Project Documentation
Document SPEC-0022-2
Rev I1

Common Services Framework Reference Guide

Keith Cummings, John Hubbard, Steve Wampler, Bret Goodrich

Software Group

April 2014

	Name	*Signature*	*Date*
Approved By:	Bret Goodrich High Level Software Group Manager	*B. Goodrich*	*15-Apr-2014*
Approved By:	Rob Hubbard Systems Engineer	*R. Hubbard*	*15-Apr-2014*

REVISION SUMMARY:

1. Date: 14 March 2005
   Revision: A
   Changes: Initial PDR version
2. Date: 3 March 2006
   Revision: A1
   Changes: Panguitch-1P2 version
3. Date: 31 August 2006
   Revision: A2
   Changes: NSF PDR version
4. Date: 3 June 2009
   Revision: A3
   Changes: Panguitch-1P11 version
5. Date: 21 October 2009
   Author: John Hubbard
   Revision: A4
   Changes: Panguitch-1P12 version
6. Date: 18 May 2010
   Author: John Hubbard
   Revision: A5
   Changes: Cleaned up install instructions
7. Date: 7 July 2010
   Author: John Hubbard
   Revision: B
   Changes: Canary-0 version
8. Date: 3 September 2010
   Author: John Hubbard
   Revision: B
   Changes: Added to build instructions
9. Date: 22 December 2010
   Author: John Hubbard
   Revision: C
   Changes: Final updates prior to Canary 1 release
10. Date: April 2011
    Author: John Hubbard
    Revision: D
    Changes: Canary 2 updates
11. Date: December 2011
    Author: John Hubbard
    Revision: E
    Changes: Canary 3 updates
12. Date: June 2012
    Author: John Hubbard
    Revision: F
    Changes: Canary 4 updates

13. Date: September 2012
    Author: John Hubbard
    Revision: F1
    Changes: Updates to install instructions
14. Date: December 2012
    Author: John Hubbard
    Revision: G
    Changes: Canary 5 updates, Major changes to sections
15. Date: September 2013
    Author: Keith Cummings
    Revision: H
    Changes: Canary 6 updates
16. Date: September 2013
    Author: John Hubbard
    Revision: H1
    Changes: Package Devel information
17. Date: April 2014
    Author: Keith Cummings
    Revision: I
    Changes: Canary 7 updates
18. Date: 15 April 2014
    Author: Keith Cummings
    Revision: I1
    Changes: Canary 7.0 critical fix with the gcc compiler installation instructions

TABLE OF CONTENTS

Preface

This document describes the ATST Common Services Framework (CSF) in detail. The first volume, the Users' Manual, provides an overview of the common services oriented toward ATST application developers. The second volume goes into the design and implementation in significant detail and serves as the Reference Guide. The final volume is the full Application Programming Interface that provides details on the use of all software components of the Common Services.

How to use this document

ATST software developers that are interested in writing applications using the Common Services should concentrate on Volumes I and III, using Volume II as needed to understand the inner workings of the Common Services, and for details about how to install and configure Common Services. These developers should keep in mind that a major goal of the Common Services design is to present a simpler interface to the application developer. Looking into the "inner workings" exposes the underlying support - where some of the inherent complexity of a large infrastructure system has been pushed. This can be likened to opening the hood on a sleek sports car and examining the engine and drive train - you'll learn something, but does it really help you drive the car better?

ATST software developers that are interested in extending or improving the Common Services should concentrate on Volumes II and III, using Volume I to get a better understanding of how the Common Services are expected to be used.

The Overview gives a quick introduction to the foundation of the ATST Common Services Framework and introduces some of the key terminology. This overview is repeated as the first section in each volume.

Reference Guide

This document provides details on the design and implementation of the ATST Common Services Framework. Unlike the Users' Manual, which is aimed at common service users (specifically Component and Controller developers), this guide is not targeted towards specific users but rather serves as a foundation for anyone needing an in-depth understanding of the ATST Common Services Framework.

While this reference manual provides details on the ATST Common Services Framework, ATST application developers should also find the extended information in the Application Programming Interface document useful.

Related ATST Project Documents

1. SPEC-0012, Glossary and Acronym List
2. SPEC-0013, Software Operational Controls Definitions Document
3. SPEC-0014, Software Design Document
4. SPEC-0017, ATST Common Services Design Requirements Document

Other Documents

5. M. Voelter, S*erver Component Patterns*, John Wiley and Sons, New York, 2003
6. Wampler, *Available middleware and common software solutions,* ALMA ACS Workshop, Garching, Germany, 2004
7. Schwarz, et al, *The ALMA Software Architecture*, Proceedings of SPIE Vol. 5496, 2004
8. *Common software for the ALMA project*, ICALEPCS 2001, San Jose, CA
9. Internet Communications Engine (ICE), www.zeroc.com
10. PostgreSQL relational database

1. OVERVIEW

1.1. BACKGROUND

Early in the conceptual design process ATST undertook a survey of observatory software control systems to determine the best approach to take on software design and implementation. A great deal of useful information was obtained as a result of this survey, one of which is that large, distributed, software projects can reduce their overall development, integration, and maintenance costs by basing as much software as possible on a standard infrastructure.

There are several viable models for this infrastructure in use in modern observatory systems. ATST has elected to use a Common Services model similar to that used for the ALMA project Common Software (ACS). The ATST Common Services Framework attempts to be more streamlined than the ACS and also less dependent on a specific middleware structure. This approach should allow the fundamental characteristics of ATSTCS to be preserved as new middleware technologies are developed during the operating lifetime of ATST.

The benefits of a common services model for infrastructure include:

All major system services are provided through standard interfaces used by all software packages. A small support team can thus support a number of development teams more easily.

The separation of the functional and technical architectures provided by the common services model means that a significant amount of the technical architecture can be provided through the common services, allowing developers to concentrate on providing the functional behavior required of their software.

There is a uniform implementation of the technical architecture across all systems. So long as the access to this technical architecture remains consistent, the implementation of the technical architecture can be modified with minimal impact on the development teams.

Since application deployment is a technical issue and hence implemented within the common services, the software system as a whole is more easily managed within a distributed environment. This makes the use of less expensive, commodity computers more feasible.

Another infrastructure approach is a controls model that combines the communications and controls aspects of observatory control. Two illustrations of this model are LabVIEW, used by SOAR, and EPICS, used by Gemini, JACH, and many particle physics accelerators. Both of these control systems provide a rich development environment and are well-suited for real-time control systems.

1.2. STRUCTURE

The ATST Common Services are grouped into several broad categories:

Deployment support—implemented based on a Container/Component Model, this support allows the uniform management of applications in a distributed system without regard to the functionality they provide. Base implementations for software components and controllers are provided as part of the deployment support. All application functionality is implemented on top of these base implementations.

Communications support—services that are necessary or useful in a distributed system. These include:

Connection services that allow applications to communicate directly with other applications, including commanding them to perform specific actions

Notification services that allow applications to publish/subscribe to broadcast messages (events) without explicit knowledge of their recipients/publishers

Logging services that allow applications to record historically useful information

Alarm services that allow applications to broadcast alarm and health messages.

Persistence support—services that allow applications to store and retrieve property information whose lifetimes exceed that of a single instance of the application.

Tools—libraries of software modules that are of common use across multiple system packages.

Application support—support for writing ATST applications. The base implementation (i.e. Component) provides the connection framework to Common Services. An extension (i.e. Controller) handles multiple, simultaneous configurations in a Command/Action/Response model. Either may be extended by developers to add specific functionality and subclasses are already provided to assist in sequencing of actions and real-time device control.

All common services are available for use in three languages: Java, C++, and Python although the access to the services varies with the language.

1.3. Design Highlights

Most of the design of the ATST Common Services Framework is of little interest to software development teams using the common services. However, a quick look at some of the key design features can be informative and also help illustrate some of the power and flexibility provided by ATSTCS. Detailed information on the use of these, and other common services features, can be found in later sections of this document.

1.3.1. Communications-Neutral Architecture

While ATST has selected ICE as the communications middleware that is the foundation for intra-application communications, ATSTCS is designed to operate as independently as possible from the choice of communication middleware. The role of third-party middleware is carefully defined and bounded. This allows ATST to remain flexible on its choice of middleware, such as ICE or CORBA, and to more easily replace one choice with another should it prove advantageous to do so in the future. Component developers should not be concerned with the choice of communications middleware—they reference no middleware-specific features, extend no middleware-specific classes, etc.

1.3.2. Separation of Functional and Technical behavior

The ATST software design distinguishes between functional and technical behavior. Functional behavior describes the actions taken to directly implement ATST operations and can be contrasted with the technical behavior - the actions required of the infrastructure needed to support the functional behavior. For example, logging a specific message into a persistent store is functional behavior—only the application developer can determine what (and when) messages should be logged. The underlying mechanism that performs the logging, however, is technical behavior. By establishing a clear distinction between functional and technical behavior, and providing the technical behavior through the ATST Common Services Framework, the application developer can concentrate on providing the required functionality.

1.3.3. Configuration-Driven Control

A fundamental precept of the ATST software design is the use of configurations to drive ATST control behavior. A configuration is a set of logically-related, named values that describe the target condition of a subsystem. Control of a subsystem is accomplished by directing the subsystem to match the target conditions described by each configuration. The set of available commands is thus kept small and generic - amounting to little more than "match this configuration". Subsystems are responsible for determining how to match the target - all details of sequencing subsystem components are isolated in the subsystem. Subsystems announce that they have met (or cannot meet!) the target using broadcast events.

1.3.4. Container/Component Model

One feature of ATSTCS is its adoption and support of a Container/Component Model (CCM). This approach, also used in the ALMA common services, is based upon the same fundamental design principles as Microsoft's .NET and Java's EJB architectures and simplifies application deployment and execution within a distributed environment. In the CCM, the deployment and lifecycle aspects of an application are separated from the functional aspects of the application. In particular, management applications (containers) are responsible for creating, starting, and stopping one or more functional applications (components). Containers are implemented as part of the common services, as are the base classes used by all components.

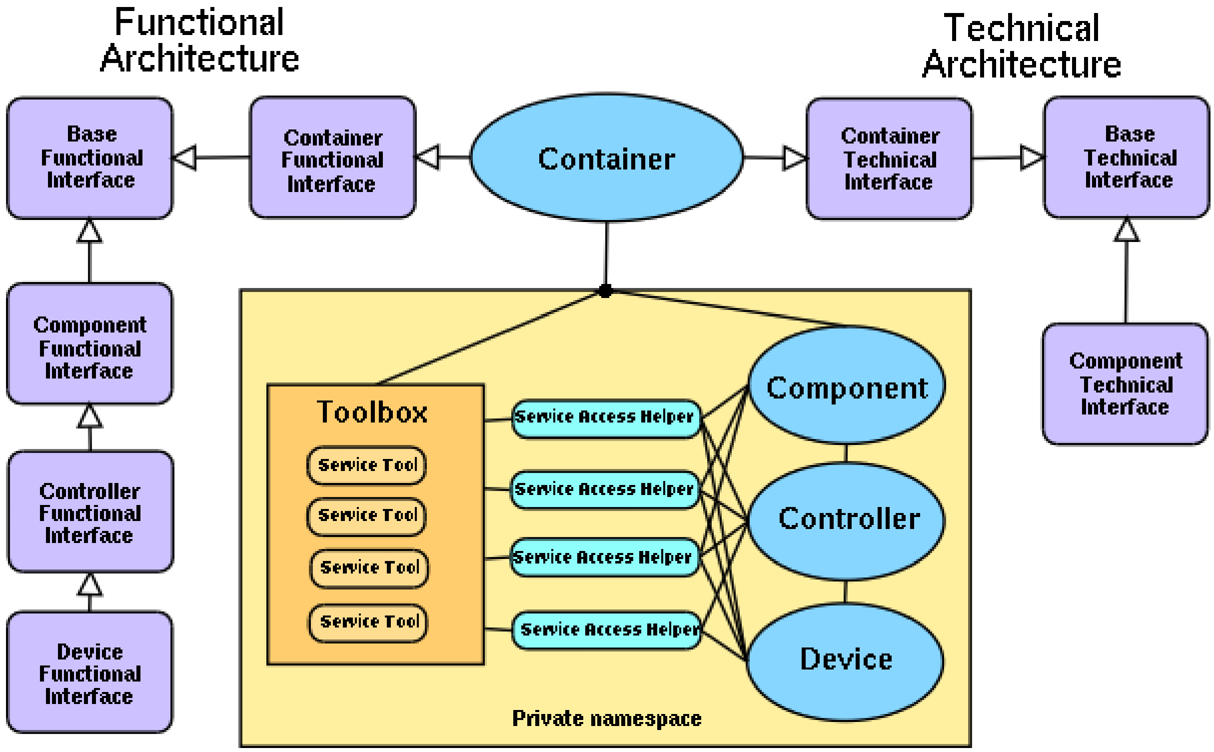

Figure 1. Functional and Technical Architectures of the Common Services

The lifecycle interfaces (appearing on the right of the above diagram) are implemented within the common services design by the underlying infrastructure. This means that developers can concentrate on providing code for performing actions visible through the functional interfaces (on the left of the above diagram). In the vast majority of cases this means subclassing the Controller class, overwriting or implementing any interface methods that access added functionality, and then writing support methods implementing that added functionality.

Components

Components are the foundation for all ATST applications as all ATST functional behavior is implemented within Component subclasses such as Controllers. (A Controller adds configuration management support to a Component.) The bulk of the ATST software effort is in designing and implementing the functionality provided by Components and Controllers.

Containers

Containers provide a uniform means of deploying and controlling the technical aspects of Component operations. Component developers can develop Components without a detailed understanding of Containers.

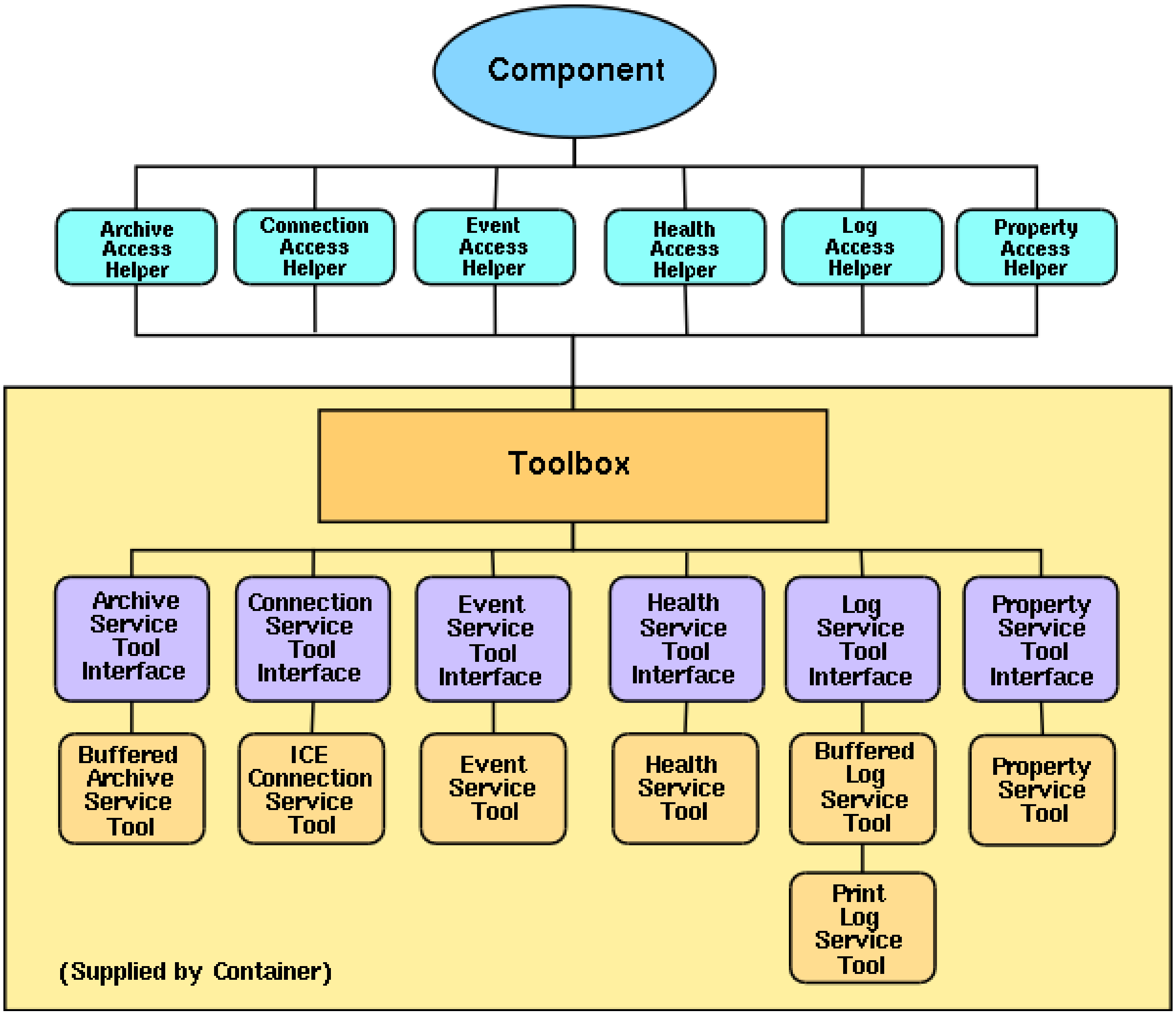

Figure 2. Illustration of the Common Services Toolbox

1.3.5. Service Toolboxes

In ATST, access to services is provided to components through the container. Some services may be shared among components while others might be unique to individual components. It is the container's responsibility to ensure that services are properly allocated. When a component is deployed to a container that container assigns a unique service toolbox to that component, and places service tools into that toolbox. Service tools are modules that understand how to access specific common services. Typically, these tools are small, with well-defined tasks. However, the tools are designed to be chained so that several simple tools can be used to perform complex actions on service access. For example, message logging may be accomplished by chaining a log database tool that logs messages to a persistent store with a filter tool that looks for specific message characteristics. When a message with those characteristics is found, the filter tool might route that message to an operator's console. As a more extreme example (though not one likely to be used in ATST), it is possible to chain a connection service tool for ICE with a connection service tool for CORBA, allowing components to connect seamlessly and simultaneously to both ICE-aware and CORBA-aware modules! A container also retains access to each component's toolbox, permitting dynamic reconfiguration of tools without involving the component itself.

An important characteristic of the toolbox and service tools is that all component specific information needed by the various service tools is maintained in the toolbox, not in the specific service tool. This allows toolboxes to contain service tools that can be shared among components if it is advantageous to do so. For example, message logging may be more efficient if a common logging tool is shared among all the components within a container. It also makes it possible for Containers to retain access to the service tools assigned to a Component, adjusting the services as needed.

While a component is free to directly access the service tools in its toolbox, by far the most common way to access services is through static service access helper classes that are also provided by common services. These classes encapsulate access to the toolbox and its tools within easy to use static methods. It is this access through these access helpers that is discussed in detail in later sections of this document. *Direct access to service tools and the toolbox is intentionally not covered*.

2. INFRASTRUCTURE

2.1. COMMON SERVICES LAYERS

One of the major goals of the ATST Common Services Framework design is to isolate ATST software systems from dependencies on specific infrastructure components. This isolation philosophy provides both flexibility and maintainability and is based on a layered software architecture.

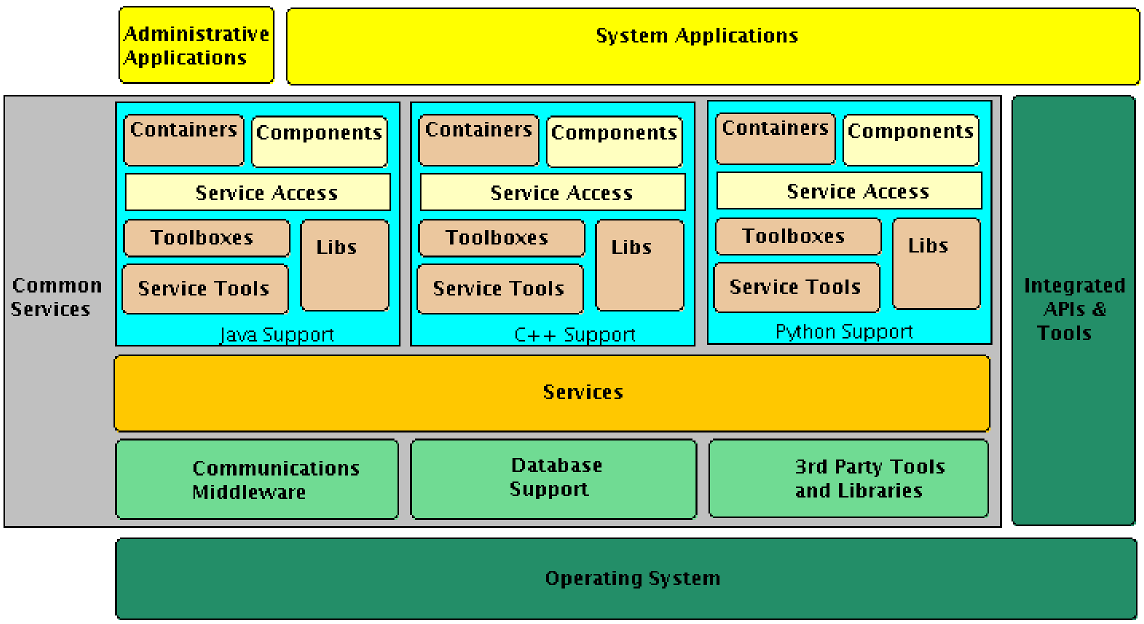

Figure 3. Common Services Layers

Strictly speaking, the top and bottom layers are not part of the Common Services. They are shown to illustrate the depth of support provided by the Common Services layers. In addition to the support available through the Common Services, applications may also use other libraries and tools that have been integrated into ATST software support.

Some of the features of this specific layered architecture are:

The service access layer introduces a simple interface for Container and Component (and hence to application) developers to the common services. The details of the Common Services below this service access layer are not visible in the Component developer's code.

The communications middleware is not tightly woven into the overall design. This makes it possible to easily replace the current communications middleware choice (ICE) with another middle product should it prove advantageous to do so.

The same holds true of the underlying database support.

The services themselves are isolated from applications by the service tools, so service behavior can be customized on a per Container or even a per Component basis as the need arises.

The use of a standard toolbox that holds the service tools allocated to a particular Component makes it easier for Containers to furnish service tools to Components. In fact, Containers may adjust a Component's tool set dynamically.

As much as reasonable, the logical structure is preserved across languages, making it easier for ATST software maintainers to understand the operation of an application, regardless of its implementation language. This is not, however, carried to an extreme - C++ and Java have their own "style" and the language support attempts to maintain this style.

The design allows the fundamental characteristics of the Common Services to be reasonably independent of ATST specifics. By replacing pieces of one layer or another, the Common Services can be adapted to use in other projects, or accommodate major changes to the overall ATST software design.

2.2. Containers and Components

Components implement functionality needed to operate ATST systems. Containers organize and manage Components.

2.2.1. Containers

In ATST, Containers are part of the technical architecture and have the following responsibilities:

Manage Component lifecycles, including creating, starting, stopping, and destroying Components. This means that there is a uniform mechanism for controlling Component lifecycle operations. Multiple Components may run under the aegis of the same Container, but each Component operates in its own namespace (with the exception of interface definitions which are always shared across all namespaces).

Provide services to the Components. Because a Component may share some services between components, more efficient use of services is possible.

Containers are language-specific, there are separate Containers for Java-, C++-, and Python-based Components.

Figure 2 shows the basic structure of all ATST Containers.

The Container's lifecycle control module is responsible for managing the Component lifecycles. It responds to requests on the Container to create, and destroy specific Components. The Component Manager handles these external requests for the Container, calling on the Component Loader to create each Component. External requests generally come from Container Managers provided by the ATST OCS.

The Component Loader works by establishing a new namespace for the Component and then creating the Component and a Toolbox for that Component within that new namespace. The Component Loader then instructs the Toolbox Loader (within the service access control module) to load that Toolbox with service tools. Individual tools may be private (existing within the Component's namespace) or shared (existing outside the Component's namespace). It is the Toolbox Loader that determines which tools are private and which are shared—different Toolbox Loaders may make different determinations.

An important point is that the Container, through the Toolbox Manager module, retains access to each Component's Toolbox—and hence to all the service tools. This allows a Container to dynamically adjust service properties on a per-Component basis.

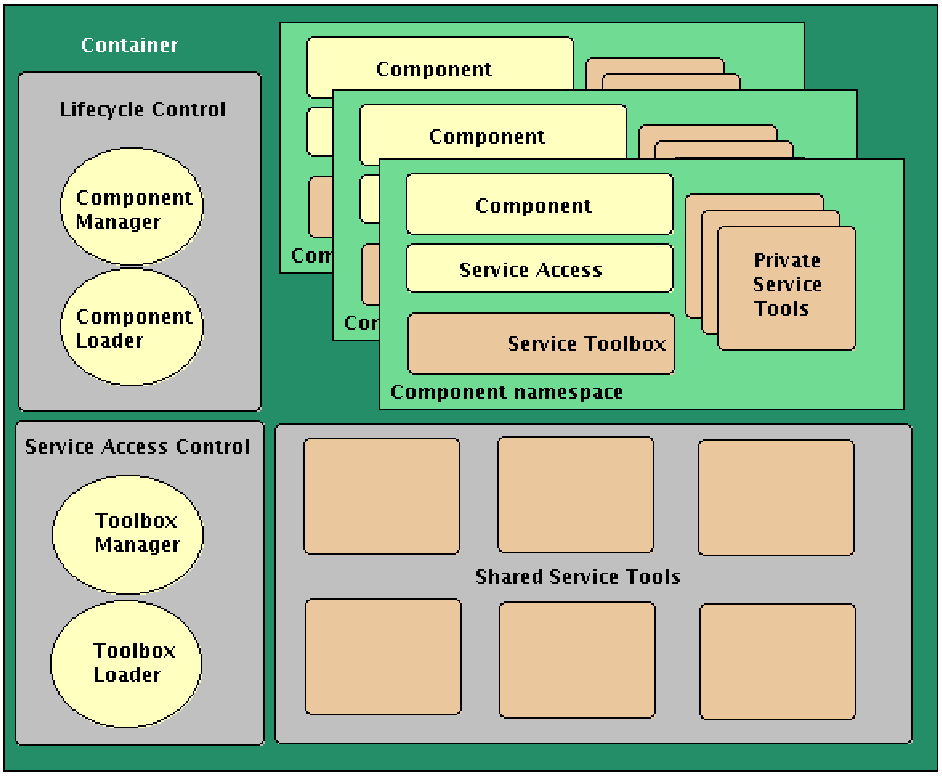

Figure 4. The basic structure of a container

2.2.2. Components

Components are used to implement ATST functionality. Common Services support for Component development is limited primarily to implementation of the base classes (and a few key subclasses, such as Controllers) from which all ATST Components are derived. The Users' Manual includes information on writing Components and Controllers while the sections on Components and Controllers in this guide discuss their implementation.

2.3. Service Access

A Component accesses the Common Services through access helpers. These are modules that simplify service access by wrapping the actions provided by the Toolbox. Each service has its own access helper. Service access helpers are covered in more detail in the ATST Common Services Users' Manual.

2.4. Toolboxes and Service Tools

As noted above, every Component has its own Toolbox associated with it. The Toolbox provides an interface used by the service access helpers and acts as a holder for the specific service tools that have been assigned to the Component by the Toolbox Loader. Since there is a 1-1 association between Toolboxes and Component, the Toolbox is also a convenient place to retain Component-specific knowledge needed by the various services and service helpers. For example, the Component name is retained within the Toolbox as is a reference back to the enclosing Container. (These items are not easily held within the service helpers because service helpers may end up being shared among multiple Components.)

When a Toolbox method is called from a service helper (i.e. by a Component), the Toolbox method typically performs some value-added actions before invoking the appropriate service tool. For example, when the Component uses the Log service to record a message, the Toolbox automatically computes a timestamp and passes the timestamp, the name of the Component, and the message to the log service tool. This value-added capability of the Toolbox is one of the reasons that service access modules can present simplified service access methods to Components.

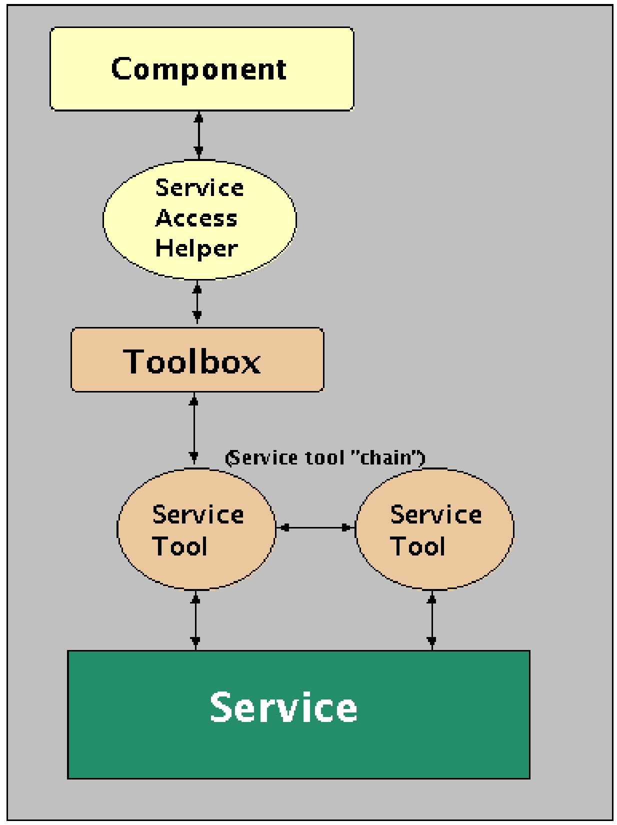

Figure 5. Toolbox Usage

The service tools themselves act as the intermediary between the Component (through the access helper and the Toolbox) and the underlying service. Only the service tools for a service know the details of actually accessing the service. For example, log service tools know how to communicate with the logging service. More than one service tool may exist for a given service. For example, a "stub" log service tool may not access the log service at all, but may instead simply print the message (and Component name and timestamp) to standard error. Another log service tool might filter severe log messages and perform some additional action (perhaps posting an event). Yet another might buffer log messages into blocks before writing them into the log persistent store (for efficiency reasons).

Generally speaking, service tools are kept small and simple, performing some simple, well-defined task. They can, however, be chained together so that complex actions may be performed. The Toolbox Loader decides which service tools should be loaded into a Component's Toolbox and whether multiple service tools for a service should be chained together. It is possible to chain both private and shared tools together. These chains of service tools can also be manipulated dynamically by the Service Access Control module in the Container.

The specific service tools currently available for a service are covered in detail as part of the discussion of that service.

2.5. SERVICES

Since the only interface to the services is through the service tools via the Toolbox, there is a great deal of latitude in the design of the individual services. In fact, alternative implementations of the services are possible and accessible through different service tools. For example, the standard log service is implemented through direct database access for efficiency and robustness. However, an indirect implementation through a proxy application is possible and suitable for situations where direct database access is infeasible (such as on some real-time systems).

ATST somewhat arbitrarily divides services into two categories:

major services are those that are expected to be used by all Components

minor services are those that might be useful in specific Components

The minor services are commonly referred to as tools (note that these tools are distinct from the service tools discussed earlier), while the major services are referred to simply as services.

Details on individual services can be found in the guide sections on services and tools.

2.6. COMMUNICATIONS MIDDLEWARE

Many of the ATST services are built on top of communications middleware. ICE (Internet Communications Engine) is the middleware used by ATST. ICE features are extended and enhanced by ATST Common Services Framework code, but the result is still referred to here as ICE. The tiered architecture of the Common Services makes it possible to encapsulate ICE-specifics and keep the use of ICE invisible to layers above the service tools. For example, ATST Components do not extend ICE objects and do not use any of the ICE-specific interface definitions. The implementation approach is to use wrapper classes that translate between the ATST communications model and the ICE model. It is this implementation approach that frees the ATST Common Services Framework from being inexorably tied into ICE and allows for the possibility of future upgrades to other communications middleware products.

Details on the implementation of the communications layer in ATST can be found in the guide section on internals.

2.7. DATABASE SUPPORT

The ATST Common Services Framework make heavy use of persistent stores. For example, all log messages are recorded in a persistent store, property meta-data in another persistent store, archived data in yet a third, etc. As with the communication services, these persistent stores are implemented on top of support software. In the case of the persistent stores this support software is the PostgreSQL relational database system. ATST Common Services Framework provides a small, vendor-neutral database interface on top of PostgreSQL. As with the approach described above to isolate dependencies on ICE, the same approach here isolates the dependencies on PostgreSQL, with the same potential upgrade benefits.

Details on the implementation of the database support in ATST can be found in the guide section on internals.

2.8. THIRD PARTY TOOLS

In addition to ICE and PostgreSQL, the implementation of the ATST Common Services Framework makes use of other 3rd party tools. As with ICE and PostgreSQL, these third party tools are isolated within the tiered architecture to permit their replacement should better alternatives appear. These 3rd party tools are covered in more details in the internals section of this guide.

Component developers are likely to find other 3rd party tools that are useful for specific applications. Often such tools would be useful to other developers as well and hence good candidates for inclusion in the Common Services. The administration section of this guide discusses how to integrate such tools into the ATST Common Services Framework.

2.9. COMMON DATA STRUCTURES

While most data structures used in ATST are language specific, some data structures are passed between applications and so must be capable of automatic translation from one language to another. These data structures are transferred by having the appropriate connection service tool map the contents of those structures to and from ICE compliant structures. For this reason the set of common data structures is kept small, consistent, and generic.

2.9.1. Attributes

The base of the set of common data structures is the *Attribute*. Attributes are (name, value) pairs where the value is a vector of string values. In most instances this vector has a single element but there is no limit imposed on the number of elements. The primary mirror actuator force map, for example, might be (but then again, might not) be represented by a single Attribute whose value is a vector of over a hundred actuator force elements.

Note that all values are strings. This is a policy of ATST communications that all information be transferred between applications using a string representation. The communication service access helpers include support methods for converting data objects of other types to and from string representations.

Attributes can also have metadata associated with them, but this metadata is not transmitted as part of the Attributes themselves. Instead, the metadata for a given Attribute can be obtained from the property service persistent store. Attribute metadata is information that is useful for the processing of that Attribute and typically includes limits, datatype (where the datatype is an ATST logical datatype, not necessarily a language specific datatype), enumerated values (for datatypes with only a small set of discrete values), and similar information.

2.9.2. AttributeTables

Sets of logically-related Attributes are collected into *AttributeTables*. For example, all the information that is required to perform a specific action might be collected into a single AttributeTable. AttributeTables are unordered collections of Attributes. Only a single Attribute with a given name can exist in an AttributeTable. Locating an Attribute within an AttributeTable is a O(1) operation.

2.9.3. Configurations

An important, widely-used in ATST, extension of AttributeTable is the *Configuration*. Configurations are AttributeTables with additional, standard information attached. Every Configuration is uniquely identified, contains identification of the entity that created it, and when it was created. Details of the roles of these items are covered on the section on ***Controllers***.

3. MAJOR SERVICES

3.1. LOG SERVICE

The log service provides the ability to record messages into a persistent store. Typically these messages fall into two general types:

informative - messages that provide information about the normal operation of ATST

diagnostic - messages that provide information useful in diagnosing specific problems

Diagnostic messages are typically referred to as debug messages.

The log service access helper that provides a simple log service interface for Components is covered in detail in the Users' Manual. This section focuses on the underlying architecture of the log service.

3.1.1. Log service dependencies

The base log service tool is implemented directly on top of the Common Services' database support and makes use of no other services. Consequently, the base log service tool is always loaded first by any Toolbox Loader so that the log service is available to other service tools.

Note that variant log service tools may depend upon the presence of other services. For example, a proxy log service tool would require access to the connection service, so a Toolbox Loader working with a proxy log service tool would need to restructure the order in which the service tools are loaded.

3.1.2. The log database

The log database contains single active table with one tuple per message. Older messages are archived into inactive tables with the same structure—the frequency of transferring messages from the active table to an inactive table is a policy decision that should be established as part of the ATST operations model. At the current time it is TBD.

The log database is optimized for insertion, not for queries on the assumption that insertions occur far more frequently than queries.

Log message tuples contain the fields (see the log service discussion in the Users' Manual for detailed information on the meaning of category, mode, and level):

time_stamp—the time at which the message was created (set by the Toolbox)

source—the name of the Component that produced the message (set by the Toolbox)

category—the message category

mode—the mode of the message (note, warning, severe, or debug)

level—the level of the message if the mode is debug, undefined otherwise

message—the actual message as an arbitrary-length string

UID—serial field to allow messages coming out faster than once per millisecond to be ordered (set by the DB).

Time stamps are recorded down to the millisecond level. Although the level is visible to Components as an integer value, it is recorded in the log database as a string.

Please note that the category field is assumed to be a simple string, and is not sanitized before being inserted into the DB. If it is not a simple string (e.g. it contains \t, \n, or ‘) then the behavior is undefined.

3.1.3. Log service tools

The log service tools that are available vary from one implementation language to another and are covered under the appropriate language support sections below.

3.1.4. Java support

3.1.4.1. Log service tools

All the log service tools subclass atst.cs.services.log.AbstractLogServiceTool and implement atst.cs.services.log.ILogServiceTool. All of the log service tools are found in the Java package atst.cs.services.log. That prefix is omitted from this discussion for brevity.

The following Log service tools are available to Java-based Toolboxes (remember that service tools can be chained):

NullLogServiceTool—this tool does absolutely nothing. Its primary use is in testing interface behavior above the service tool layer and as a means to completely disable all logging. It is rarely used.

PrintLogServiceTool—this tool sends all log messages to standard error.

LogServiceTool -- this tool records log messages to the log database individually as they are produced. This is a slow, but reliable, means of recording log messages. Because it is slow, it is commonly loaded as a private service tool to take advantage of parallel execution.

BufferedLogServiceTool—this tool buffers log messages until either (a) a fixed amount of time has passed (default is 0.5 seconds) or a set number of log messages have been received (default is 5000 messages). It is highly optimized for performance and performs double buffering. It is slightly less reliable then LogServiceTool since log messages produced in the last 0.5 seconds may be lost on a Component crash. However, since this is a rare occurrence and the performance is so much better (factor of 10 or more) than LogServiceTool, this is the preferred log service tool. The BufferedLogServiceTool is usually loaded as a shared service tool but can also operate as a private service tool.

Both the LogServiceTool and the BufferedLogServiceTool use a pool of database connections to improve performance. Also, while it is possible to chain LogServiceTool and BufferedLogServiceTool, it makes no sense to do so. The PrintLogServiceTool is often chained with one of them.

3.1.4.2. Toolbox access to the log service tools

Once the Toolbox Loader has loaded the appropriate chain of log service tools, the Toolbox uses the atst.cs.services.log.ILogServiceTool interface to access the first (possibly only) log service tool in the chain. There is only one method of interest in this interface:

```
void log(IToolBoxAdmin tb, String timeStamp, String source, String mode,
      String category, int level, String message)
```

The Toolbox fills in the timeStamp and source parameters while passing the remaining parameters on from the Log service access helper.

3.1.4.3. Log service helper access to the Toolbox

These are the methods provided by the atst.cs.interfaces.IToolBox interface that are used by the atst.cs.services.Log access helper:

```
void note(String category, String message)
void warn(String category, String message)
void severe(String category, String message)
void debug(String category, int level, String message)
void setDebugLevel(String category, int level)
int getDebugLevel(String category)
```

3.1.4.4. Component access to the log service

See the Users' Manual for details on how Components use the atst.cs.services.Log class to access the log service.

3.1.5. C++ support

3.1.5.1. Log service tools

All the log service tools subclass atst::cs::services::log::AbstractLogServiceTool and implement atst::cs::services::log::ILogServiceTool. All of the log service tools are found in the namespace atst::cs::services::log. The namespace is omitted from this discussion for brevity.

The following Log service tools are available to C++based ToolBoxes (remember that service tools can be chained):

NullLogServiceTool—this tool does absolutely nothing. Its primary use is in testing interface behavior above the service tool layer and as a means to completely disable all logging. It is rarely used.

PrintLogServiceTool—this tool sends all log messages to standard error.

LogServiceTool -- this tool records log messages to the log database individually as they are produced. This is a slow, but reliable, means of recording log messages. Because it is slow, it is commonly loaded as a private service tool to take advantage of parallel execution.

BufferedLogServiceTool—this tool buffers log messages until either (a) a fixed amount of time has passed (default is 0.5 seconds) or a set number of log messages have been received (default is 5000 messages). It is highly optimized for performance and performs double buffering. It is slightly less reliable then LogServiceTool since log messages produced in the last 0.5 seconds may be lost on a Component crash. However, since this is a rare occurrence and the performance is so much better (factor of 10 or more) than LogServiceTool, this is the preferred log service tool. The BufferedLogServiceTool is usually loaded as a shared service tool but can also operate as a private service tool.

3.1.5.2. ToolBox access to the log service tools

Once the Toolbox Loader has loaded the appropriate chain of log service tools, the ToolBox uses the atst::cs::services::log::ILogServiceTool interface to access the first (possibly only) log service tool in the chain. There is only one method of interest in this interface:

```
void log(std::tr1::shared_ptr<IToolBoxAdmin> tb,
            const std::string& timeStamp,
            const std::string& source,
            const std::string& mode,
            const std::string& category,
            int level,
            const std::string& message)
```

The ToolBox fills in the timeStamp and source parameters while passing the remaining parameters on from the Log service access helper.

3.1.5.3. Log service helper access to the ToolBox

There are three methods provided by the atst::cs::interfaces::IToolBox interface that are used by the atst::cs::services::Log access helper:

```
void note(cosnt std::string& category, cosnt std::string& message)
void warn(cosnt std::string& category, const std::string& message)
void severe(const std::string& category, Const std::string& message)
void debug(Const std::string& category, int level, Const std::string& message)
void setDebugLevel(Const std::string& category, int level) int
      getDebugLevel(Const std::string& category)
```

3.1.5.4. Component access to the log service

See the Users' Manual for details on how Components use the atst::cs::services::Log class to access the log service.

3.1.6. Auxiliary support

There are a few additional tools provided by the Common Services for working with the log service. These tools are primarily for use during development and will be enhanced or replaced by the ATST OCS.

ArchiveView—the ArchiveView application presents a simple GUI for manipulating the log database archive tables discussed above. The user can see the list of existing archive times and their sizes, create a new archive table, and move data from the active table to this new archive table.

LogView—the LogView application presents a simple GUI for viewing selected messages from the log database active table. Log messages can be selected by combinations of time range, source Component name, category, and mode.

3.2. CONNECTION SERVICE

The connection service allows Components to connect to each other in a fully distributed system. The source Component needs no knowledge of the target Component's location, just the type of interface that the target Component presents. There are five different interfaces that the connection service can return:

IRemote interface – The most basic interface. It only defines get/set, enable/disable logging and get/set debug level. All remote objects are a subclass of this.

IComponent interface – Extends the IRemote interface and adds lifecycle methods, and get language, host, container.

IController interface – Extends the IComponent interface and adds the submit and other related methods.

IContainer interface—The container interface allows components to be loaded and allows an alternative means of managing the lifecycle of a component.

IContainerManager interface – These are rare and used mostly by the technical architecture. They provide methods to allow the management of containers.

The connection service also allows Containers to register Components, making them available for remote access, and later to unregister any Component so it is no longer available for remote access. An important aspect of the connection service is the embedded command system that supports the remote operations on a target Component.

The connection service access helper that provides Component developers with a simple interface to the connection service is covered in detail in the Users' Manual. This section focuses on the underlying architecture of the connection service.

3.2.1. Connection service dependencies

The base connection service tool is implemented on top of the ICE communications middleware and uses both the log service and the id service and so must be loaded by the Toolbox Loader after the log and id service tools. The connection service itself may be used by other services.

The connection service tool masks the use of ICE by encapsulating the ICE object that represents the remote Component within an ATST object providing the proper interface. So, when a source Component connects to a target Component, it obtains access to the encapsulating object. Functionally, however, this is indistinguishable from a direct connection to the target Component.

3.2.2. The connection service naming system

The connection service maintains a global repository mapping Component names with the connection to that Component. The interface that each Component presents to other Components is also maintained as part of this mapping. Component registration with the connection service, then, is simply the recording of the name -> (Component, Interface) mapping in this repository. A source Component connects to a target Component, via the connection service tool, by contacting the connection service and requesting the target Component by name. The connection service returns the (Component, Interface) pair to the service tool, which encapsulates the target Component into the appropriate ATST object with a matching Interface. The encapsulation is returned to the source Component, which then uses this object for direct, peer-to-peer, communication.

3.2.3. The connection service command system

The interfaces used for Component connections provide a set of methods for commanding the target Component. The specific methods depend upon the particular interface presented by the target Component and are covered in more detail in the section on Components in this guide.

3.2.4. Connection service tools

The connection service tools that are available vary from one implementation language to another and are covered under the appropriate language support section below.

3.2.5. Java support

3.2.5.1. Connection service tools

There is only one connection service tool of interest:

atst.cs.ice.IceConnectionServiceTool—provides connection services layered on top of ICE communications middleware. This tool establishes (using ICE) the connection and constructs and returns the encapsulating object. It is also responsible for closing the connection and releasing resources.

The IceConnectionServiceTool may be either a private or a shared service tool.

3.2.5.2. Toolbox access to the connection service tool

The Toolbox uses the atst.cs.services.app.IConnectionServiceTool interface to access the connection service tool. The methods of interest in this interface are:

```
// Makes the named object available for remote access
void bind(IToolBoxAdmin tb, String name, atst.cs.interfaces.IRemote object)

// Makes the named object unavailable for remote access and releases
// any unneeded resources.
void unbind(IToolBoxAdmin tb, String name)

// Connect to the named target object.  The name of the source Component
// is provided in case the connection service tool is shared.
atst.cs.interfaces.IRemote connect(IToolBoxAdmin tb, String source, String
      target)

// disconnect from the named target object.  The name of the source Component
// is provided in case the connection service tool is shared.
void disconnect(IToolBoxAdmin tb, String source, String target)
```

In the case of connect and disconnect methods, the name of the target object is supplied to the Toolbox by the connection service access helper. The Toolbox fills in the name of the source object in all of the above methods.

3.2.5.3. Connection service helper access to the Toolbox

The following methods provided by the atst.cs.interfaces.IToolbox interface that are used by the connection service access helper:

```
// Register the Component object
void bind(atst.cs.interfaces.IRemote object)

// Unregister the Component
void unbind()

// connect to the named target
atst.cs.interfaces.IRemote connect(String targetName)

// disconnect from the named target
void disconnect(String targetName)
```

3.2.5.4. Component access to the connection service

See the Users' Manual for details on how Components use the atst.cs.services.App class to access the connection service.

3.2.6. C++ support

There is only one connection service tool of interest:

atst::cs::ice::IceConnectionServiceTool—provides connection services layered on top of ICE communications middleware. This tool establishes (using ICE) the connection and constructs and returns the encapsulating object. It is also responsible for closing the connection and releasing resources.

The IceConnectionServiceTool may be either a private or a shared service tool.

3.2.6.1. ToolBox access to the connection service tool

The ToolBox uses the atst::cs::services::app::IConnectionServiceTool interface to access the connection service tool. The methods of interest in this interface are:

```
// Makes the named object available for remote access
void bind(const string& name, atst::cs::interfaces::IRemote& object);


// Makes the named object unavailable for remote access and releases
// any unneeded resources.
void unbind(const string& name);


// Connect to the named target object.  The name of the source Component
// is provided in case the connection service tool is shared.
tr1::shared_ptr<atst::cs::interfaces::IRemote>
connect(const string& source, const string& target);


// Disconnect from the named target object.  The name of the source Component
```

```
// is provided in case the connection service tool is shared.
void disconnect(const string& source, const string& target);
```

In the case of the connect and disconnect, the name of the target object is supplied to the ToolBox by the connection service access helper. The ToolBox fills in the name of the source object in all of the above methods.

3.2.6.2. Connection service helper access to the ToolBox

The following methods provided by the atst::cs::interfaces::IToolBox interface that are used by the connection service access helper:

```
// Register the Component object
void bind(atst::cs::interfaces::IRemote& object);

// Unregister the Component
void unbind();

// Connect to the named target
std::tr1::shared_ptr<atst::cs::interfaces::IRemote>
connect(const std::string& target);

// Disconnect from the named target
void disconnect(const string& targetName);
```

3.2.6.3. Component access to the connection service

See the Users' Manual for details on how Components use the atst::cs::services::App class to access the connection service.

3.3. EVENT SERVICE

ATST relies heavily on the event service to provide non peer-to-peer communications. The event service provides a publish/subscribe mechanism that is robust and high-performance. There are two basic operations:

Publishing—information is broadcast without regard to recipients.

Subscribing—information is received without regard to sources.

Each piece of information is packaged as a named event consisting of an AttributeTable holding the information as a set of Attributes.

A callback mechanism is used by subscribers to perform actions on receipt of an event. The event service guarantees that these callbacks are invoked in the order in which events are received - even if events need to be queued within the subscriber. For this reason, processing within a callback should be kept short - more involved actions should be performed using some other processing thread.

Events that are posted when there are no subscribers are lost. Further, a subscriber only sees events posted during the time that they are actively subscribed. The event service does not queue events for delivery to "late" subscribers.

All events are automatically timestamped when posted. This timestamp is recorded when events are logged or archived, but is not generally available to application code.

The event service access helper that provides Components with easy access to the event service is covered in detail in the Users' Manual. This section focuses on the underlying architecture of the event service.

3.3.1. Event service dependencies

The event service is implemented on top of the ICE communications middleware and uses the log service. It must be loaded after the log service by the Toolbox Loader.

3.3.2. Event service tools

The event service tools that are available vary from one implementation language to another and are covered under the appropriate language support section below.

3.3.3. Java support

3.3.3.1. Event callbacks

All event callbacks must subclass atst.cs.services.event.EventCallbackAdapter (which implements the atst.cs.interfaces.IEventCallback interface) and override the method:

void callback(String eventName) —process the named event

Note that the event value is not passed in. Instead, there are supplied methods for accessing the event's value. While the value is always an AttributeTable, it is common for that AttributeTable to contain a single Attribute (since the service access helper provides support methods for posting single-valued events). Some of the callback methods represent inverse operations for those access helper methods:

atst.cs.data.IAttributeTable getValue()—produce the entire value as an AttributeTable.

atst.cs.data.IAttribute getAttribute(String attributeName) —get the named Attribute.

The last four methods can be used with their companion event posting methods found in the atst.cs.services.Event event service access helper class, by using the event name as the value for the attributeName parameter. Also, getAttribute, getString, getDouble, and getLong return null if they cannot provide the value in the indicated form.

See the Users' Manual for details on the atst.cs.services.Event class, as well as more detail on the above methods.

3.3.3.2. Event service tools

All the event service tools subclass atst.cs.services.event.AbstractEventServiceTool and implement atst.cs.services.event.IEventServiceTool.

There are several event service tools:

atst.cs.services.event.SimpleEventServiceTool—a stub that writes events to standard output. This tool can be chained with another event service tool and can be loaded as either a private or a shared service tool.

atst.cs.services.event.EventLogServiceTool—logs event postings and subscriptions. This tool can be loaded as either a private or a shared service tool.

atst.cs.ice.IceEventServiceTool—implements the event service over the ICE communications middleware. This tool can be loaded as either a private or a shared service tool.

atst.cs.ice.IceBatchEventServiceTool—subclasses the IceEventServiceTool and is useful for situations when increased throughput is desired and event ordering is not required. This tool batches events sent to the event server. The number of events in one batch is configurable. The default batch size is 10,000 events or 1/2 second, whichever comes first. This tool can be loaded as either a private or a shared service tool.

atst.cs.corba.jaco.JacoEventServiceTool—implements the event service over the CORBA communications middleware. This tool can be loaded as either a private or a shared service tool.

The EventLogServiceTool service tool simply logs event activity - it is certainly always used chained to an event service tool that actually processes events or to test a Component's event service access during development. Because the posting of log messages as events can put a heavy strain on the event service, it should be used with care.

3.3.3.3. Toolbox access to the event service tools

The Toolbox uses the atst.cs.services.event.IEventServiceTool interface to access the event service tool. This interface defines the methods:

`void subscribe(IToolBoxAdmin tb, String source, String name, atst.cs.interfaces.IEventCallback callback)` - subscribe the source Component to the named event, using the specified callback to process received events.

`void unsubscribe(IToolBoxAdmin tb, String source, String name, atst.cs.interfaces.IEventCallback callback)` - unsubscribe the source Component from the named event, using the specified callback to process received events.

`void unsubscribeAll(IToolBoxAdmin tb, String source)` -unsubscribe the source Component from all events.

`void post(IToolBoxAdmin tb, String timestamp, String source, String eventName, atst.cs.data.IAttributeTable message)` - post the event from the source Component.

The Toolbox fills in the source parameter on all of these methods and the timestamp parameter on the post method. The other parameters are passed on from the Component via the event service access helper.

Note unsubscribe requires the specific callback that is being unsubscribed. It is possible for a Component to subscribe multiple callbacks to the same event. Different widgets in a GUI, for example, may each want to perform some unique action on receipt of the same event. The unsubscribeAll method is typically called by the Toolbox when the Component is shutdown.

3.3.3.4. Event service helper access to the Toolbox

The atst.cs.interfaces.IToolbox interface is used by the event service access helper to access the event service via the Toolbox. The following methods in that interface are of interest:

```
// subscribe the callback to the named event
void subscribe(String eventName, atst.cs.interfaces.IEventCallback callback)

//unsubscribe the callback from the named event
void unsubscribe(String eventName, atst.cs.interfaces.IEventCallback callback)

//post an the supplied attribute table as an event with the given name
void post(String eventName, atst.cs.data.IAttributeTable value)
```

3.3.3.5. Component access to the event service

See the Users' Manual for details on how Components use the atst.cs.services.Event class to access the event service.

3.3.4. C++ support

3.3.4.1. Event callbacks

All event callbacks must subclass atst::cs::services::event::EventCallbackAdapter (which implements the atst::cs::interfaces::IEventCallback interface) and overrides the method:

```
//process the named event
void callback(const string& eventName);
```

Note that the event value is not passed in. Instead, there are supplied methods for accessing the event's value. While the value is always an AttributeTable, it is common for that AttributeTable to contain a single Attribute (since the service access helper provides support methods for posting single-valued events). Some of the callback methods represent inverse operations for those access helper methods:

```
// Produce the entire value as an AttributeTable

std::tr1::shared_ptr<IAttributeTable> getValue();


// Get the named Attribute

std::tr1::shared_ptr<IAttribute> getAttribute(String attributeName);
```

See the Users' Manual for details on the atst::cs::services::Event class, as well as more detail on the above methods.

3.3.4.2. Event service tools

All event service tools implment atst::cs::services::event::EventServiceTool. Currently, there is only one event service tool for C++ based Components:

atst::cs::ice::IceEventServiceTool—implements the event service over the ICE communications middleware. This tool can be loaded as either a private or a shared service tool.

3.3.4.3. ToolBox access to the event service tools

The ToolBox uses the atst::cs::services::event::IEventServiceTool interface to access the event service tool. This interface defines the methods:

```
// Subscribe the source Component to the named event, using the specified

// callback to process received events

void subscribe(const string& source,

const string& eventName,

tr1::shared_ptr<IEventCallback> callback);


// Unsubscribe the source Component from the named event, using the specified

// callback to process received events

void unsubscribe(const string& source,

const string& eventName,

tr1::shared_ptr<IEventCallback> callback);


// Unsubscribe the source Component from all events.
```

```
void unsubscribeAll(const std::string& source);


// Post the event from the source Component
void post(const string& timestamp,
const string& source,
const string& eventName,
tr1::shared_ptr<IAttributeTable> value);
```

The ToolBox fills in the source parameter on all of these methods and the timestamp parameter on the post method. The other parameters are passed on from the Component via the event service access helper.

Note unsubscribe requires the specific callback that is being unsubscribed since it is possible for a Component to subscribe multiple callbacks to the same event. The unsubscribeAll method is typically called by the ToolBox when the Component is shutdown.

3.3.4.4. Event service helper access to the ToolBox

The atst::cs::interfaces::IToolbox interface is used by the event service access helper to access the event service via the ToolBox. The following methods in that interface are of interest:

```
// subscribe the callback to the named event
void subscribe(String eventName, atst.cs.interfaces.IEventCallback callback)

//unsubscribe the callback from the named event
void unsubscribe(String eventName, atst.cs.interfaces.IEventCallback callback)

//post an the supplied attribute table as an event with the given name
void post(String eventName, atst.cs.data.IAttributeTable value)
```

3.3.4.5. Component access to the event service

See the Users' Manual for details on how Components use the atst::cs::services::Event class to access the event service.

3.4. HEALTH SERVICE

The Health service provides a uniform mechanism for monitoring ATST applications. It provides a watchdog service for determining availabilityof a Component and a means of establishing the operational condition of the Component.

A Component's health status is one of:

GOOD—the Component is up and running to within specifications

ILL—the Component is up, but running at reduced capability

BAD—the Component is up, but has severe problems that prevent it from operating correctly. Corrective action is required.

UNKNOWN—the Component is not responding

Components do not report their own health status, but are expected to provide information about what their health might be which will be used in the determination of their health status. In particular, Component developers should call Health.set() whenever they encounter a problem (or lack thereof) that indicates the component’s health. The Health.set method takes a string which is the category, and a HealthStatus object which represents one of the supported ATST healths: GOOD, ILL, BAD, UNKNOWN. The health service then looks at all categories, and choses the worst to report. A controller for example might in one thread monitor the number of resources available in some pool, and in another thread attempt to communicate with hardware. If the resources became low the health for the 'resource' category could become ILL. Even if hardware communications were successful the component's overall health would be ILL. If the hardware communications had failed, the health for the 'hw-comm' category could become BAD. In this situation the BAD health for 'hw-comm' would be the component's health and that is what is reported back up. A subsequent change of the 'resource'-category health to GOOD would have no effect on the overall health. Developers are free to create as many or as few health categories as they wish.

In an hierarchical organization of Components, a Component's health does not depend upon the health of its child Components. The propagation of health through the hierarchy is performed by the health service outside of the operation of any Component.

The health service operates as a separate task (or thread) running in parallel with the Component. At a periodic interval the health service polls the healths for all categories and determines the worst health to report. The polling interval is controlled by the health service but could be different for different Components and could support changing dynamically.

Changes to health are automatically logged by the health service. The log service includes both the category that changed, and the overall health of the component. The log messages get progressively more dire as the health worsens. The posted event's name is "__health" and contains a health report. A health report includes both the current health status, as well as the health status at the last polling.

3.4.1. Health service dependencies

The health service depends upon the Log, and Event services and must be loaded after these services have been loaded by the Toolbox Loader.

3.4.2. Health service tools

The health service tools that are available vary from one implementation language to another and are covered under the appropriate language support section below.

3.4.3. Java support

3.4.3.1. The Health.set() method

Component developers should call the Health.set method whenever the components health has changed or might potentially have changed.

static void atst.cs.services.Health.set (String category, IHealthStatus status)

See the javadoc for atst.cs.data.HealthStatus details on creating an IHealthStatus object. There are a number of factory constructors for the various health types.

The atst.cs.services.Health class provides the following static methods:

IHealthStatus atst.cs.services.Health.get(String category) – get the health of the given category for this component.

void atst.cs.services.Health.set(String category, IHealthStatus status) – sets the health of the given category for this Component

IHealthStatus atst.cs.services.Health.getWorst() – get the worst health available for this component.

void atst.cs.services.Health.setGood(String category) – sets the health of the given category to good for this Component. This is a convenience method.

void atst.cs.services.Health.setIll(String category, String reason) – sets the health of the given category to ill and sets the reason for this Component. This is a convenience method.

void atst.cs.services.Health.setBad(String category, String reason) – sets the health of the given category to bad and sets the reason for this Component. This is a convenience method.

The current health of a Component is maintained in that Component's Toolbox.

3.4.3.2. Health service tools

There are several health service tools available, including:

atst.cs.services.health.LoggingHealthServiceTool—logs health changes

atst.cs.services.health.PostingHealthServiceTool—post health changes

3.4.3.3. Toolbox access to the health service tools

The Toolbox uses the atst.cs.services.health.IHealthServiceTool interface to access the health service tool. The method defined by this interface is:

void changeHealth(IToolboxAdmin toolbox, String category, IHealthStatus oldHealth, IHealthStatus newHealth) – carry out any behavior based on a, potential, change of health.

3.4.3.4. Health service access to the Toolbox

The atst.cs.interfaces.IToolbox interface defines the following methods for performing health actions on the Toolbox:

void setHealth(String category, IHealthStatus health) —set the health status for this Component

IHealthStatus getHealth(String category) – get the health status for the given category of this Component.

IHealthStatus[] getHealths() – get all the health statuses for this Component

IHealthStatus getWorstHealth() – get the worst health of this Component

3.4.4. C++ support

The C++ support for health is very similar to the Java support. The differences have to do mostly with C++'s use of smart pointers (std::tr1::shared_ptr) and passing strings as "const string&". This document just refers to strings as 'string' in order to be more succinct.

3.4.4.1. The Health.set() method

Component developers should call the Health::set method whenever the components health has changed or might potentially have changed.

static void atst::cs::services::Health::set (string category, pIHealthStatus status)

See the javadoc for atst::cs::data::HealthStatus details on creating an IHealthStatus object. There are a number of factory constructors for the various health types.

The atst::cs::services::Health class provides the following static methods:

pIHealthStatus atst.cs.services.Health.get(string category) – get the health of the given category for this component.

void atst.cs.services.Health.set(string category, pIHealthStatus status) – sets the health of the given category for this Component

pIHealthStatus atst.cs.services.Health.getWorst() – get the worst health available for this component.

void atst.cs.services.Health.setGood(string category) – sets the health of the given category to good for this Component. This is a convenience method.

void atst.cs.services.Health.setIll(string category, string reason) – sets the health of the given category to ill and sets the reason for this Component. This is a convenience method.

3.4.4.2. void atst.cs.services.Health.setBad(string category, string reason) – sets the health of the given category to bad and sets the reason for this Component. This is a convenience method.Health service tools

There are several health service tools available, including:

atst::cs::services::health::LoggingHealthServiceTool—logs health changes

atst::cs::services::health.PostingHealthServiceTool—post health changes

3.4.4.3. ToolBox access to the health service tools

The Toolbox uses the atst::cs::services::health::IHealthServiceTool interface to access the health service tool. The method defined by this interface is:

void changeHealth(pIToolboxAdmin toolbox, string category, pIHealthStatus oldHealth, pIHealthStatus newHealth) – carry out any behavior based on a, potential, change of health.

3.4.4.4. Health service access to the ToolBox

The atst::cs::interfaces::IToolbox interface defines the following methods for performing health actions on the Toolbox:

void setHealth(string category, pIHealthStatus health) —set the health status for this Component

pIHealthStatus getHealth(string category) – get the health status for the given category of this Component.

vector<pIHealthStatus> getHealths() – get all the health statuses for this Component

pIHealthStatus getWorstHealth() – get the worst health of this Component

3.5. PROPERTY SERVICE

The property service maintains metadata about Attributes, using a persistent store. This metadata varies depending on the role of the Attribute in the system, but may include a number of key items. See the Users' Manual for more details.

The property service can operate in one of two basic modes:

- preload mode—tells the metadata for all Attributes associated with the Component are loaded from the property persistent store during component initialization
- on-demand mode—tells the metadata for a specific Attribute is loaded the first time it is needed

The specific mode for a Component depends upon the particular property service tool(s) loaded into that Component's toolbox.

3.5.1. Property service persistent store

The persistent store is implemented in a relational database. The major database table is designed as a compromise between efficient access and flexibility. The aim is to trade improved disk access against an increase in CPU use. The database itself is implemented with as little knowledge of ATST attribute specifics as possible. For example, almost all metadata is stored as string values regardless of the ATST type associated with a given attribute. (The only exceptions are the vector and permission flags which are stored as Booleans.)

Each attribute's metadata is represented by a database tuple (row), with separate columns for each metadata item. The columns are:

id—the identification of the application "owning" this attribute as a string value

name—the attribute name as a string value

type—the ATST type of the attribute as a string value

vector—a Boolean that is true if this attribute is a vector of type

permission—three Boolean flags; one each for get/set/submit

description—a string that briefly describes the attribute's purpose

value—a CSV string holding the saved value for this attribute (may be null).

defaults—a CSV string holding the default values for this attribute (may be null).

limits—a CSV string holding any limits on acceptable values (may be null)

deltas—a CSV string holding any change deltas for the value of this attribute. Changes less than the change deltas are not monitored. (may be null)

When metadata is loaded from the property database it is converted internally to an object implementing the atst.cs.services.property.IProperty interface. Limits, change deltas, and default values are all converted to the appropriate type at this time, so references to these metadata items are efficient and require no further conversion. Each ATST attribute type must be one of:

- "string" your basic character array
- "integer" your basic integer number
- "real" your basic floating point number
- "boolean" your basic boolean

3.5.2. Pragma Properties

In addition to 'real' properties a property table for a component may also have some properties which dictate the behavior in a more broad way. At the moment there are two such properties.

The first is the "&inherits" property. This is used to allow one property table to inherit properties from another. This is useful in allowing a group of controllers that have common properties (e.g. thread model) to all inherit a common set or properties instead of having to define them multiple times.

The inheritance logic is essentially:

```
PropertyTable table = buildTable(appName);
if (table.has("&inherits")) {
    for (String s: table.get("&inherits)) {
        table.merge(buildTable(s), overwrite = false);
    }
}
```

The merged properties have qualifying-prefix portion of their names changed to match the name of the application. For example a controller `test.ctrl` inheriting a table named `single-threaded-controller`, with the property `single-threaded-controller.threadModel` would see change the name of the property to `test.ctrl.threadModel`. This ONLY effects the name of the property and does not affect the saved values, default values, limits, or any other property field.

The second pragma property is "&missing". It is used to define the behavior that a component or controller should exhibit when an attribute without a corresponding property is passed in. There are three acceptable values for missing:

- pass – allow attributes without corresponding attributes to be passed though the technical architecture on to the functional architecture. This is the default behavior.
- warn – log a warning that an attribute without a corresponding property was received. After logging the warning the attribute will be passed through the technical architecture to the functional architecture. This can be extremely useful in development and testing because it allows people to see what properties are not yet defined.
- fail – remove an attribute from a get/set attribute table, or reject an entire configuration when an attribute without a corresponding property is received. This should only be applied to controllers who understand all attribute that they receive. If a controller ever 'blindly' forwards attributes to other controllers, it should NOT use this.

3.5.3. Properties and Running Components

Depending on the property service tool(s) being used, a component's properties will either be read in on first access, or at initialization time. In both cases, the properties are then cached in the component and subsequent 'look-ups' will not result in more DB traffic. This is generally useful, but it does mean that it is awkward if you need to change the properties of a running component. This is generally seen as an engineering task, and as such should not need to happen frequently. Most components provide a back door for querying/setting and reloading properties at runtime. If you think of an attribute as a piece of data, then the property for that attribute is metadata.

It is possible to query this metadata by perform a get() on the component with an attribute of the form `foo?metadata`. For example, given an attribute named "pos", performing a get() with an attribute: `pos?type` would return the ATST data type of the "pos" attribute. All of the metadata fields of a component can be queried. Supported metadata fields are: "type", "vector", "readable", "writeable", "executable", "description", "values", "defaults", "limits", "deltas", and "mask". In addition to querying metadata on a per attribute/property basis, it is possible to get one field for all properties by performing a get with an attribute of the form `*?metadata`.

Additionally it is possible to set *some* of the metadata by performing a set() on the component with an attribute of the form `foo?metadata`. For example, given an attribute named `pos?readable`, with a value of false, the pos attribute could be made non-readable. It is important to note that you cannot set all of the metadata and the metadata that you can set cannot lead to more access only less. For example you can make a readable attribute non-readable, but you cannot make a non-readable attribute readable. Additionally you can decrease the range but you cannot increase it.

Finally it is possible to reload an attribute from the database, using a metadata-like set. This is useful if you update the properties in the database and need the controller to refresh them. You can perform a set with an attribute `foo?reload`, to reload the property foo, or `*?reload` to reload all properties. Note that in addition to going out to the DB and fetching the new entry for one or more properties the Cache entry for the reloaded property will also be cleared.

Please not that while simple names were used in the preceding examples, as always when passing attributes around CSF fully qualified attribute names must be used. This `pos?type` should really be `some.controller.pos?type`.

3.5.4. Java support

3.5.4.1. Types

The four ATST property types are mapped to Java objects as:

"string" → java.lang.String object

"integer" → java.lang.Long object

"real" → java.lang.Double object

"boolean" → java.lang.Boolean object

3.5.4.2. Property service tools

The primary role of property service tool is to map properties between the internal representation used by the access helper and the persistent store representation. There are two principal service tools:

- PreloadingPropertyServiceTool—preloads a memory resident cache of all of a Component's properties on the first access of any property. This allows subsequent accesses to be performed optimally.
- OnDemandPropertyServiceTool—loads each property separately when it is first accessed. This is generally only useful for Components that expect to only reference a small subset of their total properties on any given run

In addition, there is a utility service tool that may be chained by either principal service tools:

- LoggingPropertyServiceTool—detects changes made to properties and logs that change. It does not do anything more and so must be wrapped by some other property service tool.

Property service tools are typically shared by all components within a container.

3.5.4.3. Toolbox access to the property service tools

The Toolbox accesses the property service tool using the atst.cs.services.property.IPropertyServiceTool see the java doc for details on the method present in property service tools.

3.5.4.4. Property service helper access to the Toolbox

The property service access helper uses the toolbox to provide access to the service tool(s). See the javadoc for IToolbox (paying attention to methods that return, or take as an argument an IProperty) for details about those methods.

3.5.4.5. Component access to the property service

See the Users' Manual for details on how Components use the atst.cs.services.Property class to access the property service.

3.5.4.6. Adding new ATST types to the property service

New attribute types are easy to add, but require the approval of the ATST central office before they are accepted. The changes needed to add a new type to ATST are:

1. Pick a short string to denote the type that does not conflict with any existing type. An example might be "xtype".
2. Decide how to represent objects of this type in Java. For example, you may introduce a new class XType for this representation. Include an appropriate toString method.

3. In the source directory atst.cs.services.property.types you will need to add a new property type. (See the existing types as examples). You will need to add methods to convert from a string to the underlying object, and perform range checking.
4. In the PropertyFactory class of the same directory, you will need to add a factory constructor for your new type.
5. In the source code file for class atst.cs.services.Property you might convenience methods for accessing properties of this type.
6. Add the new type to this documentation.

3.5.5. C++ support

C++ support is very similar to the Java. Specifically the same service tools are present, and the doxygen is really your best place for details. The only major difference is that while Java allows null to be returned as the value of a Boolean, C++ does not. This means that whenever you are accessing non-string properties there is the potential for an exception to be generated when attempting to parse the string. For example attempting to parse ‘abc’ into an integer will throw a TraceableException.

4. MINOR SERVICES (TOOLS)

The minor services, also known as tools (not to be confused with the service tools used to implement service access), are services that Component developers may find useful in specific circumstances. Some are used within the ATSTCS and simply exposed for outside use.

4.1. ARCHIVE SERVICE

The archive service provides high-performance archiving of Attributes. All archived Attributes are archived with a timestamp and the name of the source Component that initiated the archive. The intent is to provide a means of saving bursts of high-speed engineering data for later analysis. See the Users' Manual for examples on how the archive service might be used.

4.1.1. Archive service dependencies

The archive service is implemented directly on top of the database support and uses the Log service. It must be loaded after the log service by the Toolbox Loader.

4.1.2. The archive database

The archive persistent store is currently implemented as a single active table in a single PostgreSQL database but could be easily distributed across multiple databases and hosts if more performance is needed during operations (this is not anticipated). There is one tuple stored in the active table for each archived Attribute.

Archived data is expected to be moved out of the active table into inactive tables periodically, and possibly removed to permanent offline storage infrequently. There is no requirement that archived data be kept readily accessible for any given period of time, so all of decisions on the movement of archive data through the persistent store are policy decisions to be made during operations.

There is currently no support beyond the facilities provided by PostgreSQL for querying the database. The ATST OCS will develop such facilities as their requirements are uncovered by the operational model for ATST.

The archive database is optimized for insertion, not for queries, on the assumption that insertions occur far more frequently than queries. Performance is enhanced when archiving Attributes with short, simple, values.

Archive message tuples contain the fields:

time_stamp—the time at which the Attribute was archived

source—the Component doing the archiving

name—the attribute name

value—the attribute value

The attribute value is recorded as a string. Timestamps are recorded to the millisecond level.

4.1.3. Archive service tools

The archive service tools that are available vary from one implementation language to another and are covered in the appropriate language support sections below.

4.1.4. Java support

4.1.4.1. Archive service tools

There are three archive service tools that are available, all three of these tools of extend atst.cs.services.archive.AbstractArchiveServiceTool:

atst.cs.services.archive.PrintArchiveServiceTool—prints a message to standard output on each archived Attribute. This service tool is usually chained with one of the other archive service tools.

atst.cs.services.archive.ArchiveServiceTool—this tool archives Attributes to the archive database individually as they are produced. This is a slow, but reliable, means of archiving data. Its use is not encouraged and although usable as both a private or a shared service tool, it should always be used as a private service tool.

atst.cs.services.archive.BufferedArchiveServiceTool—this tool buffers up Attributes until either (a) a fixed amount of time has passed (default is 0.5 seconds) or (b) a set number of archived Attributes have been collected (default is 5000 Attributes). It is highly optimized for performance and performs double buffering. It is slightly less reliable than the ArchiveServiceTool because a Component crash may result in the last 0.5 seconds of archived Attributes to be lost. However, since this is (one hopes!) a rare occurrence and the performance is so much better (factor of 10 or more) than ArchiveServiceTool, this is the preferred service tool. The BufferedArchiveServiceTool is usually loaded as a shared service tool but can also operate as a private service tool.

Both the ArchiveServiceTool and the BufferedArchiveServiceTool use a pool of database connections to improve performance. Also, while it is possible to chain those two service tools together, it would always be a bad idea to do so. The PrintArchiveServiceTool may be chained with either one.

4.1.4.2. Toolbox access to the archive service tools

The Toolbox access the archive service tool using the atst.cs.services.archive.IArchiveServiceTool interface. There is one method of interest in this interface:

void record(IToolBoxAdmin tb, String timestamp, String source, String name, String value) —record an Attribute.

Note that the Attribute's name/value pair has been split into separate parameters and the value field has been converted into a simple String. The Toolbox adds the timestamp and source parameters and is responsible for splitting the Attribute fields.

4.1.4.3. Archive service helper access to the Toolbox

The atst.cs.interfaces.IToolbox interface provides a single method for accessing the archive service:

void archive(atst.cs.interfaces.IAttribute attribute) —record an Attribute.

4.1.4.4. Component access to the archive service

See the Users' Manual for details on how Components use the atst.cs.services.Archive class to access the archive service.

4.1.5. C++ support

Same as java.

4.2. CONSTANT SERVICE

The constant service is implemented on top of the property service to provide access to manifest constants (values that are immutable and consistent across all ATST applications).

4.2.1. Java support

4.2.1.1. Constant service persistent store

The constant service persistent store is implemented as a separate table within the property service database. The following fields are found in that table:

name—name of the constant

value—value of the constant as a string

description—a short description of the purpose of the constant

Access to this table from Java code is provided by the atst.cs.services.property.PropertyDBServer class.

4.2.1.2. Constant service tool

There is no separate constant service tool. Instead a method for accessing a constant is part of the atst.cs.services.property.IPropertyServiceTool interface and is implemented by all property service tools (see below).

4.2.1.3. Toolbox access to the constant service tool

The atst.cs.services.property.IPropertyServiceTool interface provides a method for accessing constants:

IConstant getConstant(IToolBoxAdmin tb, String constantName)—return the value of the constant

4.2.1.4. Constant service access helper to the Toolbox

The constant service access helper accesses a constant through the following method declared in the atst.cs.interfaces.IToolBox interface:

IConstant getConstant(String constantName)—return the value of the constant

4.2.1.5. Component access to the constant service

See the Users' Manual for details on how Components use the atst.cs.services.Constant class to access the property service.

4.2.2. C++ support

Same as Java.

4.3. USER INTERFACES SUPPORT

One of the roles of the ATSTCS is to provide support for user interfaces. While not responsible for the user interfaces themselves, this support allows for consistent "look-and-feel" across user interfaces and provides user interfaces with a ready-to-go set of development tools.

This support is JES. See the separate JES user's manual for details on JES. See atst.cs.util.jes for the source.

5. COMPONENTS AND CONTROLLERS

ATST software applications are built on top of the Component and Controller base classes provided by the Common Services. This chapter discusses the implementation of these base classes and presents the ATST Test Harness that can be used by developers to quickly test Components and Controller subclasses within the ATST software environment. One of the drawbacks to the extension model used by components and controllers is also its biggest advantage; namely polymorphism. In many cases there is behavior that subclasses of controller must NOT override. In order to accomplish this there are almost always a `doFoo()` method for method foo defined in the public interface. The `doFoo()` method is intended to be overridden by subclasses, and allows function behavior to be easily added. The `foo()` method should NOT be overridden, and is where the technical behavior lives.

5.1. JAVA COMPONENTS & CONTROLLERS

In Java the public foo() methods are generally made final to strictly enforce the fact that they are not supposed to be overridden. The `doFoo()` method could be made abstract, but because some components and controllers have no behavior that they carried out, there is normally a sensible method stub, that carries out a no-op.

5.1.1. JNI within Java Components & Controllers

Java provides a framework for running C, C++ and assembly code from within a Java Virtual Machine, JNI. Using JNI is possible from within Components but there are some added complexities because of the way CSF uses custom class loaders, as well as different class loaders for different components. If a JNI library is loaded with a `System.loadLibrary()` call from within a component, when that component goes away all access to the library will be lost. Additionally, even while that component is around other Components within the same container will not have access to the JNI Library. Java prevents loading the same library twice, and CSF is not capable of forcing the JVM class loader to load the library instead of the component's class loader. In order to be able to use a library in more than one component, or after a component has been reloaded you must use the `Misc.getSharedObject()` method to get access to the object that loads the library. This method will force the JNI library to be loaded by the JVM/Container's class loader. The next problem with this approach is that classes loaded by different class loaders are not the same. This means that `atst.Foo` loaded by class loader one is not the same as `atst.Foo` loaded by class loader two. Trying to reference the object will result in `ClassCastExceptions` with the especially frustrating and awkward message "Foo is not of class Foo". The solution to this problem is to use interfaces. The CSF custom class loader always delegates the loading of interfaces to the JVM class loader. This delegation allows interfaces to be shared across namespaces. While this solves one problem it causes another. If an interface refers to a class, either as an argument, or a return type, then that class will also be loaded by the JVM's class loader. When attempts are made to reference the class from within the component's namespace Java throws a `java.lang.LinkageError`. To avoid this error an interface should only refer to Java native types, primitives, and other interfaces.

So what does all of this mean? Follow these few simple rules and you should be fine:

1) `System.loadLibrary()` should **only** be called from objects loaded by the JVM/Containers class loader. You can get one of these objects with `Misc.getSharedObject()`.
2) You should cast the object obtained by `Misc.getSharedObject()` to an interface, not a class.
3) Any time an object is being passed between namespaces an interface must be used to reference that object.

4) Interfaces should only reference Java native types, primitives, and other interfaces. The interface you write for your shared object must not reference other CSF classes or classes that you have written. If class objects must be passed as arguments or returned from method calls they must be passed using their corresponding interface.

5.2. C++ Components & Controllers

There is no C++ equivalent to the Java 'final' construct (at least not until we adopt C++11). The closest thing is the 'virtual' keyword. However because components and controllers implement/extend a pure virtual class/interface, it is not even possible to omit 'virtual' to make inadvertent overriding harder. For this reason, there is a danger that someone will override a controller method that they were not supposed. Only code inspection will prevent against this. For the same reasons as `doFoo()` methods were not made abstract in Java, they were not made pure-virtual in C++.

C++ components are not able to leverage the same custom classloader approach to isolate objects at runtime. The approach taken on the C++ side to allow static access helper methods to resolve to different toolboxes (allowing proper log/event sources for example), was to keep a map of toolboxes and component names, and use thread-local storage to store the key. CSF components all set the key as one of their first actions within any public method (e.g. submit()). This setting is done via the Namespace class.

5.3. Technical Architecture Blocks (TABs)

TABs are used to add technical code to a component or controller. A large portion of the code in many of a component's methods is bookkeeping and error checking that needs to be done regardless of what else is going on. TABs are standalone objects that have some close relation to a component or controller and are constructed in the component/controller's namespace. The PropertyTAB for example is a TAB that allows get/set calls to a component or controller to query or set property metadata for a component. Every time that a component's get method is called the component will iterate through a list of all TABs that implement get() and call each of their get methods. In the case of the PropertyTAB, its get method will look for queries of property metadata and fill in the appropriate values in the attribute table. There are three types of TABs: IParamTAB (has get and set methods), ILifecycleTAB (has init, startup, shutdown, uninit methods) and IDebugTAB (has setDebugLevel methods). There are a few advantages to using TABs instead of just using the associated do*Xxx* method. The same TAB can be used by both a component and controller. This means that the only duplicated code is registering the TAB with the super class. Since TABs are standalone and are iterated through, there is no fear of overriding the method from a superclass and forgetting to call the superclass's method. Since TABs are standalone objects they can be mixed and matched as needed and there are no issues with multiple inheritance. Since each TAB only has to do one thing it makes it easier to track down bugs, and modify behavior.

5.4. Testing Components and Controllers

The ATST Test Harness is available for testing Components and Controllers (collectively referred to here simply as components unless there is a need to distinguish them). The Test Harness supports the testing of components within an ATST execution environment. The intent is to provide a significant portion of a legitimate test environment for developers by implementing the technical architecture aspects of a general purpose test environment. The Test Harness includes a test manager responsible for controlling the tests and a user interface that provides software developers with access and control over the test environment and any components being tested. An event listener can monitor and display events that are posted during a test run. Support facilities include the ability to configure, save, and restore AttributeTables and Configurations that can be passed to test components. The Test Harness supports both interactive and batch operation.

Writing test cases for a Container/Component Model-based system is difficult without an appropriate test environment. Because the container is responsible for both managing the lifecycle characteristics of components and controllers as well as providing the essential services to those components and controllers, the Test Harness environment embeds both the test objects and the test manager within one or more containers. To ensure that the functional behavior of the test objects is available through the standard ATST functional interfaces, the test environment also restricts the test manager's access to the test objects to just those interfaces. See the CSF tools guide for details about the Testharness.

For more sophisticated tests, developers may customize the test harness through subclassing and either interact with it more programmatically, or create a more powerful text of graphical user interface.

5.4.1. Preparing for use

As with all ATST applications, you should set up your computers to allow password-less ssh connections between them, including a password-less ssh connection from any machine to itself.

5.4.2. Using the Test Harness command-line interface

A simple command-oriented interface is provided as part of the test harness. This interface can be used to load one or more test objects into one or more containers, drive those components through their lifecycles, and issue ATST commands to those components. For a details about the Test Harness see TN-0120 (CSF Tools).

6. ADMINISTRATION

6.1. INTRODUCTION

This section describes the basic administrative operations needed to install and operate the ATST Common Services Framework.

6.2. INSTALLING, BUILDING, AND STARTING

This section covers the process of bringing the ATST Common Services Framework online at your site. It is strongly suggested that you read all parts of this section before performing any of the steps outlined here. While there is a preferred Linux distribution for CSF, this guide is trying to provide information generic enough for installing CSF on ANY modern Linux distribution.

6.2.1. Installation

There are two types of CSF installations: client-side, and server-side. This guide covers both, but was written with the more common client-side installation in mind. For this reason some of the server-side instructions are found out of order in the sections below. While awkward, the assumption is that client installations are more common, (one server can serve many clients, and one machine could have many different version of the client software installed in parallel), and because repeating the server install steps, could cause outages for other people using the server.

Preparing your System

You need:

1. **Linux** with all the standard development tools, including gnu-make, gcc/g++. The preferred distribution is 64-bit CentOS 6 (a community build that is equivalent to Red Hat Enterprise 6) and all source code in this release has been built on 64-bit CentOS 6. However, with some work it should be possible to build on other Linux distributions (32 or 64 bit). The ATST Common Services Framework has been built and operated successfully under Fedora Core 14, 15, 16, 17, Debian, Ubuntu 10.04 11.10, 12.04, Linux Mint 13, 14, Scientific Linux 5, 6, CentOS 4, 5 and Java only on Darwin, in the past. No significant problems are anticipated building and using ATST CSF with other modern Linux distributions. This guide is written with CentOS in mind but it tries to be as generic as possible so that users of other distributions may obtain the information that they need.
2. **zsh** is used by the development support scripts. The version supplied with CentoOS 6 is sufficient but might not be installed by default. It is contained in the package "zsh".
3. **ICE** (version 3.5.1, currently) must be installed. On CentOS, you can install ICE via yum. See section 6.8.2 below for details.
4. **Oracle Java** Development Kit 1.7 installed. The latest version of the Oracle JDK should be obtained from the Oracle website. By convention the tar-ball should be downloaded and installed in /opt/java7.
5. **gcc & gcc-c++** version 4.8.1 or later should be used. Since CentOS 6.4 doesn't make this version of gcc/g++ available via its standard package manager, it must be installed as a third-party package. See the Third party sorfware section of this document for detailed installation instructions.
6. **Doxygen** (version 1.3.9 or later) is used to create the API documents. The default version supplied with CentOS 6 is sufficient but might not be installed by default. It is contained in the package "doxygen"

7. **ICU** is used by C++ CSF as part of its date library. The versions supplied with CentOS 6 are sufficient but might not be installed by default. They are contained in the packages “libicu” and “libicu-devel”. NOTE: You need the headers to build so you MUST install “libicu-devel”!
8. **Postgresql** is the database management system used by CSF. Detailed installation instructions are found in the Third party software section of this document.
9. Libzdb is a C++ database connection pool manager. It efficiently and effectively handles connection to our Postgresql databases. Detailed installation instructions are found in the Third party software section of this document.

Under CentOS 6 you can install most the packages included install all the packages included in the official CentOS repository (i.e. everything except ice, postgresql, and Java) by doing a:

```
yum install gcc gcc-c++ cvs zsh doxygen libicu-devel openssh openssh-clients
```

Most of these are straightforward, but the server-side of PostgreSQL requires careful configuration. Proper configuration of PostgreSQL is required, and efficient operation of the database depends upon this configuration. In particular, note that the default configuration is not appropriate. The notes in the 3rd party software section below provides some useful information about configuring PostgreSQL, but you may also want to take advantage of the excellent on-line documentation at the PostgreSQL website.

Installing via CVS

Before an ATST Software Development Tree (ASDT) can be used, it must first be created using the tools provided in the admin directory. This is a bit of a chicken-and-egg problem, since admin is part of the ASDT tree. The simplest solution is to check out the entire ATST distribution before running the tools in the admin directory. Assuming you want the ASDT to be rooted at (say) /opt/atst, based upon the current Canary_5 release, and you have ${CVSROOT} set to “:pserver:${USER}@maunder.tuc.noao.edu:/home/atst/src/cs“:

```
-> cd /opt
-> cvs login
-> cvs co –P –r Canary_5 atst
-> export ATST=/opt/atst
```

The cvs supports checking out to a non-standard directory using the “-d dirName” option. This option is NOT recommended for an ASDT because of the way subsystems are layered on top of an ASDT. This is not to say that you cannot use the –d option, it will just make things harder.

A Quick and dirty approach to install/configuring

The following commands build an ASDT (provided you have configured cvs to access the repository) and then uses that the createDevel command from the admin directory to finish the creation:

```
-> cvs login
-> cvs co –P –r Canary_5 atst
-> cd atst
-> export ATST=`pwd`
-> ./admin/createDevel --make-all
```

Here, the base directory atst is created by the CVS checkout command (co). Once the environment variable ATST has been set to the base directory of this ASDT, it is available for development. The createDevel script, when run as the above example, prompts the user for all required site-specific information. A complete listing of those parameters, and their respective values as entered by the user, is created by the createDevel script in the file ./site.config.out. The **Site Config Parameters** section (6.7) below gives information on the key parameters.

A better approach to install/configure

The above quick-and-dirty method works by prompting the installer for site-specific information during the execution of admin/createDevel. It is better to set up the site-specific information before running createDevel. In this case, the installation steps would be:

```
-> cvs co –P –r Canary_5 atst
-> cd atst
-> export ATST=$(pwd)
-> cp admin/site.config.template admin/site.config
-> vi admin/site.config        # set site-specific variable definitions
-> ./admin/createDevel --make-all
```

This approach leaves the site-specific parameter definitions in the file admin/site.config and allows createDevel to run without prompting the installer for additional information. Note that the chosen release may be specified in admin/site.config. Configuration parameters that are left unspecified in admin/site.config will be asked for as createDevel runs. The site.config template contains comments about many of the parameters, but for more details see section 6.7 (Site Config Parameters) below.

Installing an ASDT from tar or zip file

ATST software developers can download a tarball or zipfile of the ATST software from the ATST software TWiki at: http://maunder.tuc.noao.edu/AtstSoftwareDevelopment using the CVS web access found at the ATST Common Services Framework topic page.

If you have downloaded a tarball of the entire atst source tree as atst.tar.gz, you can create an ASDT using the following steps:

```
-> tar -zxopf csf.tgz
-> cd atst
-> export ATST=`pwd`
-> cp admin/site.config.template admin/site.config
-> vi admin/site.config
-> ./admin/createDevel --make-all
```

Note that, after the first step, this is identical to the previous process.

Installing an older version of the ASDT

There are times when it is useful to work on an older release of the ATST software (for example, when looking into bug reported by vendors using that older version). Official ATST software releases are tagged. You can build an ASDT for software release TAG by using (for example) the following command sequence:

```
-> mkdir oldAtst
-> cd oldAtst
```

```
-> cvs co -P -r TAG atst
-> cd atst
-> export ATST=`pwd`
-> cp admin/site.config.template admin/site.config
-> vi admin/site.config
-> ./admin/createDevel --make-all
```

6.2.2. Building

Once the ASDT has been created and configured, you can build the software with:

```
cd $ATST
make build_all
```

You may build the Java javadoc and the C++ doxygen using the commands:

```
cd $ATST
make docs
```

If you are building a new release on top of an existing release, you should also clean out existing class files:

```
cd $ATST
make ice_clean class_clean gcc_clean
make build_all
```

Running make with no arguments displays a list of possible make targets, including the following:

- ice—use the slice2java and slice2c++ to turn the slice definitions into C++ and Java source files.
- classes—produce Java class files for all Java source files in the current directory (and any referenced classes from Java source files in $ATST/lib/Java/generated_java). The class files are placed into subdirectories of $ATST/lib/Java/Classes
- class_clean—remove all java class files.
- jni—produce jni shared libraries. This uses the information from site.config to determine which jni libraries to build.
- jni_clean—remove all jni shared libraries, and delete the associated object files
- gcc_all—produce all C++ shared libraries, and executables.
- gcc_clean—remove all C++ objects.
- docs—generate and install source code documentation into subdirectories of $ATST/doc/api-docs
- install_scripts—any scripts found in the current directory are installed into subdirectories of $ATST/bin as described earlier
- build_all—perform ice, classes, jni and gcc_all
- install_properties—crawl the tree looking for XXX.prop files and attempt to read them into the property database with PropertyReader
- extract_properties—crawl the tree looking for XXX.prop files and attempt to identify which component(s) are included in the file. Then replace the files with the output from PropertyWriter.

6.2.3. SSH

You will also need to be able to use SSH to move among the various machines involved without a password. Many of the scripts rely upon SSH internally and you will be asked for the password a truly amazing number of times if you have not set this up! Even more important than the number of times that you are asked, is the number of times that things will not run properly because SSH has not been configured. If you are not familiar with setting up password-less SSH see the instructions below (section 7.5.7) below.

6.2.4. Starting the base ATST services running

The ATST Common Services Framework needs to have both PostgreSQL and the ICE servers running, but only one instance needs to be running for a large deployment. If you already have a CSF server up and running you do not need to worry about setting up the databases or ice. If you run the services locally, or are setting up a server, you will need to configure and start them.

Building the ATST databases and tables

See the subsections of 6.8.1 on setting up PostgreSQL to be the database server IF the machines being set up will be used to host the PostgreSQL server.

ICE

See the subsection of 6.8.2 on Starting Ice Services if the development tree being set up will be used to host the ice server.

6.3. TESTS

The last step of building ATST Common Services Framework also tests a number of key features of using ATST. Among things, it tests the basic services, the connection services, and the Container/Component model operation.

There are some other simple tests that can be performed to verify that the ATST Common Services Framework are ready for use. If you can run all of the tests outlined below your installation should be up and running properly.

6.3.1. Testing the basic services

The basic services (other than the Connection and Event services) can be tested with:

```
$ATST/bin/ajava atst.cs.ccm.component.demos.ServiceDemo
```

The output includes several stack traces. This is deliberate to check the ability to include such traces in log messages.

6.3.2. Testing the Connection and Event services

The following can be used to test the Connection and Event services. The publisher and subscribers may be started on the same machine or on separate machines. The example shown here assumes two machines: machineA and machineB.

On machineA, start a subscriber running:

```
-> $ATST/bin/ajava atst.cs.services.event.TransientEventSubscriber
```

On machineB, start the publisher:

```
-> $ATST/bin/ajava atst.cs.services.event.TransientEventPoster --mark
```

The publisher publishes events as fast as it can and the subscriber notes their receipt. After the test completes you should see output indicating the throughput of the event service.

6.3.3. Basic Testharness test

The following interactive test uses the Testharness to run though some basic commands and make sure that things are working as expected. User commands are shows in bold, information returned by the testharness is shown normally.

```
$ $ATST/bin/Testharness
[snip] notes and a big long help message
atst> load test.ctrl atst.cs.controller.Controller
atst> init test.ctrl
performing init
test.ctrl initizlized
atst> start test.ctrl
performing startup
test.ctrl started
atst> new table tbl1
Table tbl1 added
atst> new table tbl2
Table tbl2 added
atst> new config cfg
Configuration cfg added
atst> add tbl1 test.ctrl.foo a
inserting test.ctrl.foo: a
atst> add tbl2 test.ctrl.foo “” # that last is a pair of double quotes
inserting test.ctrl.foo
atst> set test.ctrl tbl1
atst> get test.ctrl tbl2
 test.ctrl.foo:
                'a'
atst> submit test.ctrl cfg
Configuration excepted
event: test.ctrl.configstate
        __.configid: cfg.31272.22
        __.source: test.ctrl
        __.status: scheduled
        __.timestamp: 2012-06-12 18:02:01.800

event: test.ctrl.configstate
        __.configid: cfg.31272.22
        __.source: test.ctrl
        __.status: running
        __.timestamp: 2012-06-12 18:02:01.811

event: test.ctrl.configstate
        __.configid: cfg.31272.22
        __.errNo: -11
        __.reason: no action support in this controller
        __.source: test.ctrl
        __.status: aborted
        __.timestamp: 2012-06-12 18:02:01.815
atst> shutdown test.ctrl
performing shutdown
test.ctrl shut down
atst> uninit test.ctrl
performing uninit
test.ctrl uninitialized
atst> quit
```

The above commands load a controller named "test.ctrl", whose class is "atst.cs.controller.Controller" into the Testharnesses default container. If you changed the class to say "my.package.MyController" you would load your class. After the controller has been loaded, initialized, and started we create a few data objects to use (two attribute tables, and a configuration). The first attribute table is passed to the component, and CSF stores the value of 'foo' in component's Cache. The second attribute table is used to read the value back out. Finally a configuration is submitted to the test controller, and the action then fails. See atst.cs.controller.Controller.doAction() to understand why the action fails.

6.4. MAINTENANCE

ATST is not designed to operate as a maintenance-free facility. There are a number of tasks that need to be performed as part of the routine operation of the ATST Common Services Framework, such as monitoring of disk usage, CPU loads, database resources, etc.

Identifying and addressing ATST Common Services Framework maintenance issues are part of the process of defining observatory operations and are TBD at this time.

6.5. TROUBLESHOOTING AND PROBLEM REPORTING

6.5.1. Troubleshooting

ATST Common Services Framework is a moderately large software system that is both heavily threaded and distributed. Troubleshooting in such an environment can be problematic. Developers are encouraged to use the ATST Log Service (particularly the debug methods) heavily. The ability to generate stack traces at any point in the code can be particularly useful.

Simply displaying log messages, even with stack traces, is not sufficient, of course. Other tools and approaches to troubleshooting are needed. Such facilities are not yet part of the ATST Common Services Framework - they will be added over time and documented here as they become available.

6.5.2. Problem reporting

ATST maintains an online problem reporting and tracking system for use by ATST developers. Access is only available to registered ATST developers—you should visit the site http://maunder.tuc.noao.edu:8080 and request an account now to avoid delays when you do uncover a problem. You may also contact the csf-devel mailing list at csf-devel@noao.edu with questions or problems.

6.6. KNOWN PROBLEMS

6.6.1. SELINUX

Currently the CSF team has not spent the time necessary to get CSF to work on systems with SELinux enabled. If you are going to be using CSF, SELinux should be disabled. Suggestions are welcome for instructions on how to run CSF on machines with SELinux enabled.

6.6.2. Firewalls

Currently the CSF team has not spent the time necessary to get CSF to work on a system with a firewall enabled. If you are going to be using CSF, you should disable any and all firewalls on your CSF machines. Suggestions are welcome for instructions on how to run CSF on machines with firewalls enabled.

6.6.3. Known problems in the Java support

Check the release notes for the specific release version you have for more known problems.

The class loader used to separate each Component into its own namespace is currently pretty simple. In particular, it separates classes defined in the atst class hierarchy, but NOT classes from 3rd party tools or the java class hierarchy. However, interfaces are not separated (by design, this will not change) so all components and containers share the same interfaces. It is likely that a later revision of this class loader will separate more classes into component namespaces.

It is ridiculously easy to hang a Java container—simply start a Thread or a Timer as a non-daemon in a component and fail to shut it down properly when the component shuts down. This is a property of non-daemon Java threads—JVMs will not terminate if there are non-daemon threads still running. For this reason, all threads and timers inside of the ATST Common Services Framework are implemented as daemon threads. Component developers should carefully design threads and timers so that they can either exist as daemon threads or are properly shutdown.

6.6.4. Known problems in the C++ support

A segmentation fault thrown within one controller/component will bring down the entire container, and all controllers and components held in that container.

6.6.5. IP Addresses and Networking

ICE is particular about how it expects networks to be setup. Things work very well if all machines have static IP addresses, and all machines that are part of the distributed system have all other machines hostnames and IP addresses in the /etc/hosts file. This is not strictly necessary and if DNS resolves hostnames properly things will work properly. However things work very poorly with machines using typical DHCP configurations. Things get worse still with machines hiding behind NAT as is often the default for virtual machines. For this reason we do NOT recommend running virtual machines, unless they can have a bridged network connection, and their IP address can be resolved via DNS or /etc/hosts. This does not mean that it is impossible to run Virtual Machines, but it needs to be looked at on a case by case basis to determine how best to implement.

6.6.1. Threads

CSF uses lots of threads. Most of these threads are parked in efficient sleeps, and will not cause problems but they do eat up resources. Specifically CentOS 6, by default limits, a single non-root user to only 1024 threads. This might sound like a lot, but if you are multi-tasking, and testing some heavily loaded containers you can run up against this limit. Additionally if one of your containers does not properly shutdown, it becomes very easy to hit this limit. On the Java side this presents itself as an error of the form: "java.lang.OutOfMemoryError: unable to create new native thread". If you encounter this problem, or worry that you might, you should increase the thread limit. This can be done by editing the file "/etc/security/limits.d/90-nproc.conf". See the Redhat documentation for details. How high you should raise this number depends on the resources present for your machine. If you are running on a single core you probably should not raise it. If you are running on a 32 core machine you should be able to go pretty high without problems.

6.7. Site Config parameters

There are a number of parameters that need to be set when configuring an ASDT. These parameters may be defined in the $ATST/admin/site.config file. Any parameters that are not assigned values in that file are prompted for by the $ATST/admin/createDevel script as needed. This section gives a brief description of each of these parameters.

baseDir This is the full pathname of the root of the ASDT. Normally it should match the value of $ATST (e.g. /opt/atst).

ATST_RELEASE_FLAG The release of ATST that this ASDT is based upon (e.g. Panguitch-1P3). Be aware that if you leave this blank or set to 'trunk', updates will come from the trunk of the ATST CVS repository. The trunk code is not guaranteed to be in a runnable state and may actually conflict with your current release!

ATST_PACKAGE_LIST A list of the ATST packages that are also installed in this ASDT. For example, if you plan to use (and you *do* plan to!) the Base package, you should include "base" in this list. Set to "none" if there are no such packages. The contents of this list are used to determine which $ATST/tools to install.

ATST_RUN_ENVIRON ATST needs a spot to put various files created during execution (log files, etc.). This is the full path to that spot. It will be created if it doesn't exist. (e.g. /var/tmp/atst).

USE_JAVA Set to "yes" because you want to build the Java support. You want to build the java support because many of the support tools, including some used during initialization of the ASDT, are written in Java.

ATST_JAVA_HOME The full path to Java. ATST uses JDK1.7 currently (e.g. /opt/java7).

ATST_JNI_PACKAGES This list of packages tells the make system which JNI packages to build. If you write your own JNI library, be sure that the package name you use in Makefile.jni matches what you add here. If you do not have any JNI packages you may use a double quote “” to indicate no packages.

ATST_JLIB_DIRS This is a list of directories holding JNI libraries to be linked into Java application. Set to "none" if there are no such libraries needed. (Right now, this need only be set if the ATST TCS package is also installed in this ASDT.

ATST_JAVA_DOXYGEN Change from "no" to "yes" if you want to build the Java API documentation with doxygen instead of javadoc.

ATST_JAVAC_FLAGS Special flags for the Java compiler. The default settings should be fine.

USE_CPP Set to "yes" if you want to build the C++ support.

ATST_TARGET_ARCH This identifies the target architecture to build C/C++ code to. You shouldn't need to change this parameter from its default setting unless attempting to cross-compile.

ATST_CPP_CPP_BIN This should be set to the pathname for the correct version of g++ to use with ATST when USE_CPP is "yes". Right now this needs to point to the g++ version 4.4+ compiler.

ATST_CC_BIN This should be set to the pathname for the correct version of gcc to use with ATST when USE_CPP is "yes". Right now this needs to point to the gcc version 4.x.

ATST_CLIB_DIRS A list of directories holding extra C/C++ libraries that need to be linked with ATST. Set to "none" if there are none. When using PostgreSQL 9.3 on CentOS you will need to include the path the Postgres files, which will likely be /usr/pgsql-9.3/lib/.

ATST_CONCURRENT_CMAKES How many cores you want to use when building C++ code. A value of 0 indicates that the system should detect how many cores you have and use them all. This should work in most cases.

ATST_ARCHIVE_DB_HOST The machine running the archive database (e.g. maunder.tuc.noao.edu).

ATST_ID_DB_HOST The machine running the id database (e.g. maunder.tuc.noao.edu).

ATST_LOG_DB_HOST The machine running the log database (e.g. maunder.tuc.noao.edu).

ATST_PROPERTY_DB_HOST The machine running the property database (e.g. maunder.tuc.noao.edu).

ATST_HEADER_DB_HOST The machine running the header database (e.g. maunder.tuc.noao.edu). This should only be changed from the default value of "none" if the Base package is also installed in this ASDT.

ATST_EXPERIMENT_DB_HOST The machine running the experiment database (e.g. maunder.tuc.noao.edu). This should only be changed from the default value of "none" if the OCS package is also installed in this ASDT

USE_ICE This flag determines if ICE support will be built. Set to yes if you want ICE, no (or leave blank) otherwise.

ATST_ICE_HOME The base of the installed ICE support. If ICE has been installed as described below, this should be left as the default "/usr"

ATST_ICE_VERSION This flag is used to identify the correct Ice to use. It should track the version of Ice that is installed (e.g. 3.5.1)

ATST_ICE_LIB_SUFFIX This flag is used to identify the correct Ice shared libraries to use. It should track the version of Ice that is installed (e.g. 35 for Ice version 3.5.x)

ATST_ICE_JAR The pathnames of the ICE java jar files (e.g./usr/share/java/Ice.jar:/usr/share/java/IceStorm.jar:/usr/share/java/IceGrid.jar:/usr/share/java/Glacier2.jar)

ATST_DB_JAR The pathname of the DB4 jar file needed for Ice Java support (/usr/share/java/db-4.8.30.jar)

ATST_JAVA_ICE_LIB_PATH The pathname of the directory holding Java support for Ice (/usr/share/java)

ATST_ICE_CONNECTION_HOST The machine that is to run the Ice name service. (e.g. maunder.tuc.noao.edu). This must exactly match the value returned by hostname on that machine.

ATST_ICE_CONNECTION_PORT Port on that machine to be used by the Ice name service (e.g. 11000).

ATST_ICE_EVENTS_HOST The machine that is to run the Ice event service. (e.g. maunder.tuc.noao.edu). This must exactly match the value returned by hostname on that machine.

ATST_ICE_EVENTS_PORT Port on that machine to be used by the Ice name event service (e.g. 12000)

ATST_ICE_ICEBOX_PORT Port on that machine to be used by the Icebox container [which holds the event service] (e.g. 11998).

USE_NDDS Flag that determines if NDDS support will be added. You need the root password to configure the network if this is the case.

ATST_RTI_LICENSE_FILE That absolute path location of the RTI NDDS license file.

ATST_DDS_EVENTS Include NDDS event service support

ATST_DDS_EVENT_DOMAIN_ID Allows setting up separate test domains. This should be left as the default ‘100’

ATST_DDS_EVENT_NIC Which network controller to use for DDS Events. All will use all interfaces. eth0 would limit the dds event service to only multicasting on eth0.

ATST_DDS_BDT Include NDDS Bulk Data Transport support.

ATST_BDT_NETINTF The network interface that the BDT should use to transmit data.

ATST_QAS_NETINTF The network interface that the QAS should use to transmit data.

ATST_BDT_NETWORKS The network interfaces are used with BDT and QASThe list should be separated by spaces.

6.8. THIRD PARTY SOFTWARE

6.8.1. Notes on preparing PostgreSQL for ATST

This section gives some hints and suggestions for configuring PostgreSQL for use with ATST Common Services Framework. It's not intended to provide a full set of instructions for installing and configuring PostgreSQL, you will need to examine the PostgreSQL documentation for that level of help.

Installing PostgreSQL

ATST needs the C++ bindings for the PostgreSQL driver (libpqxx, libpqxx-devel), and those are not found in the official CentOS repository. Therefore you must install those packages from the PostgreSQL maintainers. You can optionally install the CentOS packaged version of the PostgreSQL server, or the PostgreSQL version. Both servers appear to work fine, and depending on which one you install, your config files will be in different places. If you are installing the PostgreSQL maintained version you should use version 9.3, as it is known to work, but problems are not expected with other versions.

A good source of instructions for installing PostgreSQL is the web site: http://yum.postgresql.org. This site guides you through the steps needed to install PostgreSQL using the RedHat/CentOS installation tool "yum". They even provide an rpm to enable their repository. At a minimum, you need the following RPM packages from either the CentOS or PostgreSQL repository:

- Postgresql93
- Postgresql93-libs
- Postgresql93-server

For C++ support, you also need the following packages from the PostgreSQL repository:

- postgresql93-devel
- libpqxx-devel

```
At the time of writing running the following command run as root will enable
     the PostgreSQL maintainer’s repository and install the necessary
     packages:rpm -Uvh http://yum.postgresql.org/9.3/redhat/rhel-6-
     x86_64/pgdg-centos93-9.3-1.noarch.rpm && yum install postgresql93
     postgresql93-libs postgresql93-server postgresql93-devel libpqxx-devel
```

If you are installing the ASDT for development only (i.e. you have an additional machine that runs the services) you may skip ahead to section 6.8.2.

Otherwise as the last step, you should create an initial database installation:

Configure Postgresql

It is convenient if the PostgreSQL server starts running at boot, and has been initialized. Assuming that you are running the CentOS version of PostgreSQL Server, you may do this with the following commands run with super user privileges:

Initialize the postgresql database:

```
# service postgresql-9.3 initdb
```

Configure postgresql to start at boot:

```
# chkconfig --level 345 postgresql-9.3 on
```

Start postgresql running the first time:

```
# service postgresql-9.3 start
```

If you are ever in doubt about whether or not postgres is running you may run:

```
# service postgresql-9.3 status
```

Initial configuration steps (PostgreSQL 8.4)

After you have followed the PostgreSQL installation instructions described above, and installed from the downloaded rpm, you will need to modify some files. All of the files that need to be modified are in PostgreSQL's data directory. Assuming that you are using CentOS's PostgreSQL server, the files should be in /var/lib/pgsql/9.3/data. *Be aware that the default settings are quite conservative. Performance can be greatly enhanced through proper reconfiguration*.

- postgresql.conf - this file contains most of the major configuration options for PostgreSQL. Postgresql needs to be configured to listen to all Ethernet adapters which might have incoming connections. If you change the listen address (listen_addresses) to '*' it will listen on all Ethernet ports. There are other configurations which will work but this option should work in all cases.
- pg_hba.conf - this file identifies the nature of connections to the PostgreSQL back end. You'll need to make sure that all machines that you expect to have ATST software running on have permission to use PostgreSQL. You should also allow database users other than the database owner access to databases. For example, the Tucson based ATST software group's computers are all assigned addresses in the 140.252.[0~255].[0~255] range. They trust connections coming from that entire range. PostgreSQL additionally trusts all connections coming from the localhost (127.0.0.1). You should also trust connections over IPv6 unless you are sure your network does not use it. Consider the following example entries that should likely be included in your pg_hba.conf file:

```
local   all     all                         trust
host    all     all     127.0.0.1/32        trust
host    all     all     140.252.0.0/16      trust
```

host all all ::1/128 trustThese are very loose settings: a PostgreSQL expert could tighten these up. After making these changes, you will need to restart PostgreSQL. Under CentOS Linux, this can be done with (as root):

```
/sbin/service postgresql restart
```

Create DB User

As the Linux user postgres launch the postgresql tool "psql" and create a user atst.

postgres=# CREATE USER atst WITH CREATEDB;

postgres=#\q

Initialize the Databases

After you have gotten postgresql up and running, you need to create the databases/tables that CSF needs. First make sure that you have run though the Postgresql configurations mentioned above. Second make sure that you have successfully built CSF. Third, make sure that the Ice services are running. Finally you should run the commands

cd $ATST

./admin/createDevel --init-db

6.8.2. Installing ICE

The easiest approach to install is ICE using the RedHat/CentOS installation tool "yum". A yum repository is available. Visit the web site:

http://www.zeroc.com

and follow the instructions in the section "Download" → "Linux RPMs". In particular, you should look at the subsection:

Red Hat Enterprise Linux 6 Yum Repository

The ATST software team currently supports Ice version 3.5. Early or later versions may work but have not been fully tested. Please use either the upgrade or clean install instructions provided below. Note that the upgrade instructions below may fail depending on the state of your yum dependency tree. If that happens, a clean installation of CentOS is recommended. A few packages, such as redhat-lsb-core and perl-devel are know to cause issues with the upgrade path. If you have these libraries installed, you will need to uninstall them before proceeding with the upgrade.

To upgrade from Ice 3.4 to Ice 3.5 you will need to run these commands:

```
sudo mv /etc/yum.repos.d/zeroc-ice.repo /etc/yum.repos.d/zeroc-ice.repo.old

sudo wget -O /etc/yum.repos.d/zeroc-ice.repo
http://download.zeroc.com/Ice/3.5/el6/zeroc-ice-el6.repo

sudo yum erase ice ice-servers ice-c++-devel ice-java ice-java-devel ice-libs
ice-utils

sudo yum install ice-3.5.1-1.el6 ice-servers-3.5.1-1.el6 ice-libs.x86_64
0:3.5.1-1.el6 db53-devel db53-utils db53-java ice-c++-devel-3.5.1-1.el6 ice-
java-3.5.1-1.el6 ice-java-devel-3.5.1-1.el6
```

To install Ice 3.5 on a system not containing a previous version of Ice you will need to run these commands:

```
wget -O /etc/yum.repos.d/zeroc-ice.repo
http://download.zeroc.com/Ice/3.5/el6/zeroc-ice-el6.repo

yum install ice-3.5.1-1.el6 ice-servers-3.5.1-1.el6 db53-devel db53-utils
db53-java ice-c++-devel-3.5.1-1.el6 ice-java-3.5.1-1.el6 ice-java-devel-
3.5.1-1.el6
```

Starting Ice Services

If you are installing the ASDT for development only (i.e. you have an additional machine that runs the services) you are done setting up Ice. If not you should consider creating a script to automatically start ice every time the system boots. By convention this script would be called by /etc/rc.local. All that the script needs to do is set the ATST environment variable, and call $ATST/bin/startIceServices. If you are security conscious, you might consider calling using 'su' to run startIceServices as the owner of the development tree instead of the root user. If you are installing for the first time, you need to make sure that you call this script AFTER initially building your Software Development Tree.

6.8.3. Installing libzdb

Libzdb is a c++ database connection pool manager. Documentation and source code can be found at:

http://www.tildeslash.com/libzdb/

To install libzdb on your system do the following:

```
sudo yum install flex
Download libzdb-3.0.tar.gz from http://www.tildeslash.com/libzdb/
Expanded it in a temp directory with "tar -zxvf libzdb-3.0.tar.gz"
      You can delete this directory after the installation is complete.
cd libzdb-3.0
./configure  --without-sqlite  --without-mysql  --with-postgresql=/usr/pgsql-9.3/bin/pg_config --enable-protected --enable-optimized
      You may need to change the postgres path above (or you can leave this paremeter off and let it look in the default location).
make clean
make
sudo make install
      The following files will be created:
      /usr/local/lib/libzdb.a
      /usr/local/lib/libzdb.la
      /usr/local/lib/libzdb.so
      /usr/local/lib/libzdb.so.10
      /usr/local/lib/libzdb.so.10.0.0
      /usr/local/include/zdb/*.h
```

6.8.4. Installing gcc/g++ 4.8.1

CentOS 6.4 standard yum package for gcc/g++ is v4.4.7. CSF requires gcc/g++ 4.8.1 or higher. Until this version is available via standard CentOS package repos, we must install it from a third-party site. The following provides instructions on how to install gcc/g++ 4.8.1 on CentOS 6.4.

```
Add the repo:
      sudo yum-config-manager --add-repo http://puias.princeton.edu/data/puias/DevToolset/6/x86_64/
Add the key:
      sudo rpm --import http://puias.princeton.edu/data/puias/6/x86_64/os/RPM-GPG-KEY-puias
Install:
      sudo yum install devtoolset-2-gcc-4.8.1 devtoolset-2-gcc-c++-4.8.1 (takes a few minutes to run)
Verify:
      /opt/rh/devtoolset-2/root/usr/bin/gcc –version
      /opt/rh/devtoolset-2/root/usr/bin/g++ --version
Configure CSF:
```

```
vi $ATST/admin/site.config
      ATST_CPP_CPP_BIN= /opt/rh/devtoolset-2/root/usr/bin/g++
      ATST_CC_BIN= /opt/rh/devtoolset-2/root/usr/bin/gcc
$ATST/admin/createDevel --make-all
```

6.9. PACKAGE DEVEL

The Package Devel Framework is an attempt to bring createDevel-like functionality to additional packages layered into an ASDT. The primary goal is to simplify the installation of applications within an ASDT. This includes populating $ATST/bin with the appropriate shell scripts, loading the various databases with their initial values, and initializing the $ATST/data directory with the necessary files. Like the rest of CSF, package devel is a framework that is trying to reduce the amount of work developers have to do, and to create a more uniform look and feel. The goal is for developers to be able to create a short collection of ZSH functions that can be used by end-users to setup their system. First this document will describe how end-users are expected to run the package devel scripts.

6.9.1. Package Devel and End Users

The steps for package devel end users closely mirror those of the create devel script and site configuration options file. The basic process is to use a template to generate a configuration file, edit the configuration file, and then run devel script to finish configuring the system.

End users should be given a `$ATST/admin/<pkg>/<pkg>Site.config.template` file which can be either appended to `$ATST/admin/site.config`, or copied to `$ATST/admin/<pkg>/<pkg>Site.config`. End users would then update the various options (thing like container host machines, hardware connection addresses, flags to load connection simulators, etc.). After that end users would run `$ATST/admin/pkgDevel <pkg> --option1 –option2 …` to complete the deployment/configuration of the system.

As mentioned above, users have two options for where to place their options. If users want a single configuration file for the entire ASDT, then they can append the template to `$ATST/admin/site.config`. Alternatively users can keep the configuration files for the various packages separate, and create the new `$ATST/admin/<pkg>/<pkg>Site.config` file. The package devel framework will always source both files (`admin/site.config` first). This means that options in the package specific configuration file have priority. As a third option users can pass in the configuration file to use with the `--file=FILE` option.

For a complete list of the flags and options that can be passed to the `pkgDevel` script see it's built in help by using the `--help` option.

6.9.2. Creating a Package Devel

Now that you understand the basic steps for end users, we can start looking at how to build your own script. The first question to ask is what fundamental option your users should be able to select when configuring the system. In theory any CSF-Property could be a configurable option. In practice however many are not really options (e.g. for the system to work they must have a given value). Other options are very unlikely to change (e.g. timeout values) and do not warrant inclusion in the configuration file. This leaves things that the typical user may want to change or alter. This includes location specific things, such as the name of the host machine that the application's containers run on, whether to user real or simulated hardware connections, etc. After you have a list of these fundamental options you are ready to create the `pkgSite.config.template` file.

After you have your configuration file, you are ready to start writing the script (or more precisely functions) that will make use of the data. There are four zsh functions which must be placed in a

`$ATST/admin/<pkg>/<pkg>Devel.functions` file. These four functions are responsible for doing the different pieces of configuration.

- `doInitData` -- This function should populate $ATST/data with any data files needed by the system. If CSF used package devel, this is where it would turn things like the `ATST_ID_DB_HOST` into a data file.
- `doInitBin` -- This function should populate $ATST/bin with any scripts or binaries that are needed by the system. Scripts found in the source directories and installed via "`make install_scripts`" are installed as is without any chance of modifications. Scripts installed with `doInitBin` in contrast may start with a template and modify that template. This is where most systems should take the four $ATST/admin/templates/systemXxx.sh templates and modify/rename them to handle starting and stopping their system.
- `doInitDb` -- This function should create any databases needed by the package. If CSF used package devel this is where it would create the Property DB, ID DB, Constant DB… This will likely be a no-op for most systems.
- `doLoadDb` -- This function should load any data into CSF-provided, and/or package specific databases. This includes loading application data, properties, parameter sets, scripts… The expectation is that a template property and app file will be modified and then inserted into the database. This allows properties that are expected to need configuration (e.g. the address of a hardware connection) can be altered by the function, and then loaded into the database.

6.9.3. Final notes

- How to run pkgDevel:

  cd $ATST

  ./admin/pkgDevel PACKAGE_NAME…
- Keep in mind that users are likely to create `<pkg>Site.config` files and as such there should be a `.cvsignore` file which prevents CVS from tracking them
- If there are any other files that you create (e.g. a property file to then later read into the DB) they should also appear in the DB
- Try to clean up old properties and application data before writing new data. Ideally your property files should delete all properties for a component, before adding the new ones. Also you should clear out all components from your containers before setting new ones.
- Have a look at the functions in `$ATST/admin/pkgDevel.functions` for some useful support functions.
- Ask for examples of what other systems did.

7. ATST SOFTWARE DEVELOPMENT TREE

7.1. INTRODUCTION

7.1.1. Purpose

This section describes the layout and use of the ATST Software Development Tree (ASDT). The purpose is to define a common structure for all components of the ATST software system and to provide a uniform basis for software development. This common structure simplifies installation, maintenance, and use of the various parts of the software system. Developers of sections of the ATST software can develop those sections within a framework containing other parts of the software system by taking advantage of the layout and access methodology described here.

7.1.2. Goals

There are any number of ways to structure a software development environment. The approach used by ATST is based on the following goals:

- the structure should distinguish between development and installation environments but all software to run in either type of environment with minimal adjustments
- it should be possible to build and work with multiple, independent development environments
- the structure should support development of software for multiple hardware/software platforms
- a subsystem should be easily integrated into a development environment with other subsystems
- files for each subsystem should be readily identifiable and extractable. In particular, there should be namespace isolation between subsystems
- external packages must be easily integrated into the structure
- the structure should 'protect' the source tree by building into areas separate from the source tree
- the structure should permit builds of subsections of the development tree
- the entire structure should be easily relocatable
- development within the structure should be independent of the developers personal environment—for example, no tools provided as part of the source tree should depend upon the type of shell used by the developer, nor upon the values of environment variables set by the developer

7.2. RELEASES

ATST software is organized around periodic releases. Each major release provides a specific level of functionality while minor releases provide bug fixes and minor improvements. A source code repository is used to manage releases. At the current time, this is a CVS repository but this may change in the future. The ATST project office is responsible for maintaining this repository.

Generally speaking, the specific release identification is only of interest when an ASDT is first created - development within that ASDT can be thought of as extending that specific release. Occasionally, extensions on one release may be merged into other (possibly new) releases as well.

7.3. STRUCTURE

The ASDT structure directly supports many of these design goals and relies upon access methods to support the remaining goals. This section describes the physical structure. The access methods are described in the following sections.

The ASDT is separate from the ATST Source Code Repository. The repository is a storage structure designed to hold the sources from which instances of the ASDT can be constructed. The relationship between ASDT and the repository is discussed in more detail in a later section.

An instance of the ASDT is called a release and can be a development environment or the installed software environment. All ASDT releases share a common structure whose top level is show in the following figure:

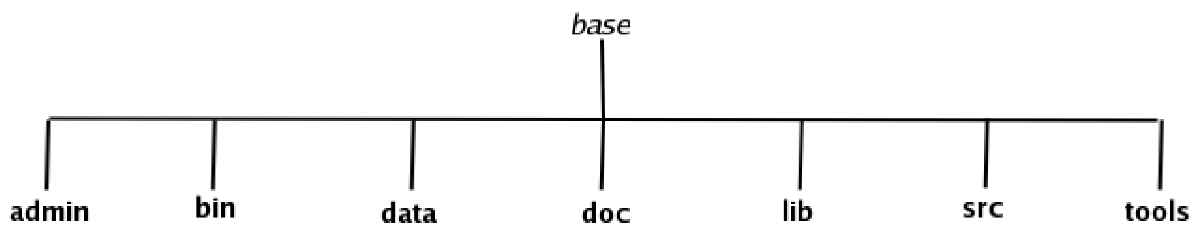

Figure 6. Structure of the Common Services directory

Each of the directories is discussed in turn.

7.3.1. base

The base directory name is arbitrary. There can be separate bases for multiple ASDTs. Convention, however, suggests that /opt/atst be the base of the installation release. Also by convention the environment variable ATST should point to the base of the current ASDT. (This is the only environment variable needed to work within an ASDT.)

7.3.2. admin

The admin directory is optional for most ASDTs. This directory holds the administrative support for manipulating an ASDT. In particular it includes the templates for various configuration files and scripts for creating an ASDT.

Normally, when an ASDT is created, the creation process populates the tree with third party tools and subsystem sources for the current ATST software release. Two files control which versions of these items are installed: admin/templates/tools.list for the third party tools and admin/templates/systems.list for the subsystems. If you want to modify either of these two files you should do so in your own copy of admin. There is normally no need to modify these or any other files in admin/templates.

The file admin/site.config.template is available as a template for describing site-specific configuration information. When installing at a new site, a copy of this file should be named site.config and edited to include site-specifics. The createDevel script used to install an ASDT uses the admin/site.config file to set up site-specific features of an ASDT. Details of the parameters defined in admin/site.config can be found in the Administration section of the ATST Common Services Reference Guide.

The contents of the admin directory are under change control and maintained in the ATST source code repository.

7.3.3. bin

The bin directory holds the applications that have been released in this ASDT. There are two classes of applications: administrative and ATST applications. Administrative applications are those applications needed to support the use of this ASDT. These include routines for adding Makefiles to source code packages, scripts that run Java applications in the environment(s) provided by this ATST, and scripts to simplify adding new software systems to this development environment. Administrative applications come from the admin/templates directory and are installed into the bin directory by running createDevel (more on that below). ATST applications are applications that are part of the ATST software system itself and typically come from elsewhere in the ASDT, and are installed into the bind directory by make (more on that below).

7.3.4. data

The data directory holds configuration files needed to work within the ASDT. For example, creating an ASDT populates the data directory with configuration files containing useful defines and functions need by the Makefiles and administrative scripts. As ATST is a database-oriented software environment, the runtime configuration information in the data directory is generally restricted to only those configuration parameters needed to bootstrap a running ATST system.

As with the bin directory, data contains subdirectories for each ATST system found in this ASDT. So, data/ocs contains configuration information for the Observatory Control System.

7.3.5. doc

Documentation is held in the doc directory. There are three classes of documentation:

user guides and manuals found in system-specific subdirectories of doc/guides

source code documentation found in system-specific subdirectories of doc/api-docs.

uml diagrams found in system-specific subdirectories of doc/uml.

7.3.6. lib

Libraries are found in architecture dependent subdirectories of the lib directory. For example, all released Java classes and jar files are found in lib/java. Several subdirectories of lib/java deserve special note.

The lib/java/classes directory holds the class file hierarchy for all Java classes released into this ASDT. These files are automatically installed when a build version of the ASDT is released.

The lib/java/ext directory holds Java jar files (more precisely, symbolic links to Java jar files) that are to be automatically searched when compiling or running Java applications. This includes jar files built from sources in the ASDT as well as jar files provided by 3rd party software packages. For example, symbolic links to the jar files provided by ICE and Berkeley DB are usually found in lib/java/ext.

Java source files that are generated from SLICE sources are found in lib/java/generated_java.

7.3.7. src

The src directory holds all ATST source code for the ATST systems found in this ASDT. For example, the common services Java source code is accessed through src/java/atst/cs.

The subdirectories of src deserve further explanation:

slice holds all SLICE source files.

java/atst holds the Java source files of all ATST systems available in this ASDT

c++/atst holds the C++ source files of all ATST systems available in this ASDT

The src directory and subdirectories of src/java, src/c++ and src/slice are under change control and maintained in the ATST source code repository.

7.3.8. tools

Third party tools are installed into system specific subdirectories of the tools directory. Symbolic links typically exist to allow easy switching between different versions of the same tool.

7.4. ADDITIONAL PACKAGES

The ASDT provides some support for integrating additional software packages. The general approach for this is to create sub-directories within an ASDT that are NOT parts of the CS CVS repository but are part of separate software repositories. When the additional package is checked out its directory structure is interwoven with the CS directory structure. There are a number of specific places where additional packages are expected to place their code.

- $ATST/admin/<package>/ should contain installation files and scripts. All of the package devel stuff mentioned in section 6.9.
- $ATST/doc/guides/<package>/ should contain users manuals, reference manuals, and any other document which is convenient to have co-located with the source code.
- $ATST/doc/uml/<package>/ should contain all UML diagrams that are convenient to have co-located with the source code.
- $ATST/resources/screens<package>/ should contain the xml definitions for all Java Engineering Screens (JES)
- $ATST/resources/images/<package>/ should contain all images that are needed for JES, or other things at runtime.
- $ATST/src/java/atst/<package>/ should contain all Java (and JNI) source code.
- $ATST/src/c++/atst/<package>/ should contain all C++ source code.
- $ATST/src/idl/atst/<package>/ should contain all IDL source code.
- $ATST/src/jython/atst/<package>/ should contain all Jython source code.
- $ATST/src/powerpmac/atst/<package>/ should contain all the powerpmac project files.
- $ATST/tools/<package>/ should contain any third party tools that are needed. Tools should be distributed as tar balls, which will be un-tarred and included in the library/class path. In addition to checking out the directory you must modify your site.config to include the package name in the ATST_PACKAGE_LIST to ensure that the packages are properly installed when "createDevel --init-tools" is run.

Note that no guarantees are made that the source and doc directories will be available at runtime. It should be possible to trim those directories from an ASDT and successfully run. Anything that needs to be present at runtime should be in the resources directories.

While this interleaving approach works fairly well there are a few short comings. Specifically it is currently not possible to chain additional functionality into createDevel.

7.5. WORKING WITHIN AN ASDT

The installed software environment is the ASDT that has been selected to actively control the ATST telescope and instruments. The base of the installed ASDT is always found in the environment variable ATSTROOT. So, for example, the class files currently used for ATST system operations can be reached at the shell level with $ATSTROOT/lib/java/classes. Code should not, however, refer to the ATSTROOT variable directly, except in checks to determine if a section of code is currently running as a part of the installed system or not.

When developing and testing, there is no need to set ATSTROOT at all.

7.5.1. ATST environment variable

The only environment variable that is absolutely required to work with ATST is the ATST environment variable. This must point to the root directory of the ASDT.

Since the ATSTROOT environment variable is not permitted in code, how does code gain access to other files in the same ASDT? There are two methods:

access to files within the same subsystem should always use relative paths. This allows subsystem code to be moved between ASDTs easily

access to ASDT files outside of the current subsystem should be based off of the ATST environment variable.

Note that the values of ATST and ATSTROOT are identical only in the installed ASDT. Scripts supplied with an ASDT and scripts written as part of ATST software use the ATST environment variable in this way. Only the ATST environment variable needs to be set. ASDT provides all access support based off of the ATST environment variable by giving access to support files in $ATST/data and to $ATST/Make.common for use with makefiles.

7.5.2. Makefile support

Make.common provides definitions that allow makefiles to easily access ASDT functionality needed to use make when building from source code. The makefile Makefile.master, which is the key makefile for building from ICE or Java source code, provides examples of the use of definitions from Make.common.

The file Makefile.master is rarely referenced directly. Instead, every source directory (and $ATST) contains a Makefile. Running the make command without any arguments in any of these directories produces a help message explaining what may be built in that directory. With the exception of running make in $ATST, all makes build only in the current directory. Running make in $ATST build across the entire ASDT.

7.5.3. Script support

In $ATST/data, there are several files supporting ATST scripts, including those added to $ATST/bin when the ASDT is created. The file functions.zsh is intended to be referenced from all zsh scripts. It identifies the current ATST software release, determines the host architecture and sets numerous environment variables (such as Java's CLASSPATH variable) needed to operate within the ASDT. To perform its work, functions.zsh sources functions.common (for some commonly useful, but not ATST-specific, shell functions) and then a file containing architecture-specific parameter values. For example, when running on a Linux-based architecture, the file functions.linux is sourced by functions.zsh.

A zsh script intended for use in ATST need only source functions.zsh. Most of the scripts found in $ATST/bin can be used as examples of this use. The $ATST/bin/ajavac script, used to invoke the javac command in the ATST build environment, is a typical example.

7.5.4. Running Java applications from an ASDT

Although the normal access to a released Java application is expected to be through a shell script wrapper, the command $ATST/bin/ajava can be used to directly execute java classes from a built ASDT. The ajava command invokes the proper java runtime with the appropriate parameters for the ASDT rooted at $ATST. There is no need to set a classpath or any environmental parameter other than $ATST. A call to StandAlone.startServices(…) is needed to access the CSF Service from the stand alone Java application. A call to StandAlone.stopServices() should be made with the application exits.

7.5.5. Running C++ applications from an ASDT

Although the normal access to a released C++ application is expected to be though a shell script wrapper, the command $ATST/bin/runC++ can be used to directly execute C++ executables from a built ASDT. The runC++ command invokes the appropriate executable with a properly defined LD_LIBRARY_PATH. The script $ATST/bin/debugC++ sets the execution environment just like runC++, but instead of invoking the application directly it uses gdb to invoke it. (Note that gdb must be installed) This can be useful for debugging C++ applications. Just like with Java stand alone application, C++ stand alone applications need to call StandAlone.startServices(…) in order to access CSF Services and a call to StandAlone.stopServices() should be made when the application exits.

7.5.6. Runtime data directories

Some applications need to be able to read/write temporary files during operation. By convention, these files should not be placed into the ASDT. Instead, a temporary file directory is available. The pathname of this directory can be found in the environment variable RUN_ENVIRON when running ATST applications and is also available as the Java system property RUN_ENVIRON. This directory is already used to hold communication system logs, logs from containers started with the startContainer script, and other items.

7.5.7. ATST and ssh

ATST is a highly distributable system that uses ssh to connect to remote machines (and often the local machine as well!). This section presents a simple outline of how to set up ssh so ATST isn't constantly asking for passwords. You should consult the ssh documentation for more detail.

7.5.7.1. Ssh files

A quick and dirty approach to setting up password-less ssh is to run the following command:

```
-> $ATST/bin/ssh-setup
```

The above command will only work if you have not used ssh much, and it will only allow you to ssh into your local machine. If you need to be able to connect to more machines read the script mentioned above to understand what it has done, and read the instructions below to understand what you still need to do.

In your home directory is a hidden directory .ssh that contains your ssh configuration files. For example, ~/.ssh/known_hosts contains a list of all the machines that you have connected to with ssh [==known_hosts== is managed for you by ssh, unless ssh tells you there is a bad key in this file, you don't have to worry about it].

There are three that you'll need to know about to have password-less logins. From the ssh man page:

```
$HOME/.ssh/authorized_keys
Lists the public keys (RSA/DSA) that can be used for logging in
as this user.  The format of this file is described in the
sshd(8) manual page.  In the simplest form the format is the
same as the .pub identity files.  This file is not highly sensi-
tive, but the recommended permissions are read/write for the
user, and not accessible by others.
```

The file authorized_keys contains the public keys for all machine/user combos that should be allowed to login into the current machine/user without a password. For ATST, you should make sure that the local machine also has an entry in this file, so applications running on the local machine can ssh to the same machine without a password.

```
$HOME/.ssh/identity, $HOME/.ssh/id_dsa, $HOME/.ssh/id_rsa
Contains the authentication identity of the user.  They are for
protocol 1 RSA, protocol 2 DSA, and protocol 2 RSA, respec-
tively.  These files contain sensitive data and should be read-
able by the user but not accessible by others (read/write/exe-
cute).  Note that ssh ignores a private key file if it is acces-
sible by others.  It is possible to specify a passphrase when
generating the key; the passphrase will be used to encrypt the
sensitive part of this file using 3DES.
```

Although this looks like three files, you only need one, matching the encryption protocal you've chosen (I'll assume rsa for this discussion): id_rsa

```
$HOME/.ssh/identity.pub, $HOME/.ssh/id_dsa.pub, $HOME/.ssh/id_rsa.pub
Contains the public key for authentication (public part of the
identity file in human-readable form).  The contents of the
$HOME/.ssh/identity.pub file should be added to the file
$HOME/.ssh/authorized_keys on all machines where the user wishes
to log in using protocol version 1 RSA authentication.  The con-
tents of the $HOME/.ssh/id_dsa.pub and $HOME/.ssh/id_rsa.pub
file should be added to $HOME/.ssh/authorized_keys on all
machines where the user wishes to log in using protocol version
2 DSA/RSA authentication.  These files are not sensitive and can
(but need not) be readable by anyone.  These files are never
used automatically and are not necessary; they are only provided
for the convenience of the user.
```

As with id_rsa, only one of these is needed: id_rsa.pub

These two files hold the private and public key parts for the current user/machine combo. (So, the contents of the id_rsa.pub on machine X should be appended to the authorized_keys file on machine Y if you want to be able to log into Y from X without a password.)

7.5.7.2. Creating the public and private keys

This section gives an example showing how to create the ssh public and private keys.

To set up your ssh keys:

On each machine X (note, if you share a common home directory among all the machines using, say, automount, you only have to do this on one machine):

```
cd ~/.ssh
ssh-keygen -t rsa
# You'll be asked for the filename, just press enter

# You'll be asked for a 'passphrase', just press enter
```

```
# You'll be asked for the same passphrase again, just press enter
```

This creates id_rsa and id_rsa.pub.

Append id_rsa.pub to authorized_keys (or authorized_keys2, if that exists!):

```
cd ~/.ssh
touch authorized_keys
chmod 600 authorized_keys
cat id_rsa.pub >>authorized_keys
```

Among other things, this lets you ssh from machine X to machine X without a password (very useful when deploying containers!)

To allow passwordless connections from machine X to machine Y:

Copy the id_rsa.pub file from machine X to machine Y, giving it a new name (this protects the existing id_rsa.pub file on Y):

```
# From machine X
scp id_rsa.pub Y:.ssh/X_id_rsa.pub
# You'll be asked for the password on machine Y.
```

Now go to machine Y and append the public key for machine X to authorized keys (again, this isn't needed if you share a common account on machines X and Y):

```
ssh Y
  # You'll be asked for the password on machine Y.
  # You are now on machine Y
cd .ssh
cat X_id_rsa.pub >>authorized_keys
chmod 600 authorized_keys
logout
  # You are now on machine X again
```

Test your connections between all the machines using ssh. You should not have to give a password.

8. SYSTEM HARDWARE REQUIREMENTS

There are three types of machines used within a Common Services Framework control system: infrastructure machines, service servers, and service clients. The requirements for the infrastructure machines (network switches) and servers (event server, log server …) vary depending on many factors including the number of clients, and how hard those clients will be using the servers. There is no simple formula for specifying a server or group of servers. Contact the ATST Software Team for advice on a particular installation.

The minimum recommended configuration (hardware and some software) for a CSF client is:

Architecture:	x86_64
Cores:	4 or more, depending on load
CPU speed:	2.8 GHz and up
RAM:	8 or more GB
Network:	GbEthernet
Disk:	250GB and up
Operating System:	CentOS 6.x
PostrgreSQL:	version 9.3
ICE	version 3.4.x
Java:	JDK 1.7.x
C++:	version 4.4.x

The above list excludes any additional hardware and or software to run your particular application (data processing/data pipeline, timing hardware, specific IO …) but should comfortably support a typical CSF setup: one java container and one C++ container, each with 10-15 components.