Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - ♦ News
  - ♦ Design
  - ♦ Common Services
  - ♦ Application base
  - ♦ OCS
  - ♦ DHS
  - ♦ TCS
  - ♦ ICS
  - ♦ ICDs
  - ♦ Docs
  - ♦ Meeting Notes
  - ♦ Problem Tracking
  - ♦ Changes

- **Misc**
  - ♦ Users
  - ♦ Groups
  - ♦ Offices
  - ♦ Topic list
  - ♦ Search

- **TWiki Webs**
  - ♦ Know
  - ♦ Main
  - ♦ Sandbox
  - ♦ TWiki

Create personal sidebar

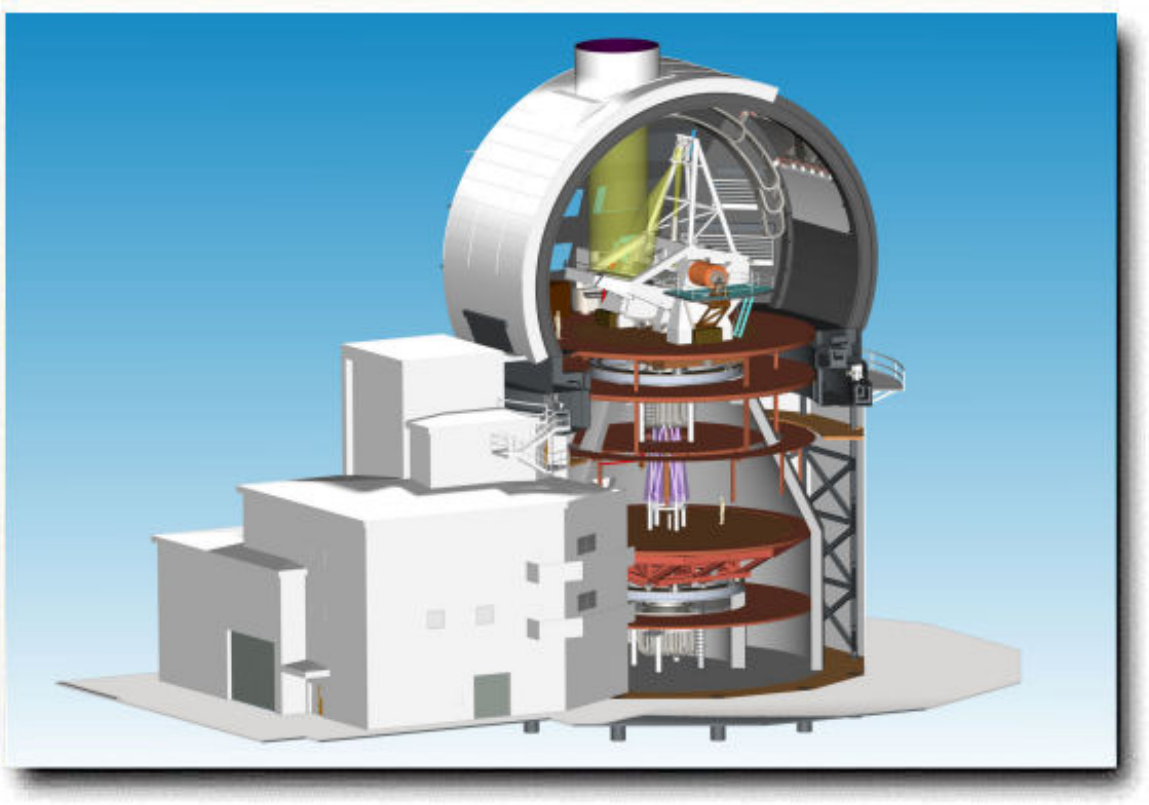
- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideCoverr1.13 - 12 Feb 2007 - 15:27 - SteveWamplertopic end

Project Documentation
SPEC-0022-2
Revision A-7

ATST Common Services Reference Guide



ATST Software Group

Panguitch-1P7
February 12, 2007
Tucson, Arizona

Advanced Technology Solar Telescope
950 N. Cherry Ave
Tucson, AZ 85719
Phone 520-318-8102
atst@nso.edu http://atst.nso.edu
Fax 520-318-8500

-- SteveWampler - 1 Feb 2006

to top

Table of Contents

Table of Contents

Table of Contents

ATST Common Services
Software Design Document

1 Introduction

1.1 Overview

1.1.1 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 (ATSTCS) 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.

1.1.2 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.

1.1.3 Design Highlights

Most of the design of the ATST Common Services 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.

1.1.3.1 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.

1.1.3.2 Separation of Functional and Technical behavior

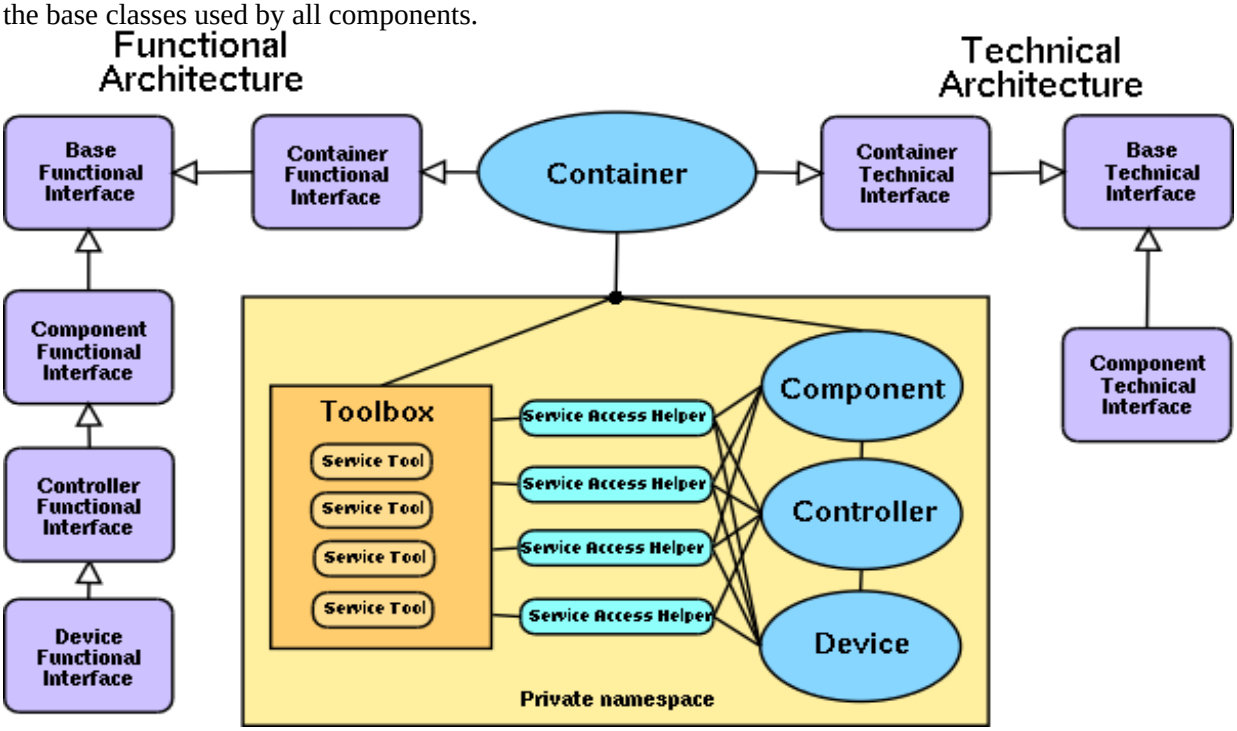
The ATST software design distinguishes between *functional* and *technical* behavior. Functional behavior describes the actions taken to directly implement ATST operations and can be contrasted with the technical behavior - the actions required of the infrastructure needed to support the functional behavior. 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, the application developer can concentrate on providing the required functionality.

1.1.3.3 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 the 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*.

1.1.3.4 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.



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

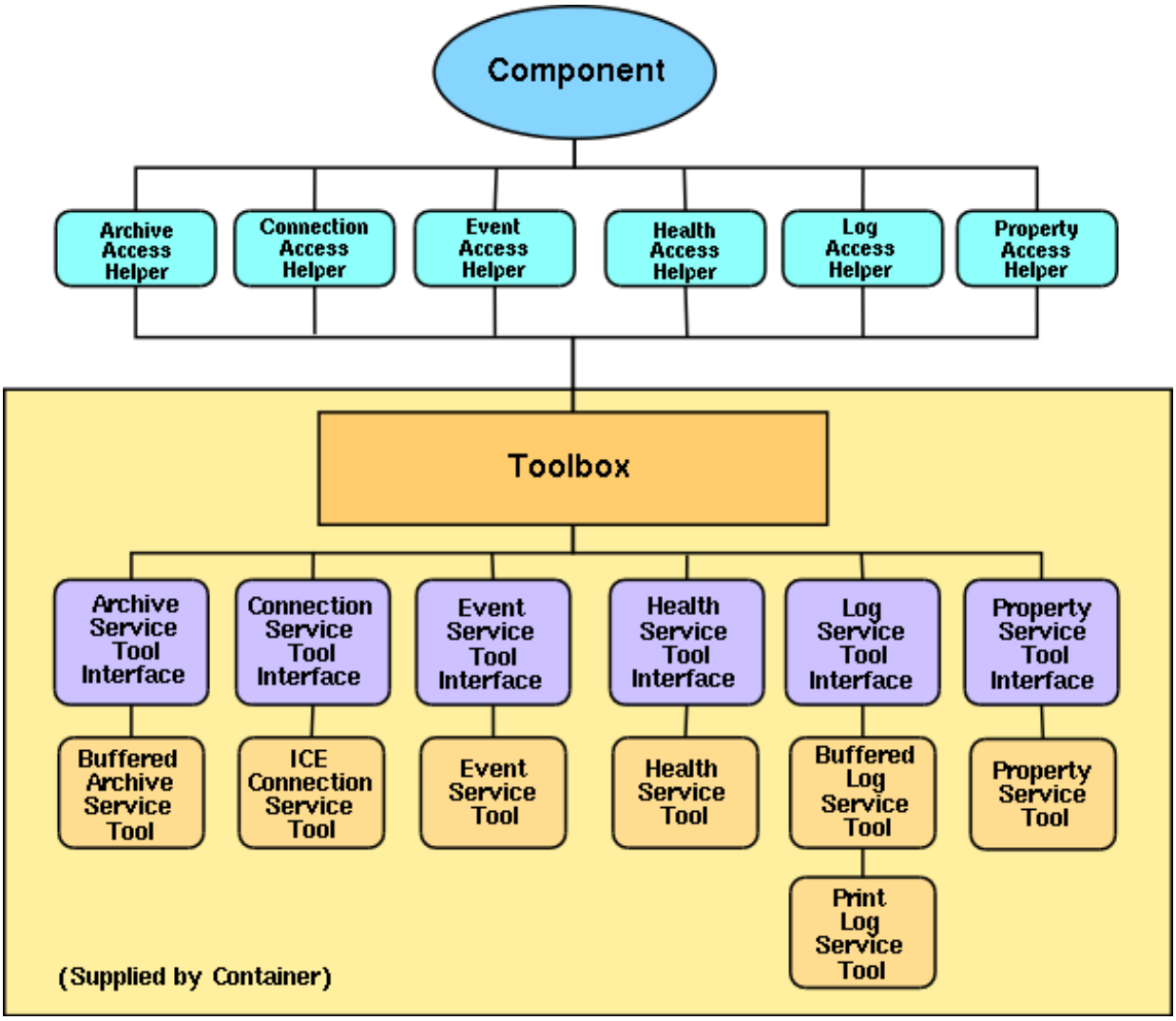
Containers provide a uniform means of deploying and controlling the technical aspects of Component operations. Component developers can develop Components without a detailed understanding of Containers.

1.1.3.5 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 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.*



-- SteveWampler - 12 Jan 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.25 | ≥ | r1.24 | ≥ | r1.23 | Total page history | Backlinks
Main.AtstCsSDDOverview moved from Main.AtstCsManualOverview on 18 Feb 2005 - 16:57 by SteveWampler - put it back
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsSDDOverview

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - ♦ News
  - ♦ Design
  - ♦ Common Services
  - ♦ Application base
  - ♦ OCS
  - ♦ DHS
  - ♦ TCS
  - ♦ ICS
  - ♦ ICDs
  - ♦ Docs
  - ♦ Meeting Notes
  - ♦ Problem Tracking
  - ♦ Changes

- **Misc**
  - ♦ Users
  - ♦ Groups
  - ♦ Offices
  - ♦ Topic list
  - ♦ Search

- **TWiki Webs**
  - ♦ Know
  - ♦ Main
  - ♦ Sandbox
  - ♦ TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

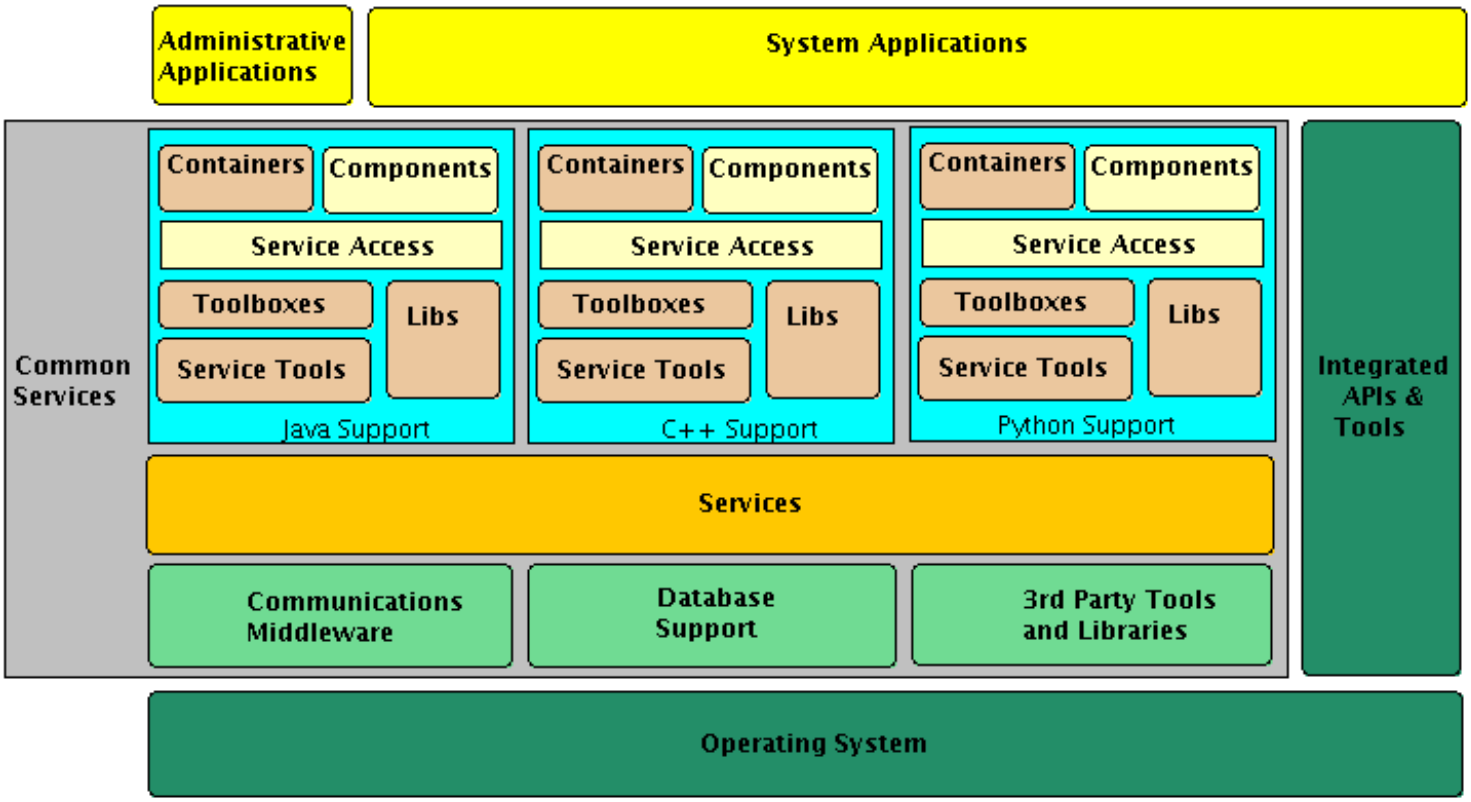
Main.AtstCsGuideCSInfrastructurer1.15 - 23 Aug 2006 - 18:32 - JanetTvedttopic end

---

2 Infrastructure

2.1 Common Services Layers

One of the major goals of the ATST Common Services 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*:



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 logicial 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++, Java, and Python 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 Components and Containers

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

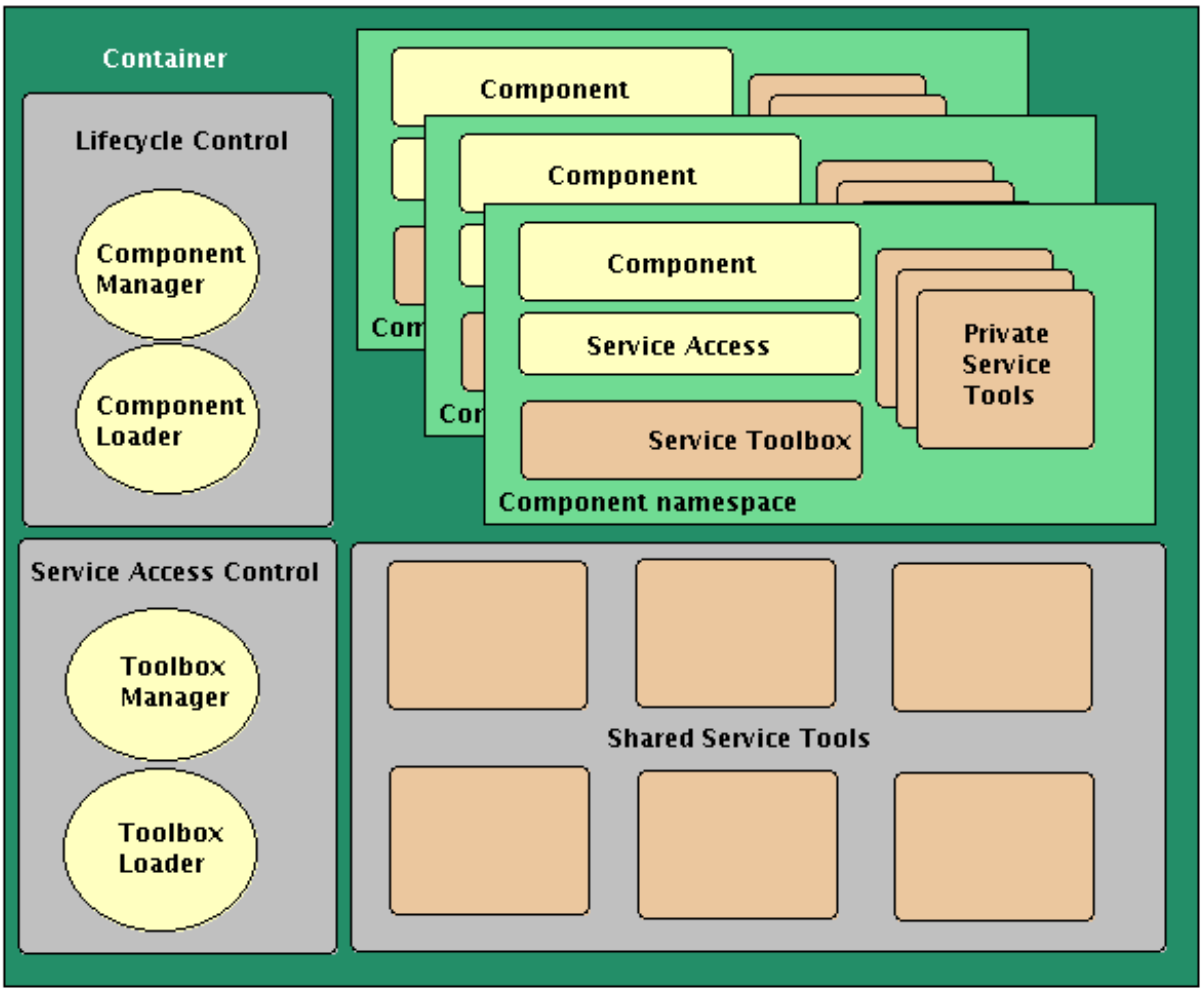
2.2.1 Containers

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. Although there are no technical prohibitions against doing so, ATST generally does not run more than one instance of a language-specific Container on a given host.

The following figure shows the basic structure of all ATST Containers:



The Container's *lifecycle control* module is responsible for managing the Component lifecycles. It responds to external requests on the Container to create, start, stop, 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.

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.

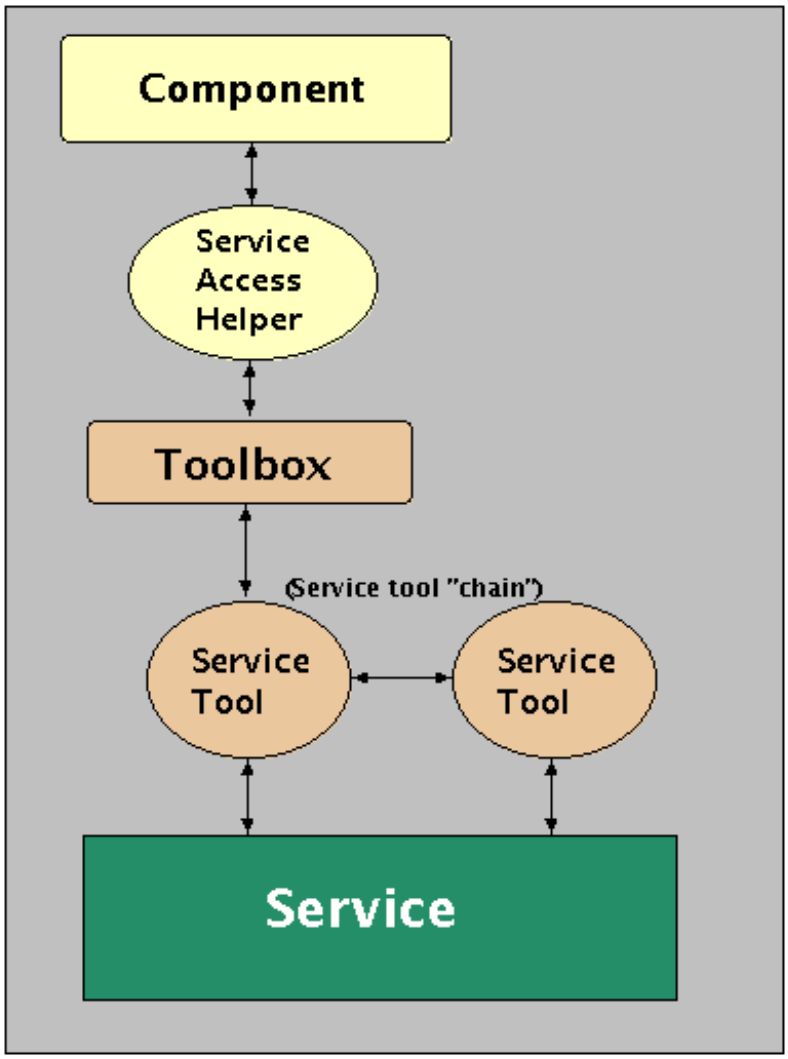
2.4 Toolboxes and service tools

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.

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 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 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 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 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 3rd party tools

In addition to ICE and PostgreSQL, the implementation of the ATST Common Services 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.

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 fast 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 an identification for the ATST *observation* with which it is associated, and maintains an *action state*. Details of the roles of these items is covered on the section on ***Controllers*** .

-- SteveWampler - 22 Feb 2005
to top

End of topic
Skip to action links | Back to top

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - ♦ News
  - ♦ Design
  - ♦ Common Services
  - ♦ Application base
  - ♦ OCS
  - ♦ DHS
  - ♦ TCS
  - ♦ ICS
  - ♦ ICDs
  - ♦ Docs
  - ♦ Meeting Notes
  - ♦ Problem Tracking
  - ♦ Changes

- **Misc**
  - ♦ Users
  - ♦ Groups
  - ♦ Offices
  - ♦ Topic list
  - ♦ Search

- **TWiki Webs**
  - ♦ Know
  - ♦ Main
  - ♦ Sandbox
  - ♦ TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideServicesr1.19 - 25 Aug 2006 - 21:11 - JanetTvedttopic end

---

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

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.

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

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

- **`void log(String mode, String category, int level, String message)`** -- log a message
- **`void setDebugLevel(String category, int level)`** -- set a category's current debug level
- **`int  getDebugLevel(String category)`** -- get a category's current debug level

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):

- **`PrintLogServiceTool`** -- this tool sends all log messages to *standard error*.
- **`LogServiceTool`** -- a stub for recording log messages to the log database individually as they are produced as database support has not been implemented.

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(const string& timeStamp, const string& source, const string& mode,
const string& category, int level, const 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:

```
// Log a debug message
void debug(const std::string& category, const int level,
const std::string& message);

// Log a note
void note(const std::string& category, const std::string& message);

// Log a warning message
void warn(const std::string& category, const std::string& message);

// Log a severe message
void severe(const std::string& category, const std::string& message);
```

```
// Set a category's current debug level
void setDebugLevel(const string& category, int level);

// Get a category's current debug level
int  getDebugLevel(const 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 Python support

TBD

3.1.7 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 two classes of interfaces:

- **controller** interface -- most real-time Components present this interface that includes methods for directing the execution of *Configurations* by the controller.
- **custom** interfaces -- some specialized Components present unique interfaces

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 indistinquishable 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 respository. 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`** -- provide 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:

- **`void bind(IToolBoxAdmin tb, String name, atst.cs.interfaces.IRemote object)`** -- makes the named object available for remote access.
- **`void unbind(IToolBoxAdmin tb, String name)`** -- makes the named object unavailable for remote access and releases any unneeded resources.
- **`atst.cs.interfaces.IRemote connect(IToolBoxAdmin tb, String source, String target)`** connect to 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)`** disconnect from the named target object. The name of the source Component is provided in case the connection service tool is shared.

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.

3.2.5.3 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:

- **`void bind(atst.cs.interfaces.IRemote object)`** -- register the Component object
- **`void unbind()`** -- unregister the Component
- **`atst.cs.interfaces.IRemote connect(String targetName)`** -- connect to the named target
- **`void disconnect(String targetName)`** -- disconnect from the named target

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.

3.2.7 Python support

TBD

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`. Subscribers may subscribe using *wildcarded names*, receiving any events whose names are matched by the wildcarded name. So, for example, subscribers using the name `"tcs.*"` would see *every event* published by any Component using any name starting with `"tcs."` and a subscriber using the name `"*"` would see all events (this is **not recommended**, as there are profound performance implications of doing so).

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`.
- **`String getString(String attributeName)`** -- get the named Attribute's value as a Java string.
- **`double getDouble()`** -- get the named Attribute's value as a Java Double.
- **`long getLong()`** -- get the named Attribute's value as a Java Long.

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:

- **`void subscribe(String eventName, atst.cs.interfaces.IEventCallback callback)`**
- **`void unsubscribe(String eventName, atst.cs.interfaces.IEventCallback callback)`**
- **`void post(String eventName, atst.cs.data.IAttributeTable value)`**

These have their obvious interpretations.

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:

```
void callback(const 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:

```
// Produce the entire value as an AttributeTable
tr1::shared_ptr<IAttributeTable> getValue();

// Get the named =Attribute=
IAttribute* getAttribute(String attributeName);

// Get the named Attribute's value as a std::string
string getString(const string& attributeName);

// Get the named Attribute's value as a double
double getDouble(const string& attributeName);

// Get the named Attribute's value as a long.
long getLong(const string& attributeName);
```

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`** throw an IllegalConversionException 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.

3.3.4.2 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:

```
void subscribe(const string& eventName, const IEventCallback& callback);
void unsubscribe(const string& eventName, const IEventCallback& callback);
void post(const string& eventName, tr1::shared_ptr<IAttributeTable> value);
```

These have their obvious interpretations.

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.

3.3.5 Python support

TBD

3.4 Health service

The Health service provides a uniform mechanism for monitoring ATST applications. It provides both a *watchdog* service for determining the connectivity of 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 set their own health status, but are expected to provide significant input into the determination of their health status. In particular, Component developers must implement a method `doHealthCheck` for use by the health service when determining the Component's health. The `doHealthCheck` method must return one of **GOOD**, **ILL**, or **BAD** and can make use of support functions provided by the health service when determining this value. The health service may *demote* the status returned by `doHealthCheck` (e.g. move **GOOD** to **ILL**, **ILL** to **BAD**, or **BAD** to **UNKNOWN**) but is not permitted to *promote* the returned status.

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 Component by invoking the Component's `healthCheck` method (which invokes **`doHealthCheck()`** internally). The polling interval is controlled by the health service but may be different for different Components and can change dynamically. A timeout mechanism prevents hanging the health service during polling. Repeated timeouts automatically demote the Component's health status.

Changes to health are automatically logged by the health service, which also posts an event on health status changes. The log messages get progressively more dire as the health worsens. The posted event's name is formed by appending **`.health`** to the Component name, e.g. **`atst.tcs.mcs.health`** and contains a single string-valued `Attribute` with the same name. The value of that attribute is the current health status as one of the strings **`"good"`**, **`"ill"`**, **`"bad"`**, or **`"unknown"`**.

3.4.1 Health service dependencies

The health service depends upon the Log, Connection, 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 `doHealthCheck` method

Component developers must implement one method in support of the health service:

- **`String doHealthCheck()`**

This method returns one of:

- **`atst.cs.services.Health.GOOD`**
- **`atst.cs.services.Health.ILL`**
- **`atst.cs.services.Health.BAD`**

where the meanings of the return value matches the corresponding health status definition given above.

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

- **`String atst.cs.services.Health.getHealth()`** -- produces the current health for this Component
- **`String atst.cs.services.Health.setHealth(String health)`** -- records the current health of this Component

The current health of a Component is maintained in that Component's Toolbox. The **`setHealth`** method is a *convenience* method - it simply records the health status in the Component's Toolbox. A simple **`doHealthCheck`** method might retrieve this health status with **`getHeath()`** and return this value.

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

The **`LoggingHealthServiceTool`** must be loaded as a *private* service tool.

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:

- **`String reportHealth(IToolBoxAdmin tb, IHealthReport)`** -- report a change in helath.

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:

- **`String setHealth(String healthStatus)`** -- set the health status for this Component
- **`String getHealth()`** -- get the health status for this Component

in addition, the following method in this interface is used by the health service to check the health of the Toolbox's Component:

- **`string checkHealth()`** -- check the Components health

The code for this method in the Toolbox simply invokes the Component's `checkHealth` method.

3.4.3.5 Component access to the health service

See the Users' Manual for more details on how Components use the **`atst.cs.services.Health`** class to access the health service.

3.4.4 C++ support

3.4.4.1 The `doHealthCheck` method

Component developers must implement one method in support of the health service:

```
string doHealthCheck();
```

This method returns one of:

```
atst::cs::services::Health::GOOD
atst::cs::services::Health::ILL
atst::cs::services::Health::BAD
```

where the meanings of the return value matches the corresponding health status definition given above.

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

```
// Produces the current health for this Component
string atst::cs::services::Health::getHealth();

// Records the current health of this Component
string atst::cs::services::Health.setHealth(const string& health);
```

The current health of a Component is maintained in that Component's ToolBox. The **`setHealth`** method is a *convenience* method - it simply records the health status in the Component's ToolBox. A simple **`doHealthCheck`** method might retrieve this health status with **`getHeath()`** and return this value.

3.4.4.2 Health service tools

Currently, only one health service tool is available:

```
// A stub that writes health changes to standard output
atst::cs::services::health::SimpleHealthServiceTool
```

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: ***TBD***

3.4.4.4 Health service access to the ToolBox

The **`atst::cs::interfaces::IToolbox`** interface defines methods for performing health actions on the ToolBox. The methods defined by this interface are: ***TBD***

3.4.4.5 Component access to the health service

See the Users' Manual for more details on how Components use the **`atst::cs::services::Health`** class to access the health service.

3.4.5 Python support

TBD

-- SteveWampler - 01 Mar 2005
to top

End of topic
Skip to action links | Back to top

- **Main Web**
- Main Web Home
    - ♦ News
    - ♦ Design
    - ♦ Common Services
    - ♦ Application base
    - ♦ OCS
    - ♦ DHS
    - ♦ TCS
    - ♦ ICS
    - ♦ ICDs
    - ♦ Docs
    - ♦ Meeting Notes
    - ♦ Problem Tracking
    - ♦ Changes

- **Misc**
    - ♦ Users
    - ♦ Groups
    - ♦ Offices
    - ♦ Topic list
    - ♦ Search

- **TWiki Webs**
    - ♦ Know
    - ♦ Main
    - ♦ Sandbox
    - ♦ TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideToolsr1.12 - 24 Aug 2006 - 21:58 - JanetTvedttopic end

---

Start of topic | Skip to actions

---

4 Minor Services (Tools)

The *minor services*, also known as *tools* (not be 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

Three archive service tools are provided, all of which 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

TBD

4.1.6 Python support

There is no python support anticipated for the archive service.

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

In either mode, the property service can be directed to *invalidate* loaded metadata and reload the metadata from the persistent store. In preload mode that reloading is immediate while in on-demand mode the reloading is also on-demand.

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

The property service is a new addition to the ATSTCS. As the design progresses, this section will contain more information about this service.

4.2.1 Java support

4.2.1.1 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 **`readonly`** 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`**
- **`readonly`** -- a boolean that is true if this attribute is readonly
- **`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 is converted to the appropriate Java type:

- **`"string"`** values are converted to **`String`** objects.
- **`"integer"`** values are converted to **`Long`** objects.
- **`"real"`** values are converted to **`Double`** objects.
- **`"boolean"`** values are converted to **`Boolean`** objects.

4.2.1.2 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 wrapped by either of the two 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.

4.2.1.3 Toolbox access to the property service tools

The Toolbox accesses the property service tool using the **`atst.cs.services.property.IPropertyServiceTool`** interface using the following methods defined in that interface:

- **`IPropertyTable loadProperties(IToolBoxAdmin tb, String componentName)`** -- load the properties for the named Component from the property persistent store. In *on-demand* mode this does **not** load any metadata - but it does clear the property table maintained by the Toolbox.
- **`void reset(IToolBoxAdmin tb, )`** -- reload the property table from the persistent store.
- **`IProperty getProperty(IToolBoxAdmin tb, String componentName, String propertyName)`** -- produce the named property for the named component. If that property doesn't exist, then **`null`** is returned.
- **`IProperty setProperty(IToolBoxAdmin tb, String timeStamp, String componentName, IProperty property)`** -- set the property into the property service. This writes the property to the persistent store.

Loaded properties are kept in the Toolbox for each component as a **`atst.cs.services.property.IPropertyTable`**.

This interface is also used by the **Constant Service** using additional methods.

4.2.1.4 Property service helper access to the Toolbox

The property service access helper uses the following methods in the **`atst.cs.interfaces.IToolbox`** interface to access the property service through the Toolbox.

- **`atst.cs.interfaces.IProperty getProperty(String attributeName)`** -- produce the metadata for the named `Attribute`.
- **`void setProperty(atst.cs.interfaces.IProperty property)`** -- set the metadata for the named `Attribute`.

4.2.1.5 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.

4.2.1.6 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 file for class **`atst.cs.services.property.InternalProperty`** add code to the **`convertOne`** method to convert from the string form for **`xtype`** to the Java representation **`XType`**. Use the existing code as a guide.
4. In the same source code file, add code to the **`inRange`** method for testing an instance of **`xtype`** against the property limits. Use the existing code as a guide.
5. In the same source code file, add code to the **`makeProperty`** factory method to create a correctly typed instance of an **`XType`** property. Use the existing code as a guide.
6. In the source code file for class **`atst.cs.services.Property`** add a new method for setting the value of this type:
   `public void setValue(String pName, XType value)` - you can use one of the existing **`setValue`** methods as a guide.
7. In the same source code file, add a new method for testing a value of this type against the limits:
   `public boolean inRange(String pName, XType value)` - you can use one of the existing **`inRange`** methods as a guide.
8. Add the new type to this documentation.

4.2.2 C++ support

TBD

4.2.3 Python support

TBD

4.3 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).

4.3.1 Java support

4.3.1.1 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.

4.3.1.2 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).

4.3.1.3 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

4.3.1.4 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

4.3.1.5 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.

4.3.2 C++ support

TBD

4.3.3 Python support

TBD

4.4 Monitor service

The monitor service works in conjunction with the property service to support monitoring the values of key `Attributes` within a Component. Both the monitor checks and the resulting actions to be taken are highly configurable.

The monitor service is a new addition to the ATSTCS. As the design progresses, this section will contain more information about this service.

4.4.1 Java support

TBD

4.4.2 C++ support

TBD

4.4.3 Python support

TBD

4.5 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.

At the current time, this support is ***TBD*** . See the documentation on the **`atst.cs.util.gui`** package in the Application Programming Interface document for a descripton of the *very limited* support currently available.

TBD

4.6 Miscellaneous services

The ATSTCS provides access to a number of simple services that may be useful. Some are language specific. Additional details can be found in the Users' Manual and the Application Programming Interface document.

TBD

-- SteveWampler - 03 Mar 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.12 | ≥ | r1.11 | ≥ | r1.10 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsGuide > AtstCsGuideTools

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - ♦ News
  - ♦ Design
  - ♦ Common Services
  - ♦ Application base
  - ♦ OCS
  - ♦ DHS
  - ♦ TCS
  - ♦ ICS
  - ♦ ICDs
  - ♦ Docs
  - ♦ Meeting Notes
  - ♦ Problem Tracking
  - ♦ Changes

- **Misc**
  - ♦ Users
  - ♦ Groups
  - ♦ Offices
  - ♦ Topic list

- Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideComponentsr1.4 - 30 Jun 2006 - 17:56 - SteveWamplertopic end

---

Start of topic | Skip to actions

---

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.

5.1 Components

TBD

-- SteveWampler - 03 Mar 2005
to top

End of topic
Skip to action links | Back to top

Skip to topic | Skip to bottom

- Meeting Notes
- Problem Tracking
- Changes

- **Misc**
  - Users
  - Groups
  - Offices
  - Topic list
  - Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideControllersr1.1 - 10 Mar 2005 - 17:04 - SteveWamplertopic end

---

Start of topic | Skip to actions

5.2 Controllers

-- SteveWampler - 10 Mar 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.1 | Total page history | Backlinks
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsGuide > AtstCsGuideControllers

to top

Skip to topic | Skip to bottom

Jump:
Main

- **ATST Software**

- **Welcome**
- **Register**

- **Main Web**
- Main Web Home
  - ♦ News
  - ♦ Design
  - ♦ Common Services
  - ♦ Application base
  - ♦ OCS
  - ♦ DHS
  - ♦ TCS
  - ♦ ICS
  - ♦ ICDs
  - ♦ Docs
  - ♦ Meeting Notes
  - ♦ Problem Tracking
  - ♦ Changes

- **Misc**
  - ♦ Users
  - ♦ Groups
  - ♦ Offices
  - ♦ Topic list
  - ♦ Search

- **TWiki Webs**
  - ♦ Know
  - ♦ Main
  - ♦ Sandbox
  - ♦ TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideTestHarnessr1.20 - 01 Dec 2006 - 23:07 - JohnHubbardtopic end

5.3 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 distinquish 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.

For more sophisticated tests, developers may customize the test harness through subclassing and redefining a few key methods.

5.3.1 Preparing for use

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

5.3.2 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.

5.3.3 Starting the Test Harness

A script is provided to start the test harness and the user interface (as with all ATST applications, the script requires that you have correctly set the **`ATST`** environment variable):

`$ATST/bin/Testharness [ `*`options`*` ]`

Where the available *options* are:

- **`--path=PATHNAME`** -- specifies a directory that holds/will hold any files that are used or created during a test run. If omitted, the current working directory is used.

- **`--datafile=FILENAME`** -- all data objects in *FILENAME* are loaded into the test harness environment for subsequent use. Here, *FILENAME* has typically been created during a previous use of the test harness.
- **`--alias=FILENAME`** -- all *aliases* in *FILENAME* are loaded into the test harness environment for subsequent use. Here, FILENAME_ has typically been created during a previous use of the test harness.
- **`--saverun=FILENAME`** -- records the commands used during a test run into *FILENAME*. This record can be played back later using the **`--exec=FILENAME`** option.
- **`--exec=FILENAME`** -- read and execute commands from *FILENAME*. This runs the test harness in *batch mode*. Typically, *FILENAME* was created during a previous use of the test harness using the **`--saverun=FILENAME`** command.
- **`--results=FILENAME`** -- redirects all output to *FILENAME*.

- **`--quiet=true`** -- start with informative message display turned off

5.3.4 Available commands

The following commands are available to direct the test. All optional parameters are given below in square brackets (**`[]`**), required arguments are simply italicized. If required arguments are omitted the user is prompted for their values:

Harness Functions

- **`alias`** ***`string1=string2`*** -- from now on when **`$`*`string1`*** is entered it will be replaced with ***`string2`***

- **`deflis`** -- changes the behavior of the default listener
  - **`-q`** supresses printing of **all** config state messages
  - **`-u`** restarts printing of **all** config state messages
  - **`-s`** ***`file name`*** saves configstate messages to a file. If file name is not given it stops saving configstate messages.

- **`exec`** ***`oscommand`*** -- starts an OS command running from inside the test harness. The harness will wait for the command to complete before resuming. Environmental variables can be used in these messages. For example the command **`exec MotorSimulator –config=$ATST/config/fileName`** will replace **`$ATST`** with the environmental variable before handing the command off to the operating system be executed.

- **`execNoWait`** ***`oscommand`*** -- same behavior as the exec command but the harness will not wait for the command to finish.

- **`help`** -- display a list of some of the more common commands.

- **`list`** ***`itemtype`*** -- display a list of all known items of *itemtype*. The possible values for *itemtype* are:
  - **`alias`**
  - **`attribute`**
  - **`component`** -- (includes both Components and Controllers)
  - **`config`**
  - **`container`**
  - **`table`**

- **`listen`** ***`eventname`*** **`[`** ***`classname`*** **`]`** -- listen for events named *eventname* and process them. If *classname* is omitted, then the default behavior is to display received events on standard output. If *classname* is provided, it *must* be a subclass of **`atst.tools.testharness.EventListenerCallback`**.

- **path [*newpath*]** -- displays or changes the data directory
- **quit** -- stops the test harness
- **quiet** -- turn off informative messages
- **unquiet** -- turn informative messages on (these messages start 'on')

Harness Script Functions

The following functions are intended to be used in scripts or to launch scripts. As with all test harness commands these will work as standard user input or from within a script.

- **echo *string*** -- prints the string to the standard output.
- **input [*"a message"*] [*varname*]** -- input blocks future commands until the user has pressed enter. If a message is given the quotes will be stripped off and that message will be presented to the user (without a new line). If no message is given the default "press enter to continue" will be printed. If a var name is given ": " will be appended to the end of the message and the user will be prompted for a input of length > 0. That input will be assigned to the alias var.
- **run *scriptname*** -- starts a script running the same as the **--exec=FILENAME** argument. After the script is done it returns control to the prompt. Scripts can be launched from within other scripts. When one script finishes it starts executing the calling script where it left off. The test harness will look for the filename as given but if that filename is not found it will look for filename.th. Testharness users are urged to use the .th extension to make it clear that a file is a test harness script.
- **wait *seconds*** -- causes the command interpretor to wait *seconds* seconds before parsing the next command.

Data Functions

- **add *itemname name* [*values*]** -- adds a name,value pair to the named item. If *itemname* refers to a **Configuration** or **AttributeTable** then the (name,value) pair form an **Attribute** that is added. If *itemname* refers to an **Attribute**, then that (name,value) pair *becomes* that **Attribute**. The *name* and *values* fields should each be enclosed in single quotes (**'**). The *values* field may consist of a list of values, separated by commas (**,**)
- **loadFile *filename*** -- loads a data file
- **save *filename*** -- saves all data to a file
- **new *itemtype itemname*** -- create a new item. The *itemtype* may be:
  - ♦ **config** -- a **Configuration**, or
  - ♦ **table** -- an **AttributeTable**, or
  - ♦ **attribute** -- an **Attribute**

Component Lifecycle Functions

- **deploy *containername hostname*** -- starts a Container named *containername* running on the host *hostname*.

- **load *componentname classname* [*container*]** -- loads a component/controller class into a container (a default container is used if none is provided here). The *componentname* is the name used to reference the instance of the *classname* for the component/controller.

- **init *componentname* [*tablename*]** -- initialize the named component. If *tablename* is given, that **AttributeTable** is passed to the component's **init()** method.

- **start *componentname* [*tablename*]** -- startup the named component. If *tablename* is given, that **AttributeTable** is passed to the component's **startup()** method.

- **shutdown *componentname*** -- shut the named component down

- **uninit *componentname*** -- uninitialize the named component

- **unload *componentname*** -- unload the named component

Component/Controller Action Functions

- **debug *componentname level*** -- set the debug level for the named component

- **get *componentname tablename*** -- display the results of calling **get(AttributeTable table)** on the named component, where **table** is the **AttributeTable** identified by *tablename*.

- **set *componentname tablename*** -- call **set(AttributeTable table)** on the named component, where **table** is the **AttributeTable** identified by *tablename*.

- **submit *controllername configname*** -- submit a the named **Configuration** to the named Controller.

5.3.5 An example: testing the `TimerController` Controller

The following example illustrates the basic interactive use of the Test Harness. In this case, the Test Harness is being used to test the simple operation of the **atst.base.controllers.TimerController** Component. Long lines have been wrapped. A few comments for clarification have been embedded in the session log shown, these comments all begin with #:

```
-> $ATST/bin/Testharness

# The test harness displays all log messages.
[2006/06/28:13:27:28.834 MST] (note::) default_listener: running...

# (Opening help message deleted for brevity...)

# Start by building up a Configuration that describes the desired timer.
#  This one counts down from 10 seconds, posting a 'tick' position every
#  three seconds while posting asynchronous alarms at 5 and 2.5 seconds
#  from the end of the countdown.
atst> new config timer1
Configuration timer1 added
atst> add timer1 'timer.name' 'timer1'
inserting timer.name: timer1
atst> add timer1 'timer.countdownlength' '10'
inserting timer.countdownlength: 10
atst> add timer1 'timer.alarm' '2.5,5'
```

```
inserting timer.alarm: 2.5, 5
atst> add timer1 'timer.eventrate' '3'
inserting timer.eventrate: 3

# Now load a TimerController instance into the default container
#   and prepare it to accept configurations
atst> load timer atst.base.controllers.TimerController
Component loaded
atst> init timer
component initialized
atst> start timer
Component started

# Listen to position 'tick' events and alarm events
atst> listen pos_timer1
atst> listen alarm_timer1

# Submit the configuration describing the timer
atst> submit timer timer1

# The test harness automatically displays any status events produced during a
# Controller's action processing for a configuration.
{(status, [scheduled]), (__timestamp, [2006/06/28:13:29:46.442 MST]), (__source, [timer]),
                        (reason, []), (configid, [timer1.000000000711.0000000000004])}
Configuration accepted OK
{(status, [running]), (__timestamp, [2006/06/28:13:29:46.481 MST]), (__source, [timer]),
                      (reason, []), (configid, [timer1.000000000711.0000000000004])}

# The test harness is immediately available for more commands, but the timer
# is going to run simultaneously.
atst>
{(__timestamp, [2006/06/28:13:29:47.492 MST]), (__source, [timer]), (pos, [9])}
{(__timestamp, [2006/06/28:13:29:50.492 MST]), (__source, [timer]), (pos, [6])}
{(__timestamp, [2006/06/28:13:29:51.491 MST]), (__source, [timer]), (pos, [5.0])}
{(__timestamp, [2006/06/28:13:29:53.492 MST]), (__source, [timer]), (pos, [3])}
{(__timestamp, [2006/06/28:13:29:53.991 MST]), (__source, [timer]), (pos, [2.5])}
{(status, [done]), (__timestamp, [2006/06/28:13:29:56.491 MST]), (__source, [timer]),
                   (reason, []), (configid, [timer1.000000000711.0000000000004])}

# All done, test successful!
atst> quit
all containers hae been shutdown
good bye
->
```

5.3.6 Customizing the Test Harness

While the command-line interface is the most convenient means of using the Test Harness, other approaches can be used to provide greater flexibility. In particularly, the Test Harness is available for programmatic control and is also designed for simple extension to add additional functionality.

5.3.6.1 Programmatic Control

The Test Harness manager class can be instantiated and used under the control of a custom Java application. The *javadoc* comments for the **`atst.cs.tools.testharness.TestHarness`** class provides details of the methods that are available for operating the test harness programmatically.

The following example simply duplicates the **`TimerController`** test performed through the command line interface that is shown above. Obviously more sophisticated tests can be created programmatically.

```
public class MyTestTimerController{
        public static void main (string[] args){
                // Build a Timer configuration
                IConfiguration timer1 = new Configuration("timer1");
                timer1.insert(new Attribute("timer.name", "timer1"));
                timer1.insert(new Attribute("timer.countdownlength", 10));
                timer1.insert(new Attribute("timer.alarm", new String[] {"2.5", "5"});
                timer1.insert(new Attribute("timer.eventrate", 3));

                // Start the test harness and perform the test
                TestHarness th = new TestHarness();
                th.loadController("timer", "atst.base.controllers.TimerController");
                th.init("timer", null);
                th.start("timer", null);
                th.listen("pos_timer1");
                th.listen("alarm_timer1");
                th.submit("timer", timer1));
                th.quitWhenDone(15);      // Give the timer time to run
        }
}
```

5.3.6.2 Extending the Test Harness

Another way to add functionality to the Test Harness is by extending some key classes found in the **`atst.cs.tools.testharness`** package. Developers are *encouraged* to examine *both* the javadocs *and* the source code for these classes to get a better understanding of what may be accomplished with this technique. This section presents a few highlights.

Customizing event handling

The class **`atst.cs.tools.testharness.EventListenerCallback`** is used by the Test Harness to listen and respond to events. The method **`doCallback(String eventName)`** may be overridden through subclassing to perform detailed analysis of events should that be desired. *If this method is overridden, however, it is **important** that the default functionality of returning the event's value be preserved!*

You can use this new subclass in place of the default **`EventListenerCallback`** class either by using the optional second argument to the **`listen`** command of the interactive interface or by using the second argumen to the **`listenFor`** method in the programmatic interface.

Other extensions

Other functionality can be added by directly subclassing the **`atst.cs.tools.testharness.TestHarness`** class. Any subclassing of this class that overrides existing methods should take care to *preserve the existing functionality*. For this reason, both the javadocs and the source code for **`TestHarness`** should be studied carefully before attempting to subclass **`TestHarness`**.

-- SteveWampler - 28 Jun 2006
to top

End of topic
Skip to action links | Back to top

- TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideAdminr1.49 - 09 Feb 2007 - 16:35 - SteveWamplertopic end

---

Start of topic | Skip to actions

---

6 Administration

6.1 Introduction

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

6.2 Configuration files

The common services makes use of some files to configure key aspects of the system. These files are created during the installation of the ATST software as decribed in the installation section below. They are all found in the **`$ATST/data`** directory. These files should not be edited directly as they are built from values supplied in the **`$ATST/admin/site.config`** file.

6.2.1 Database hosts

There are a number of databases used by the ATST common services. The services that need access to the persistent stores need to know the host for each database. There is a separate file for each database, containing the hostname. Those files are:

- **`$ATST/data/atst.archivedb.dbHost.txt`** -- host for the `archive` database.
- **`$ATST/data/atst.iddb.dbHost.txt`** -- host for the `id` database.
- **`$ATST/data/atst.logdb.dbHost.txt`** -- host for the `log` database.
- **`$ATST/data/atst.propertydb.dbHost.txt`** -- host for the `property` database.

Unless specified, the port used on each host is the default database port. You can change the port if your database is using a non-standard port by appending **`:PORT`** to the host name, e.g.: **`weaver.tuc.noao.edu:1234`**.

6.2.2 ICE hosts and services

The ICE communications middleware used by the common services needs to be able to identify the hosts of key ICE services. The files are:

- **`$ATST/data/atst.iceConnectionServiceHost.txt`** -- host and port for the connection service
- **`$ATST/data/atst.iceEventServiceHost.txt`** -- host and port for the event service
- **`$ATST/data/atst.iceIceBoxServiceHost.txt`** -- host and port for the IceBox service

In addition there are some configuration files that can be used to adjust the behavior of the ICE services:

- **`$ATST/data/iceAdmin.config`** -- configure the IcePack locator proxy
- **`$ATST/data/iceServices.config`** -- configure the various ICE services and tracing levels

For most situations the default settings in these configuration files are appropriate.

6.2.3 CORBA hosts and services

CORBA-based services are available as an alternative to ICE. There is no difference in application code when using CORBA services in place of ICE services. *The CORBA-based services are implemented only to a proof-of-concept*

level in this release of ATSTCS. In particular, there is no support implemented in the CORBA services for Container access. Completing the CORBA-based services has been deferred until a need arises.

The configuration files are:

- **`$ATST/data/atst.omniOrbConnectionServiceHost.txt`** -- host and port for the connection service
- **`$ATST/data/atst.omniOrbEventServiceHost.txt`** -- host and port for the event service

In addition there are some configuration files that can be used to adjust the behavior of the CORBA services:

- **`$ATST/data/omniORB.cfg`** -- configure the OmniORB name service
- **`$ATST/data/omniNotify.cfg`** -- configure the OmniORB notification (event) service
- **`$ATST/data/orb.properties`** -- common JacORB property settings
- **`$ATST/data/omniFunctions.zsh`** -- OmniORB configuration for shell scripts

For most situations the default settings in these configuration files are appropriate.

6.3 Installing, configuring, building, and starting

This section covers the process of bringing the ATST Common Services online at your site. It is ***strongly suggested*** that you read all parts of this section before performing any of the steps outlined here!

Installing the ATST Common Services involves the following steps:

6.3.1 Preparation

Many of the 3rd party software tools are provided with ATST Common Services and are setup during the installation process. Some required tools, however, are external to the ATST distribution:

- **`Linux`** with all the standard development tools, including **`gcc/g++`**. The preferred distribution is **`CentOS 4`** (equivalent to *Red Hat Enterprise 4*) and the binary files in the release have been built for CentOS. However, the source code is available (and, in most cases, included in the release) for rebuilding for other distributions. The ATST Common Services have been built and operated successfully under both *Red Hat 9* and *Fedora Core 3* in the past. No problems are anticipated building and using ATSTCS with other Linux distributions.
- **`zsh`** is used by the development support scripts
- **`python`** (version 2.4 or later) with **`psycopg2`** installed. [*Not required* unless you ask for Python support to be installed during configuration.]
- **`Gcc`** (version 4.1 or later, with g++ support). [*Not required* unless you ask for C++ support to be installed during configuration.]
- SUN **`Java JDK 1.5`** or **`Java JDK 1.6`**
- **`PostgreSQL`** (version 7.4 or later) appropriately configured and with a user **`atst`** able to create databases. Version 8.x is also known to work and is preferred. If you are building with C++ support, you ***must also install*** the PostgreSQL C++ library and development packages.

Most of these are straightforward, with any special details covered in the 3rd party software section below. PostgreSQL requires careful configuration, however. 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 for use with ATST, but you may also want to take advantage of the excellent on-line documentation at the PostgreSQL website.

6.3.2 Installing

The ATST Common Services are installed into an *ATST Software Development Tree* (**ASDT**).

6.3.2.1 Installing via CVS

Before an 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 checkout the entire ATST distribution before running the tools in the admin directory.

The ATST software is saved is a set of *separate* repositories to simplify access control:

- cs -- *common services* repository
- base -- *base support* repository
- ics -- *instrument control system* repository
- ocs -- *observatory control system* repository
- dhs -- *data handling system* repository
- tcs -- *telescope control system* respository
- ral -- *solar emphemeris support* repository (not publically available)

All ASDT require the code from the **`cs`** repository and most likely also **`base`**. Other repositories may or may not be needed, depending upon your particular project.

Assuming you want the ASDT to be rooted at (say) `/opt/atst`, based upon the `Panguitch-1P6` common services release, and you need the **`cs`**, **`base`**, **`ics`**, and **`tcs`** systems:

- Check out the *common services*. This builds the basic ASDT and populates with the common services:

```
cd /opt
export CVSROOT=":pserver:XXXX@maunder.tuc.noao.edu:/home/atst/src/cs"
cvs co -r Panguitch-1P6 atst
export ATST=/opt/atst
```

- Check out the *base suppport*:

```
cd ${ATST}/..
export CVSROOT=":pserver:XXXX@maunder.tuc.noao.edu:/home/atst/src/base"
cvs co -r Panguitch-1P6 atst/src/java/atst/base
```

- Check out the *instrument control system*:

```
cd ${ATST}/..
export CVSROOT=":pserver:XXXX@maunder.tuc.noao.edu:/home/atst/src/ics"
cvs co atst/src/java/atst/ics
```

- Check out the *telescope control system*. Note that three package must be checked out:

```
cd ${ATST}/..
export CVSROOT=":pserver:XXXX@maunder.tuc.noao.edu:/home/atst/src/tcs"
cvs co atst/src/java/atst/tcs atst/resources atst/screens
```

Once you have checked out the necessary systems, you can configure and build those systems in that ASDT.

6.3.2.2 Updating from CVS

Occasionally, a release may have emergency patches applied to particular files. (Normal bug fixes are applied to the *next* release.) When this happens and you have CVS access, you can update your ASDT with:

```
cd $ATST
cvs update -Pd
```

6.3.2.3 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 topic page:

If you have downloaded a tarball of the entire atst source tree as `atst.tar.gz`, you can install the ATST software with:

```
tar -zxopf atst.tar.gz
cd atst
export ATST=`pwd`
```

6.3.3 Configuring

6.3.3.1 A Quick and dirty approach to configuring

The following commands build an ASDT (provided you have checked out the files as shown above) and uses the createDevel command from the admin directory to finish the creation:

```
cd /opt/atst
export ATST=`pwd`
./admin/createDevel --make-all --tag=Panguitch-1P6
```

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 Configuration parameter section below gives information on the key parameters.

6.3.3.2 A better approach to configuring

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:

```
cd /opt/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.

6.3.3.3 Configuration 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`** The release of ATST that this ASDT is based upon (e.g. `Panguitch-1P6`). Be aware that if you leave this blank, 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_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`** This flag determines if the Java support will be built. Set to `yes` if you want this support, `no` (or leave blank) otherwise. It is *highly recommended* that you say **`yes`**, as you cannot initialize the databases without Java
  - **`ATST_JAVA_HOME`** The full path to Java (e.g. **`/opt/java6`**). Both JDK1.5 and JDK1.6 are known to work with ATST. Not needed unless **`USE_JAVA`** is set to **`yes`**.
  - **`ATST_JLIB_DIRS`** A colon-separated list of directories to search to satisfy (for example) JNI library references when using Java. Should be set to **`none`** if there are no such libraries. support *and* you will be missing a number of useful tools and applications. Not needed unless **`USE_JAVA`** is set to **`yes`**.
- **`USE_CPP`** This flag determines if the C++ support will be built. Set to `yes` if you want this support, `no` (or leave blank) otherwise. Currently, there is no C++ support, so say **`no`**.
  - **`ATST_CPP_CPP_BIN`** The path to the version of the g++ compiler to use when building the C++ support. Not needed unless **`USE_C++`** is set to **`yes`**.
  - **`ATST_CLIB_DIRS`** A colon-separated list of directories to search to satisfy external references when using C++. Should be set to **`none`** if there are no such libraries. Not needed unless **`USE_C++`** is set to **`yes`**.
- **`USE_PYTHON`** This flag determines if the (rudimentary) Python support will be built. Set to `yes` if you want this support, `no` (or leave blank) otherwise.
  - **`ATST_PYTHON_HOME`** The full path to where you have *Python 2.4* installed (e.g. `/usr/local`). Not needed unless **`USE_PYTHON`** is set to **`yes`**.
  - **`ATST_PLIB_DIRS`** A colon-separated list of directories to search to satisfy Python references. Should be set to **`none`** if there are no such libraries. Not needed unless **`USE_PYTHON`** is set to **`yes`**.
- **`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`).
- **`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_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_CORBA** This flag determines if CORBA support will be built. Set to yes if you want CORBA, no (or leave blank) otherwise.
    - ♦ **ATST_OMNI_CONNECTION_HOST** The machine that is to run the CORBA name service. (e.g. maunder.tuc.noao.edu). This must *exactly* match the value returned by hostname on that machine.
    - ♦ **ATST_OMNI_CONNECTION_PORT** Port on that machine to be used by the CORBA name service (e.g. 5555)
    - ♦ **ATST_OMNI_EVENTS_HOST** The machine that is to run the CORBA event service. (e.g. maunder.tuc.noao.edu). This must *exactly* match the value returned by hostname on that machine.
    - ♦ **ATST_OMNI_EVENTS_PORT** Port on that machine to be used by the CORBA name event service (e.g. 6666)

6.3.4 Building

6.3.4.1 Building Java (and Python) support

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

```
cd $ATST
make build_all
```

which build the Java support and (if asked for in the **site.config** file) the Python support.

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

```
cd $ATST
make class_clean
make build_all
```

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

- ice - produce Java source files from any Ice source files in the **$ATST/src/slice** directory. The generated Java source files are placed into subdirectories (depending on the package names) of **$ATST/lib/Java/generated_java**
- ice_clean - remove the generated Java files produced from the ice target.
- corba - produce Java source files from any IDL source files in the **$ATST/src/idl** directory. The

generated Java source files are placed into subdirectories (depending on the package names) of **`$ATST/lib/Java/generated_java`**

- `corba_clean` - remove the generated Java files produced from the `corba` target.
- `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`
- `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 all of the above actions in sequence.

6.3.4.2 Building C++ support

After the Java support has been built, you may then build the C++ support if that support has been asked for in the **`site.config`** file with:

```
make gcc_gen    # Expect this first make to fail with errors
make gcc_gen    # There should be no errors here
make gcc_all >make.out 2>&1
```

Because having Gcc4 incorrectly built can result in errors, the file **`make.out`** produced by the last command above should be carefully checked. (See the notes on correctly building Gcc4.)

6.3.5 Starting the ATST services

6.3.5.1 Starting the base services running

The ATST common services need to have both **`PostgreSQL`** and either the **`ICE`** or **`CORBA`** services running.

You 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 haven't set this up!

6.3.5.1.1 PostgreSQL

PostgreSQL is normally started on system boot and needs to be started on each machine identified as a database host by the configuration files discussed earlier. The Linux release should provide a means to start PostgreSQL on system boot. Under Red Hat systems, this can be done with (as superuser):

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

Similarly, the current state of PostgreSQL can be checked on Red Hat systems with:

```
# service postgresql status
```

and, if PostgreSQL is not running, it can be started immediately with:

```
# service postgresql start
```

6.3.5.1.2 ICE

The ICE naming (*IcePack*) and event (*IceStorm*) services need to be started with:

```
$ATST/bin/startIceServices
```

A similar command shuts those ICE services down:

```
$ATST/bin/stopIceServices
```

6.3.5.1.3 CORBA

The CORBA support is not fully implemented at this time.

The CORBA naming and event services may be started with:

```
$ATST/bin/startOmniServices
```

and stopped with:

```
$ATST/bin/stopOmniServices
```

6.3.5.1.4 Building the ATST databases and tables

Before you can use the ATST for the first time, you must create and populate the databases used by ATST. ***This is typically a one-time operation - each time you perform this step you may lose any information that has been saved in the databases previously.***

```
dropdb -h PROPERTY_DB_HOST -U atst atst.propertydb
cd $ATST
./admin/createDevel --init-db
```

Here, **`PROPERTY_DB_HOST`** is the hostname of the machine running the Property Service database.

N.B. make sure you have executed **`startIceServices`** before running the above command

6.4 Tests

The last step of building ATST common services 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 are ready for use.

6.4.1 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.

6.4.2 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**.

Decide whether the ICE-based or CORBA-based (OmniORB) services are to be tested. The example below assumes ICE, but replacing the `--ice` option with `--omni` would run the tests using CORBA.

On **machineA**, start a subscriber running:

```
$ATST/bin/ajava atst.cs.services.event.EventSubscriber --ice
```

On **machineB**, start the publisher:

```
$ATST/bin/ajava atst.cs.services.event.EventPoster --ice
```

The publisher publishes an event containing a simple counter and a timestamp (expressed in milliseconds since the *epoch*) at a 1Hz rate. The subscriber simply prints the events that it receives.

6.5 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, such as monitoring of disk usage, cpu loads, database resources, etc.

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

6.6 Troubleshooting and problem reporting

6.6.1 Troubleshooting

ATST common services 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 - they will be added over time and documented here as they become available.

6.6.2 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/atsthelp`** and request an account now to avoid delays when you do uncover a problem.

6.7 Known problems

6.7.1 General problems

- There is no support for *wildcards* in event subscriptions yet.
- The Java and C++ build processes are not fully integrated. (Java and Python are integrated.)

6.7.2 Known problems in the Java support

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

- At the current time, calling **java.lang.System.exit()** from a component will exit that component. *It will also exit all other components in the current container and the container itself.* It's probably not a good idea to call System.exit(), then.

- 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 won't 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.

- Some service tools must be loaded into each component's namespace, others can be shared. If a service tool makes use of the 'static' service classes (**Log**, **Property**, **Event**, etc.), then it must be loaded into the component's namespace. The class **AbstractToolBoxLoader** provides all ToolBoxLoaders with a method **void loadInComponentNamespace(String className)** that loads a class in the current component's namespace.

- 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 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.

6.7.3 Known problems in the C++ support

The C++ support is not usable yet.

6.7.4 Known problems in the Python support

TBA

6.8 3rd party software

6.8.1 Notes on using Java

Both SUN JDK1.5 and JDK1.6 are known to work with ATST. This section describes one method of obtaining and installing JDK1.6 for use with ATST.

6.8.1.1 Getting Java

You can download JDK1.6 from SUN's Java web site. The Standard Edition is assumed and used here. The *self-extracting binary* is also assumed.

6.8.1.2 Installing Java

After downloading **`jdk-6-linux-i586.bin`** into your home directory, the following steps may be used to install JDK1.6:

```
mkdir -p /opt/java
cd /opt/java
sh ~/jdk-6-linux-i586.bin
ln -s /opt/java/jdk1.6.0 /opt/java6
```

6.8.1.3 Configuring ATST to use Java

After performing these steps, there are three configuration parameters in **`$ATST/admin/site.config`** that must be set:

```
# Include Java support (recommended, as you can't do:
#    $ATST/admin/createDevel --init-db
# without it!)
USE_JAVA=yes
# Location of java: ATST currently uses jdk 1.5.
ATST_JAVA_HOME=/opt/java6
# Any extra library directories needed to support Java (: separated)
ATST_JLIB_DIRS=none
```

Once these have been set, the Java support for ATST Common Services builds successfully.

6.8.2 Notes on using Gcc

The C++ support in ATST Common Services depends upon both using version 4 of Gcc (specifically g++ support) and the corresponding version of the *C++ Standard Template Library*. The CentOS4 release does not include a suitable version of Gcc, but it straightforward to obtain and install the correct version. The approach described here is known to work with CentOS4 and automatically provides the correct STL version.

6.8.2.1 Getting Gcc4

Gcc can be obtained from the main GCC web site. I downloaded the latest (4.1.1) **`gcc-core`** and **`gcc-g++`** packages (no other packages are required as the **`gcc-g++`** package includes **`libstdc++`**).

6.8.2.2 Building and installing Gcc4

The steps here can be used to configure and build Gcc4, installing the result into **`/opt/gcc4`**:

```
mkdir /opt/gcc4      # Where I'm installing all of gcc4
mkdir ./gcc4         # Where I'm putting the source and build area
mkdir ./gcc4/build   # The build area
cd ./gcc4
tar -jxpf ../gcc-core*
tar -jxpf ../gcc-g++*
cd build
../gcc-4.1.1/configure --prefix=/opt/gcc4
make CFLAGS='-O' LIBCFLAGS='-g -O2' \
   LIBCXXFLAGS='-g -O2 -fno-implicit-templates' bootstrap
make install
```

6.8.2.3 Configuring ATST to use Gcc4

There is one configuration variable in **`$ATST/admin/site.config`** that *must* be set:

```
# If using C++ identify the correct C++ compiler (Requires g++4)
ATST_CPP_CPP_BIN=/opt/gcc4/bin/g++
```

and, because the above build installed Gcc4 in a non-standard location, you have to identify where the new Gcc libraries are located in the same configuration file:

```
# Any extra library directories needed to support C++ (: separated)
#   Use "none" (without quotes) to indicate no extra libraries needed
ATST_CLIB_DIRS=/opt/gcc4/lib/
```

With these changes, the build of C++ support in ATST Common Services can complete without difficulty.

6.8.3 Notes on using Python

The python support in ATST Common Services relies upon using Python 2.4 along with the **`psycopg2`** support package. This sections describes one method of installing those tools onto CentOS4 for use with ATST.

6.8.3.1 Getting Python2.4 and psycopg2

You can download the source for Python2.4 from the main Python web site. Similarly, the source for psycopg2 is available ftom the initd.org web site.

6.8.3.2 Building and installing python

The following steps configure, build, and install python and psycopg2 appropriately for use with ATST Common Services:

```
mkdir /opt/python2.4    # Where we're installing python 2.4
mkdir ./python          # Where we're building python 2.4
cd ./python
tar -jxpf ../Python-2.4.4.tar.bz2
tar -zxpf ../psycopg2-2.0.5.1.tar.gz
cd Python-2.4.4
./configure --prefix=/opt/python2.4 --with-thread-safety
make
make install
cd ../psycopg2-2.0.5.1
/opt/python2.4/bin/python setup.py build
/opt/python2.4/bin/python setup.py install
```

6.8.3.3 Configuring ATST to use python

Based on the above installation, there are three configuration parameters in **`$ATST/admin/site.config`** that must be set:

```
# Include (rudimentary, for now) Python support?  (Requires python = 2.4)
USE_PYTHON=yes
# If using python, identify the location of version 2.4.
ATST_PYTHON_HOME=/opt/python2.4/
# Any extra library directories needed to support Python (: separated)
ATST_PLIB_DIRS=none
```

With these changes, the build of Python++ support in ATST Common Services can complete without difficulty.

6.8.4 Notes on preparing PostgreSQL for ATST

This section gives some hints and suggestions for configuring PostgreSQL for use with ATST Common Services. 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.

6.8.4.1Initial configuration steps (PostgreSQL 7.4)

This assumes PostgreSQL has been installed on your computer. Typically, this means there is a directory **/var/lib/pgsql/data** that came with your distribution release. If you have installed PostgreSQL from another source (i.e., downloaded the tar file), this directory may be in a different place. There are two files in this directory that need to be modified.

- **postgresql.conf** - this file contains most of the major configuration options for PostgreSQL. For development work, you can probably get away with one required change (for ATST production work, there are a number of performance tuning parameters that must be adjusted; these are not covered here). Uncomment the line defining **tcpip_socket=false** and change it to **true**. There are also a few parameters you might *want* to change: Uncomment and change the define for **max_connections** to at least 64; and uncomment and set to true **log_timestamp**.

- **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, at the ATST software groups **pg_hba.conf** file ends in the following lines:

```
local  all    all            trust
```

```
host   all    all   127.0.0.1 255.255.255.255 trust
host   all    all   140.252.0.0 255.255.0.0   trust
```

These are very loose settings, a PostgreSQL expert could tighten these up.

6.8.4.2 Initial configuration steps (PostgreSQL 8.1)

Currently this PostgreSQL is not found in RedHat distributions (if you are using CentOS4, it is now available from the **centosplus** repository). If you download the *source* tar or rpm file and perform the standard installation instructions, the files that need to be modified are in directory **/usr/local/pgsql/data**. If you download the *binary* rpm file, either from the **centosplus** repository or **www.postgresql.org**, the files are in the directory **/var/lib/pgsql/data**.

- **postgresql.conf** - this file contains most of the major configuration options for PostgreSQL. Change the **listen_addresses** value to **'*'**.

- **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, at the ATST software groups **pg_hba.conf** file ends in the following lines:

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

These are very loose settings, a PostgreSQL expert could tighten these up.

6.8.4.3 Installing PostgreSQL C++ support

Under CentOS4, you need to install two RPM files:

```
http://dag.wieers.com/rpm/packages/libpqxx/libpqxx-2.6.7-2.el4.rf.i386.rpm
http://dag.wieers.com/rpm/packages/libpqxx/libpqxx-devel-2.6.7-2.el4.rf.i386.rpm
```

To install on a stock CentOS4 machine (as root):

```
rpm -Uvh libpqxx-2.6.7-2.el4.rf.i386.rpm libpqxx-devel-2.6.7-2.el4.rf.i386.rpm
```

If you have upgraded PostgreSQL to one of the 8.x releases, the above command will report a dependency failure. Use:

```
rpm -Uvh --nodeps libpqxx-2.6.7-2.el4.rf.i386.rpm \
                  libpqxx-devel-2.6.7-2.el4.rf.i386.rpm
```

For other Linux distributions you may have to build the libpqxx support from source. If **http://rpmfind.org/** doesn't turn up suitable RPMs, obtain the source from the libpqxx web site and install.

6.8.4.4 Starting PostgreSQL

After making these changes, you will need to restart PostgreSQL. Under RedHat Linux, this can be done with:

```
service postgresql restart
```

6.8.4.5 Final PostgreSQL setup steps

Once you have PostgreSQL configured and running, you need to make sure that ATST can access the database server and create new databases:

- Create a database user **`atst`** and allow that user to create databases and add users (*which gives **`atst`** other special privileges needed for database creation*):

  ```
  createuser -h DB_HOST_NAME --createdb --adduser atst
  ```

  Substitude the hostname of the machine you have the PostgreSQL database server running on for **`DB_HOST_NAME`**. (If there is more than one database server, you should run the above command once for each of those servers.)

Later, *once the Common Services are up and running*, you may want restrict the capabilities of this **`atst`** database user:

- Remove 'database superuser' privileges for user **`atst`**, but allow that user to create and remove databases:

  ```
  psql -h DB_HOST_NAME
     ALTER USER atst NOCREATEUSER;
     \q
  ```

-- SteveWampler - 03 Mar 2005
to top

End of topic
Skip to action links | Back to top

- News
- Design
- Common Services
- Application base
- OCS
- DHS
- TCS
- ICS
- ICDs
- Docs
- Meeting Notes
- Problem Tracking
- Changes

- **Misc**
  - Users
  - Groups
  - Offices
  - Topic list
  - Search

- **TWiki Webs**
  - Know
  - Main
  - Sandbox
  - TWiki

Create personal sidebar

- **Edit**

- Attach
- Printable
- PDF

Main.AtstCsGuideTreer1.21 - 18 Jan 2007 - 15:42 - SteveWamplertopic end

Start of topic | Skip to actions

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 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.

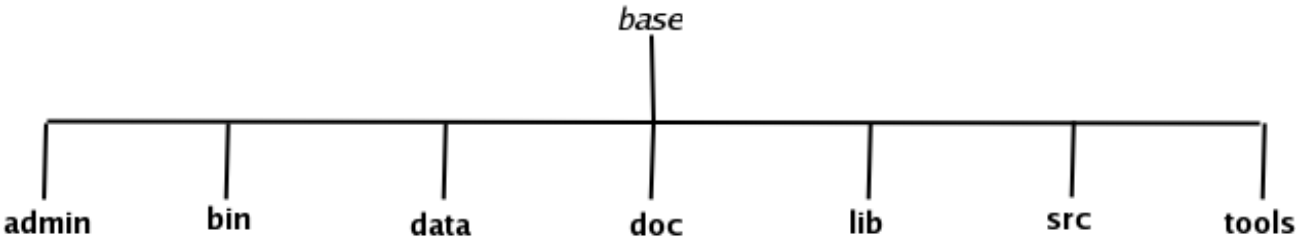
Generally speaking, the specific release identification is only of interest when an ASDT is first created - development within that ASDT can be thought of as *extending* that specific release. Occasionally, extensions on one release may be *merged* into other (possibly new) releases as well.

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:



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. ATST applications are applications that are part of the ATST software system itself and are typically built from sources found elsewhere in the ASDT. As a general rule, ATST applications are found in subdirectories of bin corresponding to the subsystem associated with that application. For example, applications that are part of the Observatory Control System are found in `bin/ocs`.

In addition to the subdirectories mentioned above, bin also includes subdirectories for holding architecture dependent applications. For example, applications built to run under Linux on Intel x86 architectures can be found in `bin/Linux_i386`. These applications are typically not referenced directly but rather are accessed through scripts that automatically determine the current architecture and invoke the proper version of the application for that architecture. So, for example, to access any ATST Common Services application it is sufficient to add `$ATST/bin/cs` to your PATH without regard to the architecture you are running on. (This is particularly useful in a distributed environment as application managers do not have to concern themselves with identifying the architecture of a machine on which an application is to be run.)

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, found in system-specific subdirectories of `doc/guides`
- operations manuals, found in system-specific subdirectories of `doc/manuals`
- source code documentation, found in system-specific subdirectories of `doc/api-docs`.

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:

- `include` holds all C++/C include files, organized into system specific subdirectories
- `slice` holds all the SLICE source files.
- `idl` holds all the CORBA IDL 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
- `python/atst` holds the Python source files of all ATST systems available in this ASDT

The `src` directory and subdirectories of `src/java`, `src/c++`, `src/python`, `src/idl`, `src/slice`, and `src/include` 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 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.

7.4.1 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 permited 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.

7.4.2 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.

7.4.3 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.

7.4.4 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 `$ATST/bin/ajava` can be used to directly execute java classes from a build ASDT. The `ajava` command invokes the proper java runtime with the appropriate parameters for the ASDT rooted at `$ATST`. There is no need to set a `classpath` or any environmental parameter other than `$ATST`.

7.4.5 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.

7.4.6 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.

7.4.6.1 Ssh files

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 passwordless 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.)

7.4.6.2 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
  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
  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.

-- SteveWampler - 7 Nov 2005
to top

End of topic
Skip to action links | Back to top

Edit | Attach image or document | Printable version | Raw text | More topic actions
Revisions: | r1.21 | ≥ | r1.20 | ≥ | r1.19 | Total page history | Backlinks
Main.AtstCsGuideTree moved from Main.AtstDevelTree on 03 Feb 2006 - 21:45 by SteveWampler - put it back
You are here: Main > AtstCommonServices > AtstCsSDD > AtstCsGuide > AtstCsGuideTree

to top