |
| TSS | Telescope and Site Software |
| XML | eXtensible Markup Language |