TMT common software update

Kim Gillies[*], Allan Brighton, Hanne Buur
Thirty Meter Telescope International Observatory, 100 West Walnut St., Suite 300, Pasadena CA, 91124, USA

ABSTRACT

TMT Common Software (CSW). CSW consists of software services and library code that is used by developers to create the subsystems and components that participate in the software system. CSW also defines the types of components that can be constructed and their functional roles in the software system. TMT CSW has recently passed its preliminary design review. The unique features of CSW include its use of multiple, open-source products as the basis for services, and an approach that works to reduce the amount of CSW-provided infrastructure code. Considerable prototyping was completed during this phase to mitigate risk with results that demonstrate the validity of this design approach and the selected service implementation products. This paper describes the latest design of TMT CSW, key features, and results from the prototyping effort.

Keywords: middleware, software design, infrastructure

1. INTRODUCTION

From a software communications and integration viewpoint, the TMT Software System consists of a set of software components interacting with each other through a software communications backbone and software infrastructure (middleware). The integration of all these software components requires software infrastructure that is outside the scope of the individual components. Figure 1 shows the TMT subsystems that rely upon and the TMT communications backbone at the telescope site. The design of the TMT Software System has resulted in a small set of services and associated software called TMT Common Software (CSW) that is focused on the task of integrating software components.

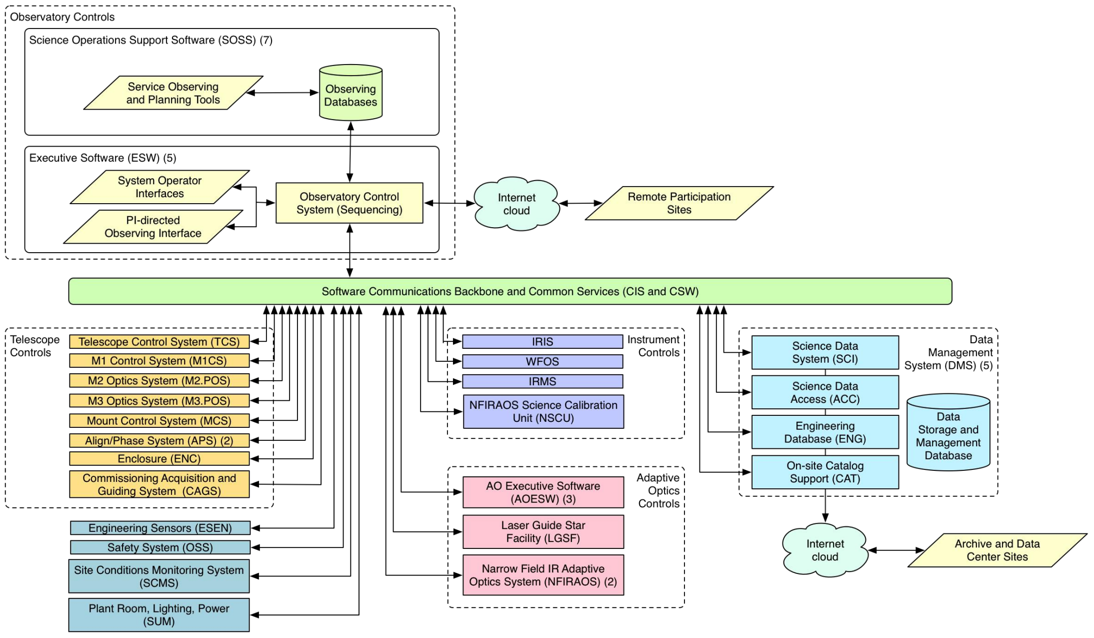
Observatory Controls
Science Operations Support Software (SOSS) (7)
Service Observing and Planning Tools
Observing Databases
Executive Software (ESW) (5)
System Operator Interfaces
PI-directed Observing Interface
Observatory Control System (Sequencing)
Internet cloud
Remote Participation Sites
Software Communications Backbone and Common Services (CIS and CSW)
Telescope Controls
Telescope Control System (TCS)
M1 Control System (M1CS)
M2 Optics System (M2.POS)
M3 Optics System (M3.POS)
Mount Control System (MCS)
Align/Phase System (APS) (2)
Enclosure (ENC)
Commissioning Acquisition and Guiding System (CAGS)
Engineering Sensors (ESEN)
Safety System (OSS)
Site Conditions Monitoring System (SCMS)
Plant Room, Lighting, Power (SUM)
IRIS
WFOS
IRMS
NFIRAOS Science Calibration Unit (NSCU)
Instrument Controls
AO Executive Software (AOESW) (3)
Laser Guide Star Facility (LGSF)
Narrow Field IR Adaptive Optics System (NFIRAOS) (2)
Adaptive Optics Controls
Science Data System (SCI)
Science Data Access (ACC)
Engineering Database (ENG)
On-site Catalog Support (CAT)
Data Storage and Management Database
Data Management System (DMS) (5)
Internet cloud
Archive and Data Center Sites

Figure 1: The TMT control system is a complex distributed system. The green bar represents TMT Common Software. All TMT components use the services that are part of Common Software.

*kgillies@tmt.org; phone: 1-626-395-1655; http://tmt.org

The individual software components have a wide range of purposes within the software system. Some will be standalone, software-only applications; others will be software components within complex software/hardware subsystems. The CSW also defines a set of standard components with specified roles that reduce the amount of documentation needed to describe a design and the component interfaces.

Creating a software architecture and common software for a modern observatory is not a trivial task. The choice of a traditional platform that has been used by other observatories is viewed as safe because of familiarity and past success, but may also be saddled with older technologies that will be difficult to support on a new project with a long lifespan. On the other hand, following current software trends can be viewed as too risky. There are always concerns for maintainability during operations by a small staff that must understand a large code base.

These issues were considered as part of the TMT Common Software design process, which successfully passed its preliminary design review in December, 2015. This paper describes the progress on TMT CSW focusing on its unique features, which include its use of multiple, open-source products as the basis for services, and an approach that works to reduce the amount of CSW-provided infrastructure code. Considerable prototyping was completed during this phase to mitigate risk with results that demonstrate the validity of this design approach and the selected service implementation products. This paper describes the latest design of TMT CSW, key features, and results from the prototyping effort.

2. STRUCTURE VIEW

Many observatory control systems define programming interfaces at a high level, but leave the design and implementation of software within the subsystems to each subsystem. This gives freedom to the developers during construction, but can result in waste effort as similar infrastructure code is developed by different groups requiring extra tests and debugging. During operations, the cost is high as the operations team must learn the codebase of many different subsystems. TMT CSW has taken the approach of defining the structure of the software system with a limited set of components that can exist in the control system and their component responsibilities. All subsystems implement instances of these component types. An example observing component structure is shown in Figure 2.

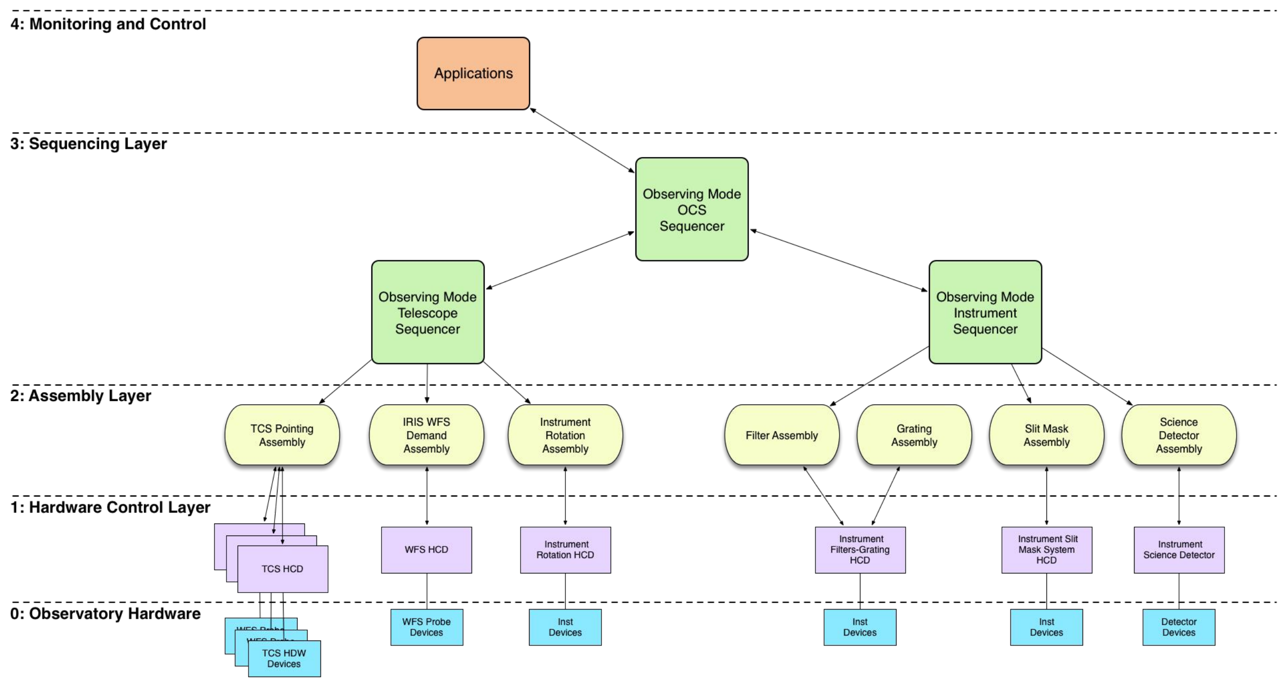
4: Monitoring and Control
Applications
3: Sequencing Layer
Observing Mode OCS Sequencer
Observing Mode Telescope Sequencer
Observing Mode Instrument Sequencer
2: Assembly Layer
TCS Pointing Assembly
IRIS WFS Demand Assembly
Instrument Rotation Assembly
Filter Assembly
Grating Assembly
Slit Mask Assembly
Science Detector Assembly
1: Hardware Control Layer
TCS HCD
WFS HCD
Instrument Rotation HCD
Instrument Filters-Grating HCD
Instrument Slit Mask System HCD
Instrument Science Detector
0: Observatory Hardware
TCS HDW Devices
WFS Probe Devices
Inst Devices
Inst Devices
Inst Devices
Detector Devices

Figure 2: TMT software subsystems are created from standard structure components with defined responsibilities in the control system.

There are four layers called Hardware Control Layer, Assembly Layer, Sequencing Layer, and Monitoring and Control.

2.1 Layer 0 – Observatory Hardware

Layer 0 represents the actual hardware being controlled and the hardware controllers that interface the hardware to the computer systems. The trend in industry and also in astronomy is to use motion controllers, other hardware controllers and I/O, or standalone Programmable Application Controllers communicating via high-level, standardized protocols over a TCP/IP-based network. This approach minimizes custom hardware within the computer and leverages the sizeable resources of industrial control vendors resulting in a more maintainable system.

2.2 Layer 1 – Hardware Control Layer

Layer 1 in the TMT software system is the Hardware Control Layer. A sea of similar software components called *Hardware Control Daemons* (HCD) control the low-level hardware of the telescope, adaptive optics, and instruments. An HCD is similar to the device driver found in many systems. Each HCD is associated with a networked motion controller, a PLC/PAC, or other low-level hardware controller present in layer 0.

The following are characteristics and responsibilities of HCDs.

- The HCDs act as adapters and provide a uniform software interface optimized for hardware control to the layers above.
- HCDs for multi-channel hardware controllers or PLCs support multiple asynchronous client requests and responses.
- HCDs monitor the status and state of their hardware controllers and provide state to the layer above. They may also provide telemetry, and alarms.
- HCDs will often convert input user or engineering units to units appropriate for the hardware.
- HCDs encapsulate the protocol and transport needed by a hardware controller. This may require vendor libraries.
- The HCD provides a suitable location for device simulation allowing end-to-end system testing without hardware presence.
- At this layer in the software each HCD and hardware controller is independent; any required synchronization with other hardware is handled in a higher layer.
- The HCDs are *always executing,* and each can be accessed at any time by the software layers above.

2.3 Assembly Layer

The Assembly Layer exists just above the Hardware Control Layer at layer 2 in Figure 2. Software at this layer consists of components called *Assemblies*. In the structure view, each Assembly represents a collection of hardware that makes sense from the user or domain point of view. Examples of Assemblies are a filter wheel, a deformable mirror, or a detector controller. Assemblies can supervise and control one or more HCDs, but it is not always necessary that an Assembly control HCDs. For instance, an Assembly may be the interface to a library of code. The following are the known characteristics and possible responsibilities of an Assembly.

- An Assembly represents a user-oriented device in the software system.
- Assemblies control HCDs, not hardware controllers (which is the role of the HCD).
- An Assembly can coordinate and synchronize multiple HCDs.
- An Assembly allows the grouping of HCDs into higher-level devices. This is needed when individual hardware devices must be considered and coordinated as a unit rather than as individual devices.
- An Assembly can read sensors, use external events, and perform computations as needed to control hardware. For instance an Assembly representing an atmospheric dispersion corrector device based on prisms needs to compute two angle positions based on the telescope's current zenith angle and parallactic angle. In this example, the Assembly has a computational role as well as the control of multiple devices.
- An Assembly may not control HCDs. It can be an adapter for a code library that is not related to hardware control.
- An Assembly may periodically run diagnostic code to determine failures and generate alarms if failures occur.
- An Assembly provides a testable unit of user-oriented functionality with minimal dependencies.

2.4 Sequencing Layer

The Sequencing Layer is layer 3 in Figure 2. Components at this level are called Sequencers or Sequence Components because they take more complex descriptions of tasks and control and synchronize the actions of the Assemblies to accomplish the tasks. Sequence Components in this layer share a software interface that allows them to be plugged together to form the sequencing engine for a specific observing mode. Individual sequencers can provide higher-level control of a set of distributed hardware (e.g., init). Individual sequencers can be programmed using scripts or written as a static pieces of code. In Figure 2, the OCS sequencer supervises and controls the Telescope Control System sequencer and the Instrument sequencer.

- Each observing mode can have a unique sequencer arrangement consisting of multiple sequence components.
- The sequencer arrangement can be created when the observation is executed.
- The sequencer scripts are loaded at the start of an observation and can be different for each observing mode.
- An observing mode sequencer communicates with Assemblies or HCDs using the services of CSW.

2.5 Monitoring and Control Layer

The Monitoring/Control Layer, layer 4 in Figure 2, is the layer of software that contains the user interface programs that are used to observe with the telescope. At TMT there will be graphical user interfaces for use by observers during observing as well as numerous applications for monitoring the state of the control system. These applications use the CSW services to control and monitor the system.

2.6 Innovation

Structuring the control system with a limited number of defined components with this granularity is an evolutionary change from recent observatory control systems. This approach has a many advantages from a number of points of view as listed in Table 1.

Table 1: The TMT software structuring approach has advantages over other traditional approaches in every project phase.

Advantage	Project Phase
Structure is simple with a limited number of component types. Each component has a well-defined role.	Design and Construction
Structure approach minimizes unique solutions and excess code waste related to elaborate subsystem designs and tool choices and the integration of different implementations.	Design and Construction
Reduces the need for large, bloated, difficult to change subsystem interfaces.	Design, Construction, and Operations
HCDs, Assemblies, and Sequence Components can be developed and tested in a orderly, bottoms-up way, allowing a working system to be built from smaller, tested components.	Construction, Integration, and Verification
More easily supports agile development of vertical features. The smaller components in each layer can be scheduled independently, designed and tested within a few iterations.	Construction and Management
Software construction cost is reduced because less unique software is needed with less required testing and debugging.	Construction and Management
HCDs and Assemblies provide ability to be grouped in ways not considered during design and construction. Sequencing layer provides the ability to dynamically group Assemblies as needed for an observing mode.	Operations

3. SERVICES

The refinement of the CSW design during the preliminary design phase has resulted in the set of nine services shown in Table 2 that components use for integration and communication. Multiple implementations of each of the services are available as open source and commercial products. The general philosophy of TMT Common Software follows the TMT requirements that software systems should take advantage of standards and open-source software. The goal is to write as little software as is necessary for CSW services by reusing existing open-source packages and isolating the implementation choices to allow alternate or improved implementations over the TMT lifespan. For each service, a TMT-focused interface and glue layer is constructed that isolates details of the product. This approach of using the best available software for each service rather than trying to find a single infrastructure that can meet all our needs has been a good choice so far. Over the preliminary design phase the critical services have been implemented with good results; in a few cases, multiple implementations have been created with different products. The following sections summarize progress and prototyping during preliminary design for a few specific services.

Table 2: TMT CSW provides nine shared integration services. This table provides the service name and a description for each service.

Service	Description
Authentication and Authorization Service	Centrally manage user authentication/access control
Location Service	Register and locate component connection information
Connection and Command Service	Support for receiving, sending, and completing commands in the form of configurations
Event Services	Services that enable publish/subscribe event streams, and status/event publish
Alarm Service	Standards-based support for component alarms and health
Configuration Service	Manage system and component configuration file changes
Logging Service	View, capture, and store local and distributed logging information
Database Service	Provides components with access to a shared, centralized, relational database
Time Service	Precision time access for synchronization based on IEEE -1588-2008 and Precision Time Protocol (PTP)

3.1 Location Service

The CSW Location Service of handles component registration and discovery in the distributed TMT software system. During the initialization of a component (i.e. a Sequencer, Assembly, or HCD), it will *register* its name along with other information such as interface type and connection information to the Location Service. The reason for the Location Service is that details of connection information should not be hardwired, they should be discovered at runtime. Discovered information includes a protocol (e.g., HTTP), interface type (e.g., command), host and port.

The service has been implemented in three ways during preliminary design. The first prototype for testing was implemented in Akka, the CSW primary framework. The second implementation using the Etcd (https://github.com/coreos/etcd) distributed key-value store was dropped due to a few undesirable features. The final implementation is based on mDNS or multicast DNS-based discovery, which is based on Internet Engineering Taskforce standards (RFC 3927, mDNS, and DNS-SD). It is familiar to OS X users as ZeroConf or Bonjour. This implementation has performed well and has the advantage that it is preinstalled in Mac OSX and many Linux distributions.

3.2 Connection and Command Service

In the TMT software design, an Application or Sequencer connects to Assemblies and commands them by submitting configurations. Assemblies connect to one or more HCDs. In TMT, commands are peer-to-peer connections between the sender of a command and the receiver. Commands flow down through the Sequence Components to the Assemblies, HCDs and hardware in a hierarchy. The service called the Connection and Command Service (CCS) provides this functionality

During preliminary design phase, the command requirements of the system were re-analyzed in depth using use cases based on example observations that traced the system behavior from the highest level to the hardware. This analysis helped to refine the kinds of commands and the support component builders need from the service. The five basic commands are shown in Table 3.

Table 3: Analysis of command use cases resulted in five commands that are issued between components.

Command	Description
submit	The component is asked to match the demand specified in the configuration argument, acknowledge receipt, and provide completion information to the sender of the command.
oneway	The component is asked to match the demand specified in the argument. Acknowledgement is required, but no completion information is needed.
request	A named set of attributes and values is sent to the component for immediate execution. Caller blocks waiting for a response like a traditional RPC call.
cancel	Actions associated with a specific submit are cancelled. The runId of the submission is used to cancel.
resume	Resume a Sequence Component paused by a WaitConfig.

Configurations that are submitted to components can take a long time to complete while devices move into position. Therefore, the submit commands are asynchronous and non-blocking. Once any actions start, the component receiving the submit command notifies the caller of successful or failed completion in the future when the actions are finished. Completion notification is needed to allow Sequencers to synchronize and sequence the systems as needed to gather science data. The following short example shows how a component is resolved and sent a configuration using the submit command of CCS.

```
val dest = resolveAssembly("Assembly-1")
val obs1 = ObserveConfig("wfos.red.detector")
           .set(exposureTime, 22.5)
           .set(repeats, 2)
           .set(exposureType, OBSERVE)
           .set(exposureClass, SCIENCE)
val obarg1 = ObserveConfigArg("2025A-Q-P012-0123", obs1)
val runId = dest.submit(obarg1)
```

The prototype Connection and Command Service is based on the Akka toolkit (http://akka.io). Akka is based on actors and messages and exactly matches the CCS model. Akka is a well-supported, open source part of the Lightbend (http://lightbend.com) Scala and Java toolset. Akka provides binary serialization of messages and a lightweight, asynchronous and non-blocking approach to HTTP. Load testing of the HTTP infrastructure measured 18,000 pings/second to far exceed what is needed in CSW.

3.3 Event Services

The Event Services are based on a publish subscribe message system paradigm. The advantage of this type of message system is that publishers and subscribers are decoupled. Publishers can publish regardless of whether there are subscribers, and subscribers can subscribe even if there are no publishers. The relationship between publishers and subscribers can be one-to-one, one-to-many, many to one, or even many-to-many. Another advantage of publish-subscribe systems is that components and subsystems can startup and stop independently without requiring special interactions or startup sequences in negotiation with other systems. This service is useful in TMT for use by subsystems such as the TCS that calculate position demands for other distributed components.

During preliminary design a new implementation based on the Redis product was created (http://redis.io) in order to have the optional ability to store events when published and read current values at any time. Redis is an open source key-value store and can store arbitrary data with keys including lists, sets, hashes and sorted sets and was one of the products benchmarked earlier for use in the Event Service [1]. Redis also provides a publish/subscribe capability where a client can watch for changes to a specific key/value and get callbacks when the value is modified. This version is called the Telemetry Service and adds a telemetry/status cache that can be read or written to at any time.

3.4 Alarm Service

A standalone, standards-based Alarm Service was suggested during conceptual design and the design was completed during preliminary design. Features of existing alarm functionality in OPC-UA and EPICS and the ISA-18.2 alarm system standard document were reviewed to provide opportunities for reuse and/or design input. The design for an Alarm Service supporting the features was created, and a prototype will be implemented for final design. The alarm features required for the TMT Alarm Service are shown in Table 4.

Table 4: The TMT CSW Alarm Service shall support the following ISA-18.2 features.

Feature	Description
Activation/Out-of-Service	An alarm can be active or inactive as desired by the operator. It only functions when it is activated, which is the default state. A reason for deactivation might be because a system or component has been taken out of service.
Acknowledged Alarms	The service should support alarms that require alarm confirmation by the operator.
Unacknowledged Alarms	The service should support alarms that do not require confirmation by the operator.
Latched Unacknowledged Alarms Latched Acknowledged Alarms	The system should support alarms that stay in the alarm state even after returning to the normal state. It is unacknowledged until the operator acknowledges the condition.
Auto Acknowledge	An alarm can be set to auto-acknowledge rather than require an operator acknowledgement.
Shelving	The operator should be able to temporarily suppress an alarm.

3.5 Configuration Service

The Configuration Service provides a centralized persistent store for "configuration files", which in this context is a set of values describing state, initialization values, or other information useful to a component. As an example the telescope control system requires look-up tables of various kinds, pointing model parameters, or parameters for setting up a motion controller. At the applications level, the GUI used by an operator could provide a button to save offsets between an instrument science field and its acquisition camera origin. These are the kinds of scenarios that use the Configuration Service. The service provides the important feature of storing and tracking versions of configuration files. All versions of configuration files are retained providing a documented record of changes for each configuration file. Components and users can save a new version without fear that the current version will be lost. It is always possible to easily restore to the most recently saved version or a default version.

The Configuration Service matured during preliminary design phase prototyping. At the request of testers the ability to set default files, store versioned binary files, and store oversized files were added. The Configuration Service functionality is based on a source code management system, which has been the Git package (https://git-scm.com). The oversized files feature uses the Git-annex package (https://git-annex.branchable.com). Based on comments from the preliminary design review, the performance during commits of the Git-based Configuration Service was investigated and found to slow down linearly with the number of commits. The Apache Subversion package (https://subversion.apache.org) was then tested, and the time to commit a file was found to be almost constant and not depend on the number of commits. The Configuration Service was re-implemented to use Subversion.

3.6 Innovation

Every service needed by CSW is similar or identical to services needed by the large Internet companies, and therefore there are multiple products to choose from. The CSW approach has been to make use of the best service solutions in the open source community. The implementation product for each CSW service is chosen entirely on the requirements of the service and the performance of the implementation product. Each CSW service can use a different implementation product, and each can be changed as needed because it is encapsulated in a simple CSW programing interface. Early on it was clear that none of the frameworks in use within astronomy were able to take advantage of the rapid changes in the open source community; something that is important for a project with a long lifespan. There are tremendous advantages for the project and team to depending on and being part of a large open source community. Over the preliminary design phase, considerable prototyping was done to reduce risk with a focus on prototyping of critical services. As discussed in this section, services have been re-implemented multiple times with relative ease suggesting that the approach of using multiple implementation products behind a TMT interface is sound.

4. LIBRARY AND FRAMEWORK

CSW has been designed as primarily a library with some optional parts that are more like a framework. The difference is subtle but important. A library is a set of programming interfaces. When using a library, the developer is in charge of running the system and the code makes calls to the library functions as needed. A framework is much more. When using a framework, the framework is in charge of running the system. The developer must use the framework and it defines interfaces where the developer "hooks in" his code. It is often stated: A framework calls your code, you call a library. This important difference is shown in Figure 3.

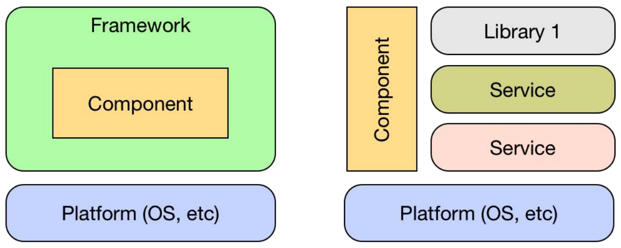

Figure 3: Framework versus library. CSW is primarily a library with some framework code available. This results in less code and code that easier to read, understand, and document.

CSW has worked hard to be only a library, but it also provides some framework code for developers that require more help or to reduce boilerplate. Even then, the framework code is focused on specific tasks such as receiving commands in a certain way or simplifying component initialization. The reason for favoring library over framework is that frameworks are complex, require a lot of programming effort in design and debugging, make code difficult to read, and usually must make design choices that are too restrictive for programmers. The TMT software structure, which includes a few kinds of components with specific responsibilities, also simplifies the programming task.

In any software system there is always a decision about which libraries or abstractions are visible to the programmer, because these are decisions that affect every programmer and become entwined in the code. Sometimes library dependency code cannot be hidden behind abstractions without causing programming issues similar to frameworks. During the preliminary design prototyping phase, the choice of Akka (http://akka.io) has shown to be a good choice as a middleware solution and has been chosen as a visible, core TMT middleware technology filling the role of middleware such as Data Distribution Service or Channel Access in EPICS. It also provides more than those products with an API and programming model for creating and structuring the internals of an HCD or Assembly and provides some functions in the area of interface control.

The question of whether or not we should attempt to hide Akka from the user programmer is one that is still being discussed. The CSW reduces the needed knowledge of Akka for many simple cases through targeted framework code, but the full Akka programming model is available for programmers to use. Akka is not complex and most programmers can pick the basics up in a few hours or days. The choice of Akka as visible middleware is a fundamental, significant decision. The most important Akka advantages and considerations are the following:

- Akka is designed to be asynchronous, non-blocking, and distributed like the TMT software system. Akka exactly matches our control scenarios.
- Akka is based on actors, a programming model that removes the programming challenges around concurrency and locking that introduce so many hard to find bugs in other systems and languages. Synchronization and locking is extremely difficult for programmers to get right. The Akka model allows safe code to be written without worrying about synchronization and locks.
- Creating a TMT API of similar code or attempting to hide the Akka code would arbitrarily limit useful functionality.
- Akka is extremely high performance and has a small memory footprint supporting millions of messages/sec on a single machine and allows 2.5 million actors/GB of memory.
- The Akka system is extremely well documented with seven known books in print and an extremely active and supportive community groups
- Akka is designed around scalability with cluster deployment support and is extensible for communication protocols and message styles. For instance, it can directly interface with ZeroMQ.
- Akka is a safe choice. Akka is backed by the Lightbend company, the same company that supports Scala. Akka and all related tools are open source on Github. Akka is used in production at many companies and is a popular library on Github.

The tradeoff is that our applications and components become closely tied to the Akka programming environment rather than a TMT-created programming environment. This is a significant product dependency and can be viewed as being a bad choice, but it can be viewed as a good choice too. It is really no different than using some other open-source package such as EPICS. But the advantage is a larger community, more support, training, and documentation.

4.1 Languages and Standards

The CSW library is written in Scala, a JVM-based language that is more powerful and expressive than Java. The services, library and frameworks will also be provided with Java language programming interfaces. An early decision was made to not support the entire parallel infrastructure in C/C++. Hardware devices interface with the system at the lowest level with relatively simple protocol requirements. Discussions are taking place around standard hardware but no decisions have yet been made in this area. There are some computationally intensive systems that are programmed in C/C++ due to real-time performance needs. The structure view of Section 2 accommodates this scenario as well with an Assembly/HCD communicating with the C/C++ code using ZeroMQ+Protocol Buffers. The advantage of this choice is that the interface can be documented in a schema that can be versioned and the endpoints of the communication are generated code guaranteeing compliance with the interface schema.

5. CONCLUSIONS

TMT has taken an unconventional approach with its middleware when compared to recent astronomy projects, but a conventional approach when compared to the software industry where applications are made up of collections of specialized technologies and services. The structure view, use of services, and library emphasis over framework are all an evolution of past designs.

The preliminary design for CSW was successfully completed in December, 2015 with a final design review planned for August, 2016. Prototyping continues during the final design phase. The implementation phase that is focused on "industrializing" the prototype code is planned to start by the end of 2016 with a vendor to be chosen by our TMT partner based in India. The CSW source code is open source and available now on the TMT software Github site at: https://github.com/tmtsoftware/csw for use and contributions by interested parties.

6. ACKNOWLEDGEMENTS

The TMT Project gratefully acknowledges the support of the TMT collaborating institutions. They are the California Institute of Technology, the University of California, the National Astronomical Observatory of Japan, the National Astronomical Observatories of China and their consortium partners, the Department of Science and Technology of India and their supported institutes, and the National Research Council of Canada. This work was supported as well by the Gordon and Betty Moore Foundation, the Canada Foundation for Innovation, the Ontario Ministry of Research and Innovation, the Natural Sciences and Engineering Research Council of Canada, the British Columbia Knowledge Development Fund, the Association of Canadian Universities for Research in Astronomy (ACURA), the Association of Universities for Research in Astronomy (AURA), the U.S. National Science Foundation, the National Institutes of Natural Sciences of Japan, and the Department of Atomic Energy of India.

REFERENCES

[1] Gillies, K. and Bhate, Y., "Performance testing open source products for the TMT event service," Proceedings of SPIE Vol. 9152, 91521H (2014).