A MIDDLEWARE-NEUTRAL COMMON SERVICES SOFTWARE INFRASTRUCTURE

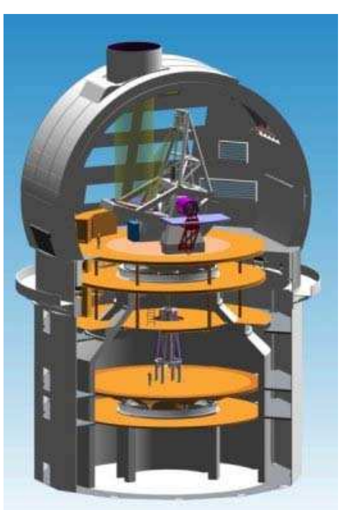
S.Wampler
National Solar Observatory, Tucson, AZ, USA,

ABSTRACT

Astronomical and Solar Observatories have recognized the advantages of establishing *common services* software infrastructures. Software development, integration and maintainability are enhanced. Because of the scope of these common services the use of standard communication middleware is becoming more prevalent. However, most common services implementations are designed around this middleware, creating a deeply embedded reliance on specific middleware choices. This effectively means that projects are locked into the choice of communication middleware from early in the development cycle throughout the lifetime of the project. Changing middleware becomes a costly exercise. To help reduce the impact of such choices, some sites have instead adopted *middleware standards,* such as CORBA, allowing the capability of replacing a standard communications component with another, equivalent, component. This still locks projects into a specific standard, not always the best option for projects whose lifetimes can be expected to span decades. The Advanced Technology Solar Telescope has adopted a different approach and has designed a middleware-neutral common services architecture. Services provided by this infrastructure are designed to meet the project software requirements and implemented using small, pluggable, service tools that map those services onto middleware components. The result is an architecture that is extremely flexible and independent of specific middleware choices. It is even possible to simultaneously support multiple middleware systems, simplifying the task of migrating a project from one middle choice to another. This paper presents an outline of the *ATST Common Services* design and experience with an initial implementation.

INTRODUCTION TO ATST

The Advanced Technology Solar Telescope (ATST)[1] is designed to be the premiere solar research facility in the world. Its 4m primary mirror provides over twelve times the collecting power of any existing solar telescope and provides resolution of solar features in the tens of kilometers. The off-axis design ensures optimal performance in polarimetry. Integrated into the design is a 1300 actuator adaptive optics system and a large, rotating Coudé platform installed in the pier. Thermal issues are a major concern and have resulted in a novel enclosure, a massive prime focus heat stop with integrated occulter, and other features not found in stellar observatories. ATST is being constructed on top of Haleakala on Maui, Hawaii.



Since ATST only sees a small fraction of the solar disk and since features of interest on the Sun move about the surface irregularly, pointing and tracking present interesting control challenges, as does off disk guiding during coronal observations. Nevertheless, the vast majority of ATST systems match with those found at any major observatory and the principles of control system design are similar.

THE EVOLUTION OF TELESCOPE SOFTWARE INFRASTRUCTURES

As part of the design for the control system, ATST undertook a survey of existing observatory software systems focused on understanding the trends in these designs[2]. The hope was to either identify an existing structure suitable for use with ATST or understand these trends enough to develop a software architecture that would serve ATST throughout its lifetime.

For the most part these trends are obvious – a more towards using commodity hardware and operating systems operating in a distributed system. There is a trend away from custom, 'home-grown' software toward relying more on community standards and commercial off-the-shelf (COTS) software. The breadth of software involved in the operation of a large, modern observatory is also encouraging the move towards developing a site-wide *common-services* infrastructure providing a common foundation for as much of this software as practical.

Advantages of Common Services Infrastructures

Providing a common infrastructure on which the majority of observatory software is based offers a number of advantages. Application developers are provided with a stable, consistent platform that frees those developers from having to worry about technical details. This allows developers to concentrate on providing the functionality required of those applications. The standardization minimizes the number of different tools, libraries, and approaches used in the observatory software. This, in turn, makes it easier for the observatory to maintain requisite software expertise, simplifies maintenance, and lowers long-term software costs.

Role of Communication Middleware

A key role of common software services is to provide the communication mechanisms between applications. This is particularly critical as observatories move toward highly distributed control systems. These communication mechanisms must be high-performance and robust. The cost of developing and then maintaining such systems in house has become prohibitive. Most common software services have adopted either commercial packages or community software as the foundation for communications. Because these packages address issues inherent in most distributed software environments, they often provide a wealth of features that are directly applicable to those needs found in modern observatories. Besides connection services, most of these *communication middleware* packages also include sophisticated event and/or notification services. Some include special support for data streaming, automatic reconnections, and other valuable features.

While specialty packages exist for communication middleware, the major driver is toward using well standardized solutions, for much the same reasons that drive observatories toward common services in general. Using a standard communication middleware package increases the likelihood that the package has been well tested by others and increases the chances of finding useful support when problems are found.

Disadvantages of Communication Middleware

There are some problems with the typical use of communication middleware. Despite being standardized, some packages are only available from select (possibly only one) vendors. This imposes a risk on the observatory should the vendor decide to change critical behaviour of the package or phase it out entirely. Because this middleware is aimed at a broader audience than observatory operations, market forces may also move the middleware in directions that run counter to the observatory's needs.

The projected lifetime of such common services, including the communication middleware, is typically a key factor. The volatile nature of software, coupled with the need to have the common services available very early on in the development process, means that the choices made on common services and communication middle can have a profound impact on observatory software. Sometimes the choices that are made by an observatory mean that the software is obsolete before it is put into operation. Further, since communication middleware packages are typically designed as an integral part of the software system, the cost of replacing one middleware with another can be considerable..

An approach taken by some observatories has been to adopt a communication middleware standard instead of a single communication middleware package. This typically increases the number of vendors from which specific parts of the package may be obtained and allows some ability to advance the software infrastructure as improved implementations of this standard become available. One standard that has been used with some success on a variety of projects is CORBA[3]. While this helps address some of the issues with adopting a communication middleware, it does not solve them. Standards evolve over time, often driven by market forces, and may even fall into obsolescense. A standard represents a snapshot of an existing technology level that can quickly fall behind as new technologies are developed.

THE ATST COMMON SERVICES INFRASTRUCTURE

The ATST Common Services[4] (*ATSTCS*) draws heavily from the design principles found in the ALMA Common Services[5] (*ACS*) architecture but has the added goal of remaining *middleware neutral*. Simply put, ATSTCS is not reliant on a specific communication middleware choice or standard and can be readily adapted to different choices as the need arises. In fact, the design is intended to support the dynamic switchover from one communication middleware to another.

Overall design characteristics

ATSTCS uses a tiered structure similar to that found in many similar projects. A (narrow) layer separates the communication middleware from all higher levels.

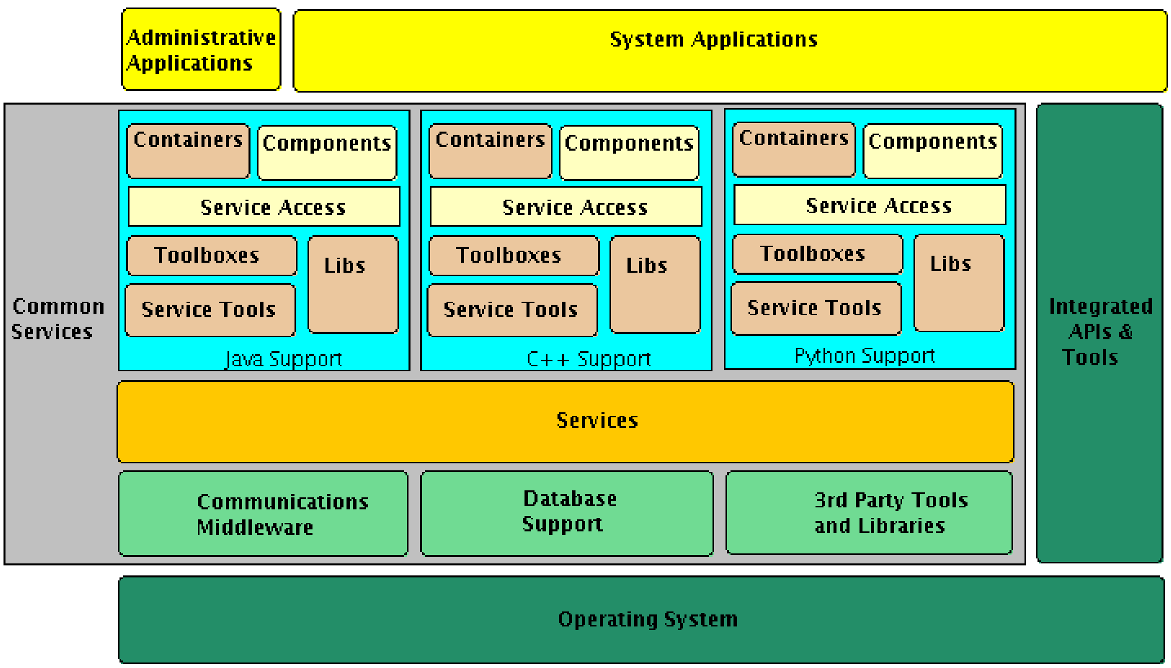


Figure 2 – ATST software infrastructure layers

Also, like ACS, ATSTCS separates out the functional and technical architectures and provides an implementation of the technical architecture. A container/component model is used to implement the foundation of the technical architecture. Developers writing applications based on ATSTCS are able to concentrate on adding the specific functionality required of their applications. Applications are implemented as components that are loaded into containers. The containers are responsible for providing these components with access to the essential common services. There are separate container implementations available for Java and C++ applications.

Service model

The major services are themselves implemented through a series of layers. Application code (i.e. code in a component) accesses each service through a *service access helper*. Each service access helper presents an interface to the developer that is designed to be easy to use and independent of the actual implementation of the underlying service. Service access helpers, in turn, reference a *toolbox* to obtain access to the underlying service. A component's toolbox is shared with the container and provides access to the underlying services to both the container and the component. Services are represented in the toolbox by *service tools*. Service tools understand the details of accessing a specific implementation of a given service. For example, a toolbox may provide an event service tool based on ICE[6], CORBA, or some other communication methodology.

The design philosophy adopted by ATST is to keep service tools small and focused on a specific task. This is analogous to the Unix philosophy of writing small, simple programs that can be joined together in scripts and pipes to perform complex tasks. Most service tools support *chaining* so multiple tools can be applied to a single service action. As a simple example, the following are some service tools that support logging of messages:

- display message on standard error
- post message as an event
- immediately log message to a database
- buffered logging of messages to a database

Any combination of these service tools may be chained in a toolbox (though chaining the last two would make no sense). Service tools may be *shared* across multiple components within a container or *private* to a single component. Most service tools are implemented so that they may be used as either shared or private service tools.

Toolboxes

As stated above, each component has an associated toolbox that holds the individual service tools. However, the component itself does not see the toolbox as it is entirely contained within the technical architecture. All component-level access to services is through the service access helpers which, in turn, reference the services through the toolbox. The service access helpers are the bridge between the functional and technical architectures.

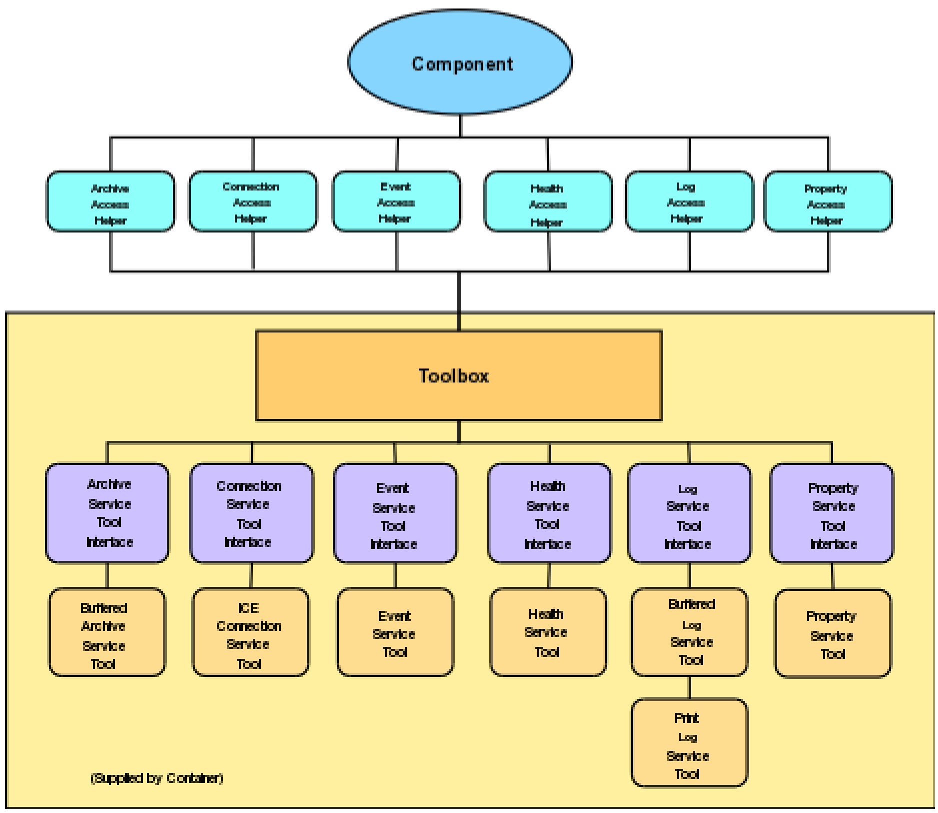


Figure 3 – ATST service access hierarchy

In addition to holding the service tools, each toolbox contains information about the component that is needed by one or more service tools. Thus the service tools do not need access to the component to obtain this information. For example, the name that the component has within the ATST control system hierarchy is held in the toolbox. When a service access is performed, the toolbox passes this name to the service tool. The toolbox also computes and delivers a timestamp for use by those services that expect a timestamp. Other information is available through the toolbox as well.

When a container creates a toolbox and attaches it to a component, it also is responsible for populating the toolbox with the appropriate service tools. To do so, the container uses a *toolbox loader*. Different toolbox loaders exist to load different tool sets and containers can be instructed to use any toolbox loader for any component. The default behaviour, however, is to use the same toolbox loader for all components in a given container.

Once the appropriate service tools have been written, changing from one communication middleware package to another is simply a matter of switching service tools. Since service tools are chainable, it is even possible to *simultaneously* load service tools for multiple communication middleware packages. By doing so, that particular service then operates in a heterogeneous communication environment. This can be useful when integrating legacy systems or when migrating a system to a new communication package. Instead of having to perform a “big bang” migration it becomes possible to perform an evolutionary approach and migrate subsystems independently.

Because containers retain access to each component’s toolbox, it is possible to instruct the container to dynamically alter the set of tools in any toolbox. So an engineer may chain an additional service tool onto the service access for a given component or even replace one service tool with another while the component is operating.

An example: the ATST event service

The ATST event service illustrates a number of the advantages offered by the above infrastructure. At the event service access level, components may *post* and *subscribe* to events. To post an event, a component must provide the event name and value. To subscribe, a component gives the event name and a *callback* operation to be performed on receipt of the event. The toolbox, when asked by the event service access helper to post an event, computes a timestamp and passes the event name, event value, timestamp, and component name to the event service tool. The event service tool is responsible for the actual transmission of this information to the recipient using some communication middleware package.

ATST currently has two event service tools available: one using a CORBA notification system and the other using ICE’s IceStorm event system. Writing a pair of components to measure event service performance, for example, is trivial. Once the two components are written, the selection of the event service tool entirely determines which communication method is used. So end-to-end performance measures using different communication packages become readily available.

An unexpected benefit was recently discovered. Software licensing conflicts currently prevent ATST from distributing code using ICE to outside developers. However, ATST can distribute the CORBA-based service tools, allowing these developers to proceed with the writing of components for use with ATST. When these components are delivered back to ATST, they can be loaded into containers that use toolbox loaders to load the ICE-based service tools.

Another example: the ATST logging service

ATST currently uses the PostgreSQL[7] relational database to record log messages. On linux-based systems, a log service tool that can connect directly to the PostgreSQL backend is used. However, this direct database connection may not be possible on some of the real-time systems being delivered to ATST. On those systems a *proxy* log service tool is envisioned that will use some communication middleware to transfer log messages from components to a log server application running on a linux box. Components on these real-time systems do not see this difference since it entirely hidden within the technical architecture provided by toolboxes and service tools.

Current status

A Java-based alpha implementation of the major ATST common services has been completed and has proven invaluable as a test bed for both the common services infrastructure and analyses of various approaches to providing specific services. A C++ alpha release is underway.

CONCLUSION

The ATST Common Services is based upon the general software infrastructure model provided by ALMA’s ACS. ATSTCS enhances this model by introducing additional layers into to the tiered architecture to isolate implementation choices for specific services from the surrounding software. The result is a software infrastructure that is highly flexible and independent from specific communication middleware packages. The expectation is that the lifetime of the overall system will be extended by avoiding the need for drastic code rewrites as software technology advances.

ACKNOWLEGEMENTS

The ATSTCS design has benefited greatly from discussions between the author and Bret Goodrich. Janet Tvedt has assumed responsibility for the C++ implementation and wrote the ICE and CORBA based service tools. Bret and Janet are also members of the ATST software design team.

This work utilizes data obtained by the Advanced Technology Solar Telescope (ATST) project, managed by the National Solar Observatory, which is operated by AURA, Inc. under a cooperative agreement with the National Science Foundation.

REFERENCES

[1] See http://atst.nso.edu/ for information on the Advanced Technology Solar Telescope.
[2] S. Wampler and B. Goodrich, "Existing Telescope Controls", ATST Technical Report #0005, available at http://atst.nso.edu/library/docs/RPT-0005.pdf.
[3] See http://www.corba.com/ for information on the Common Object Request Broker Architecture (CORBA).
[4] A complete overview of ATSTCS can be found in the ATST Common Services Preliminary Design Review documentation found at http://atst.nso.edu/meetings/cs_pdr.
[5] G. Chiozzi et al, "Common Software for the ALMA Project", ICALEPCS'2001, San Jose, California, November 2001
[6] See http://www.zeroc.com/ for information on the Internet Communication Engine (ICE).
[7] See http://www.postgresql.org/ for information on the PostgreSQL relational database.