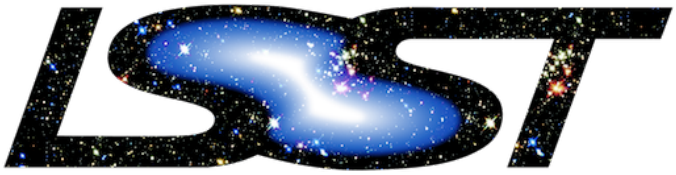

LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)
Telescope & Site

Proposal to conduct in-house CSC development.

Tiago Ribeiro and Patrick Ingraham

TSTN-001

Latest Revision: 2019-08-26

Abstract

This documents presents a proposal to have vendors, notably for the TMA and Dome, focus on the development of low-level functionality of their component and de-scope development of CSCs, resulting in in-house development using Python/SalObj. Although it appears as an increase in scope to the T&S software (TSSW) team, previous experience has demonstrated that a lack of strict requirements and guidance on CSC development has yielded unfavorable results that have become challenging to understand, maintain, deploy and upgrade. In-house development will ensures uniformity across CSCs and will provide the required functionality. The timing to pursue such changes is important as CSC development for the TMA and Dome is an a very early stage and having the contractors focus on low-level functionality will increase their delivery times and therefore should reduce their schedule. This proposal is aimed at helping current vendor schedules. It is *not* suggesting paying any additional funds for it to be implemented.

Change Record

Version	Date	Description	Owner name
1	2019-08-06	Unreleased.	Tiago Ribeiro and Patrick Ingraham

Document source location: `https://github.com/lsst-tstn/tstn-001`

Contents

Proposal to conduct in-house CSC development.

1 Introduction

LSST Observatory Control System is based on a highly distributed, component based software architecture. Each component of the system acts as an individual entity that is responsible for performing a specific task; from directly controlling hardware to coordinating the operation of multiple other components. Components interact trough a middleware communication layer based on the Data Distribution Service (DDS). [1]

To enable LSST's multiple components to communicate via DDS, LSST T&S software developed the Service Abstraction Layer (SAL), which provides a wrapper around the OpenSplice library for all the supported software programming languages (Java, C++, LabView and Python). In short, SAL provides an easier way to translate component interfaces to communicate via DDS. Although SAL considerably simplifies the task of setting up the component communication, it does not provide any high-level support to building software components, notably the CSCs, leaving developers (both external and in-house) without requirements and guidance when handling overall architectural decisions.

It was clear early-on that the lack of a high-level library made it very challenging to develop consistent behavior across software components, specifically CSCs. Fundamental behavior has varied considerably depending upon the developer's understanding of the requirements and their specific application and chosen technologies. The lack of CSC uniformity is especially apparent when it comes to vendor software development, where there is really no incentive or special reason for the vendor's developers to learn or understand a technology and architecture that will likely not be applied or used again in a perceivable future. For these reasons, already delivered components such as M2, the hexapod & rotator, and even M1M3 have or will all require significant modifications prior to being ready for full integration testing. [2]

To overcome these issues, and to expedite development of components, T&S Software concentrated efforts on developing SalObj (`https://ts-salobj.lsst.io`); a new high-level object-oriented Python library that considerably simplify the development of software components

[1] An implementation of the DDS protocol is provided by OpenSplice, a low level library developed by ADLink `https://www.adlinktech.com`

[2] Admittedly, had the current architecture been demonstrated and in-place during the time of the contract bidding, better requirements could have been written. However, there remains a strong argument to leave the vendors to their specialties and specify a more generic communication interface to higher level control software (e.g. the CSCs).

for the LSST. SalObj, combined with a standard CSC template and development environment is now in active use, is designed to handle most of the business rules required by the components, creating uniformity throughout a complex system in CSC architecture, functionality, coding practice and (eventually) deployment. This enables the developers to concentrate on implementing their functionality in a sound, powerful, yet flexible framework.

By capitalizing on the CSC development expertise in TSSW, we can shift the future work at modifying the to-be-delivered CSCs to in-house experts leaving the vendors to focus on their specialities, which is the low level control of their components, to advance their delivery schedules of their components.

2 TSSW CSC Development and Challenges

The development of CSCs has moved to follow a standard template and design in Python using SalObj (discussed above). The uniformity of CSCs is important to ensure future maintenance, troubleshooting, and additional development can be performed by a broad range of LSST software personnel. At the moment, all developers are either actively developing SalObj based CSCs, or focused on learning the skills (e.g. Python) required to do so. To date, there are over ~20 CSCs either written or in development using SalObj, with another ~10 that are planned.

As part of CSC development, unit testing and peer code reviews are now required. Furthermore, simulators are now being nested into the CSC code itself therefore using the real CSC functionality to facilitate integration testing with multiple components. The presence of these simulators is key to ensuring that when software is deployed on the summit it is fully functional and will not result in schedule delays. Although adjustment of functionality of a single CSC may be relatively straightforward, it is critical to ensure there is no impact on the higher level software components or other CSCs which rely on communications which may have been affected. Rather than going through a large number of tests using real hardware, detailed tests are run using the simulators to test these interactions to minimize the amount of testing required once deployed on the summit. Although some vendors have/are providing simulators, they are not necessarily at a level required to perform robust integration testing.

Due to the tight relationship between the SAL and software interface (XML), being able to actively manage (and adjust) interfaces to CSCs is non-trivial. This necessitates the need for LSST to control the CSC interface between it and other components, which is the part of the

interface that is not of interest to vendors. With the development of SalObj and the adopted use of it by the TSSW group, dealing with these interactions is rapidly becoming the strength and focus of the team. However, because this SAL+XML functionality is specific to LSST, it is not a generic tool used by the software community and therefore not the expertise of any vendor. Using a more widely adopted communication and interface strategy (e.g. a TCP/IP socket) will help keep vendors focused on their expertise (e.g. servo tuning and low level control) and will not be obliged to learn the LSST specific software. Lastly, as no significant development requirements on the CSCs was imposed on vendors, every vendor delivered CSC will be different and therefore developers will be forced to spend time learning each CSC rather than focusing on the functionality of the underlying component. For example, should a software developer be embedded with a vendor force knowledge transfer (e.g. the TMA), the developer would focus on the low-level functionality of the component rather than learning their interpretation of how a CSC should be architected.

Another reason for LSST to control the development of CSCs is to assist with deployment of software. In the case of hardware systems, changes to the functionality of the hardware is expected to be very limited. However, because of the coupling of SAL+XML, if a change is made to *any* component, the software must be re-deployed to *every* CSC, even if no functionality change to the hardware is required. There will be cases where communication interface changes will be required during the night. At the moment, updating the ATMCS CSC (written in LabVIEW and deployed on a cRIO) takes multiple (4+) hours. Having the vendors deliver a standard socket as the interface eliminates the deployment challenges and by moving the CSCs into python and not requiring their deployment on cRIOs will allow deployment to occur in minutes.

2.1 TSSW SalObj CSCs

Components using SalObj already completed and (at least partially) integrated:

- ATDome
- ATDomeTrajectory
- ATHexapod
- ATAOS
- ATSpectrograph

- ATMonochromator
- CBP
- DIMM
- Electrometer
- Environment
- GenericCamera
- LinearStage
- Scheduler
- ScriptQueue
- TunableLaser

Components needing to be completed (being done by TSSW):

- ATBuilding
- ATWhiteLight (under active development)
- FiberSpectrograph (under active development)
- MTAOS (under active development)
- MTDomeTrajectory
- MTEEC
- MTLaserTracker (under active development)
- SummitFacility
- Watcher (under active development)
- MTM1M3 (?)
- MTGuider?

3 Status of Vendor CSC Deliverables

This section describes the current state of the CSC for each deliverable component from external vendors. For already delivered components, it describes the current state and what is required to bring the software to a state of readiness and maintainability. As demonstrated below, with the exception of the LSST Camera, none of the components will be delivered in a way that facilitate easy integration to the LSST system.

3.1 Dome

The Dome vendor (EIE) has started to work on the low-level control aspects of the Dome control software only a couple months ago (~June). We proposed a major refactor of their original design, where they proposed each component of the Dome was represented by a CSC. Instead, they are now working on having low level components exposing sockets for a single high level component to connect and control. The CSC, to be written in LabVIEW at a future time, is still under contract to be delivered by the EIE. Their new architecture provides a natural place to de-scope the CSC development from the contract.

3.2 TMA

The existing control software for the TMA follows a satisfactory design. Low level controllers connect to intermediate controllers that are responsible for coordinating the work on the multiple hardware components. The intermediate controller exposes a TCP/IP socket that the CSC (written in C++) connects to which exposes the controls and make telemetry available to the LSST system.

With no high level library available for C++, their CSC does not follow a friendly design pattern. The software is still using a pretty old version of the SAL library (v3.5 whereas we have released versions 3.6 through 3.10 and are testing v4.0). The amount of work required to update their CSC to a current version of the library has not been scoped and may well take almost as much time as to re-write the entire CSC using SalObj. Furthermore, because the CSC is owned by the TMA vendor and not being released at the same cadence as the SAL/XML, we cannot use it for integration tests (even if it did have a useful simulator).

It is worth noting that despite the vendors provide a simulator for their software, it is only a telemetry simulator with no embedded behavior. What the simulator does is to generate and publish through SAL random numbers for the component's telemetry. Since there is no

simulated behavior it is not possible to use it for integration testing purposes. Furthermore, because the telemetry numbers are random, no unit-type testing (meaning verification of output against a known value) can be performed.

3.3 HVAC

Vendors spent a considerable amount of time porting code to perform the heavy duty from Windows to Linux. The current contract will deliver an MQTT interface and will not include development of the CSC which is the correct approach. Python has a library for MQTT communication.

3.4 LSST Camera

The code for the LSST Camera, written in Java, is at a very advanced stage. SLAC, specifically Tony Johnson, has been working for many years with Dave Mills and has already delivered working products. With the exception of working on packaging/deployment, this may not be worth modifying at this stage of the project. Detail discussion with Tony Johnson may prove informative.

3.5 Already Delivered Components

These are non-python components which are already delivered. At a minimum, most require work to bring up to current standard (SAL/XML). Many will require additional changes and all require deployment strategy implementation. The work described in these sections is what can be avoided by taking on the scope of CSC development for the remaining contracts.

3.5.1 Pointing Component

We have been working closely with the vendors (Observatory Sciences - `http://www.observatorysciences.co.uk`) on this contract, which uses C++ extensively. Even though they are very talented developers and have extensive experience with telescopes, we regularly encounter multiple issues with the CSC which typically consume substantial time to work around and implement fixes. This is a good example that shows how difficult it is to get even talented external vendors to correctly implement basic architectural procedures without a good supporting software library. Unfortunately, given the current advanced state and complexity of the software it is not recommended that this system be moved to SalObj.

3.5.2 M1M3

The M1M3 software, written primarily in LabVIEW but with significant amounts of C++, has been demonstrated to work using SAL communication to an instance of the original EFD. However, getting the communication to work was a significant challenge which is one of the reasons why the SAL version used for the testing was not regularly upgraded. Unfortunately, due to Chris Contaxis' departure, and now Harini's departure, very few people will have had any experience working with the code (Andy is the only person left?). This is an example where having a common format for CSC production is important. Currently, someone will have to be dedicated to going in and understanding the current code in order to upgrade it. This may be an ideal opportunity to re-factor the code to use the common template, particularly for the thermal control aspects that have not yet been written. Of course, the algorithm aspects of the control loops should not be modified unless absolutely necessary as it will require the re-certification of the software using the surrogate.

3.5.3 Hexapod and Rotator

Both these components have been delivered by MOOG (`https://www.moog.com`). Initial analysis of their software shows that the delivered components suffers from several issues, from software bad practices, poor documentation and other general inconsistencies. At the time, the delivered software is modular, well separated between low and high level controls. Low-level controllers are deployed on cRIOs, although using a highly non-standard approach. The low-level controllers then communicate over a TCP/IP socket connection to a high-level manager controller, written in LabView. The high-level manager controller, on the other hand does *not* use SAL. Instead, there is a third software layer, written in C++, that is responsible for receiving the SAL messages and sending the data to the manager over a second TCP/IP socket.

Overall this is a very fragile and unreliable system with a low level of maintainability. It should be possible to replace the high-level manager and SAL layer with a SalObj component that communicates directly with the low-level controllers.

3.5.4 M2

The M2 component was developed by HARRIS (`https://www.harris.com//`) using LabView. Their software architecture is modular and follow a standard FPGA deployment approach. Low-level controllers are deployed to FPGAs using the standard LabView interface. This software,

deployed on a standard computer, then communicates with a high-level controller, also written in LabVIEW, through a standard TCP/IP connection. The high-level controller is also responsible for the SAL communication to the LSST control network. It is probable that the high-level controller, which contains the CSC, could be replaced with a SalObj CSC.

3.5.5 ATMCS

After spending some time with the developers fixing several issues with the interface we managed to get the (LabVIEW) CSC to a workable stage. A simulator was built in-house in Python/SalObj. The simulator is written in a way that would be feasible to replace the LabVIEW CSC with relatively ease. The CSC would be hosted on a computer in the summit facility and connect to the cRIO via a standard TCP/IP socket. The vendor (CTIO) is on-board to make the switch, but as the CSC development aspect is essentially completed it will require some extra hours to complete. It is anticipated that such an effort would certainly result in a cost savings over the life of the project, most-likely even before the end of commissioning. As mentioned previously, currently building and redeploying the XML/SAL interface layer takes 4+ hours (without any testing).

It should be noted that the ATSpectrograph and ATDome use software deployed on cRIOs communicating via TCP/IP to their hosted externally CSCs. This model has proved very successful for integration/testing/deployment of LSST software.

3.5.6 ATPneumatics

This is developed by CTIO as well and is identical in all issues discussed in the ATMCS section above.

4 Working with Vendors to Redefine the Interfaces

Definition of the interface for on-going contracts requires input from a broader audience. However, the requirements/interfaces used in the existing contracts will not require significant modification as the functionality, deliverables and testing of the software remains the same. The state machine remains unchanged, however, the handling of the state transitions, including the commands given to each component could be moved up and handled at the CSC level, whereas the vendor could write basic scripts to demonstrate state transitions using the agreed upon API.

Should the project decide that exploring the de-scoping of CSCs from vendors, then a detailed dive into the current contracts software and interface is recommended in order to put forth a suggested modification to the vendor (e.g. a TCP/IP socket with string-based commands to their standard library). Another option would be to let the vendor suggest an interface.