NGAO Software Architecture
KAON 672: Connection Service

Author	Modified	Notes
Douglas Morrison	6/3/09	Details the Keck Common Services Framework Connection Service

1.0 Introduction

Distributed systems such as NGAO consist of a number of physically remote software objects that run on different CPUs, and within different address spaces. The communication middleware is responsible for opening a connection between objects and sending data from the client (the local object) to the server (remote commandable object), and potentially any return or status data back to the client. A key design goal of the NGAO system is to hide the details of the underlying communication middleware and present the developer with a simple interface to find, connect, and command remote components. Through the Keck Common Services Framework (KCSF) a middleware abstraction will be provided that will effectively hide the underlying communication infrastructure and offer a simple set of services and classes to performed distributed computing.

This document will discuss the design of the Connection Service and its related utilities. Additionally, communication middleware specific details will be discussed to note key implementation requirements, as well as show how the concept and use of the Connection Service from the developer's perspective remains constant, regardless of the underlying communication mechanism.

2.0 Overview

The KCSF provides developers with a suite of tools and services that simplify a wide range of typical software tasks. One of these tools, the Connection Service, provides a mechanism to deploy and connect to distributed components across the network. *Deploying* a component is the process of making a running server-side object known to the world. *Connecting* to a component is the process of finding a remote object and making a connection. The Connection Service provides a simple interface to perform these tasks through the *bind* and *connect* methods, respectively.

The *bind* method associates a unique name to a component instance to act as a global address for the object. Although not every server object needs to be addressable, every published component name must map to one and only one component instance. The *connect* method is used by the client to find and open a connection to a component identified by the unique name.

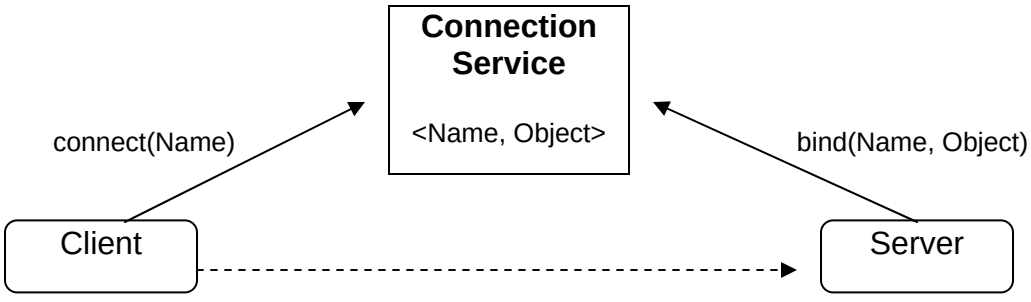


Figure 1: Connection Service Overview

In order to hide the details of the underlying communication middleware, abstract object wrappers are used to bridge the application layer with the communication protocol. Wrappers are separated

into two main types: proxies and stubs. For each proxy implementation there is a corresponding stub implementation, and vice versa. Proxies and stubs exposes an interface identical to the component they wrap, and are responsible for translating the data that is sent across the wire.

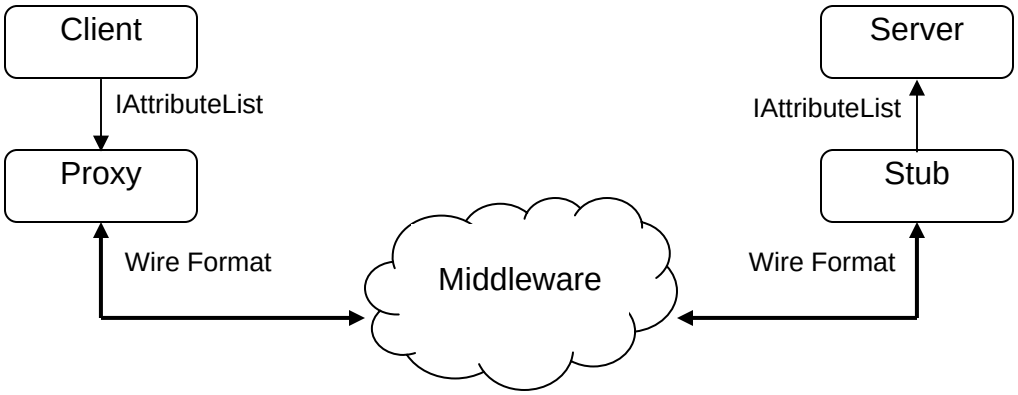


Figure 2: Proxies and Stubs

On the client side, a proxy is used to represent the existence of a remote object and maintain an open connection the component. Since the proxy exposes an interface identical to the actual remote object instance the client can treat it as a local object, without knowledge of the communication protocol or concern for its location. Any data that is passed during a call to one of the proxy's methods is translated to the format required by the communication protocol and sent on the wire. The proxy may then wait for a response from the remote component, and dispatch any callbacks events that occur.

On the server side, a component stub is used to delegate incoming communication requests to the component implementation. The stub performs a mirror opposite form of data translation as from that of the proxy, by converting the data received on the wire to the original format expected by the component middleware. The corresponding method on the wrapped component is then executed, passing the data to the instance. If necessary the stub will wait to receive a result status from the component and issue a callback event to the client proxy.

To aid in the creation of proxies and server stubs a set of object factories have been developed known as Connectors. The Connection Service utilizes connectors with the bind and connect operations to determine the type of proxy and stub to incarnate.

3.0 Connectors

Connectors are used by the Connection Service to determine the appropriate stub and proxy to use for binding and connecting to components. Essentially, connectors are a conditional proxy / stub factory, where the successful creation of a wrapper is dependent on the type of component. As with proxies and stubs, there is a corresponding connector implementation for each class of object. Specifically, the following connectors will be implemented,

- IRemoteConnector
- IComponentConnector
- IControllerConnector
- IContainerConnector
- IContainerManagerConnector

The actual proxies / stubs created by a connector are related to the class type of the connector. For example, a Controller Connector will produce Controller proxies and stubs, only. Connectors expose a very simple interface for creating proxies and stubs.

```
class IConnector {
      Stub bind(ILocal object);
      Proxy connect(IRemote object);
}
```

In the above pseudo-code a *bind* and *connect* method are provided that accept an instance of an ILocal and IRemote base object. All server-side component implementations inherit from the *ILocal* base class. All client-side proxy implementations inherit from the *IRemote* base class, which is itself a subclass of ILocal. Both interfaces represent a component level software object - communication middleware agnostic. The connector is responsible for generating a communication middleware specific wrapper that is capable of accepting and delegating commands between the transport and component.

However, not just any wrapper can be created - it has to be specific to the actual class of component passed to the connector. To this end, the connector will test the ILocal / IRemote reference to see if it is an instance of the class type implemented by the connector. If it is, a corresponding proxy / stub is created for the class type and returned to the connection service. If the object reference is not an instance of the connector's class type, a null object is returned.

The advantage of using connectors is that proxy and stub creation logic can be removed from the Connection Service and contained within simple object factories: this continues the inversion of control principal promoted throughout the framework. Since connector types are built in an increasing complexity class hierarchy that mirrors the class hierarchy of components (Remote ← Component ← Controller, etc.), the creation of stubs and proxies can be done by simply iterating through a list of connectors and attempting to bind / connect a component until a valid wrapper is returned. Connectors would be arranged in the sequence from most complex to least (Container Manager Connector → Local Connector). The first valid wrapper returned by a connector represents the highest level type of the component. The Connection Service will use the wrapper returned by the connector to bridge the component and communication middleware layers.

4.0 Proxies

Proxies are the client-side interface to distributed objects, and implement an identical set of methods and class signature as the component they represent. Proxies are designed to give the client the sense that they are directly operating on a local object. Any data marshaling and formatting required by the communication middleware is handled by the proxy opaquely: the client doesn’t need to be aware of how the data is transported.

There are five types of proxies representing each of the main base classes of objects that can be developed with the Keck Common Services Framework:

- IRemoteProxy
- IComponentProxy
- IControllerProxy
- IContainerProxy

- IContainerManagerProxy

Each of these proxies implements the client-side interface for the IRemote, IComponent, IController, IContainer, and IContainerManager classes, respectively. Proxies may also implement additional methods outside of those defined by their associated component. For example, proxies inheriting the IRemote interface will implement asynchronous version of the standard get and set methods.

```
IAttributeList get(IAttributeList);       // Synchronous implementation
boolean get(IAttributeList, ICallback)    // Asynchronous implementation

IAttributeList set(IAttributeList);       // Synchronous implementation
boolean set(IAttributeList, ICallback)    // Asynchronous implementation
```

The asynchronous version accepts an ICallback object and returns a boolean value. The return value indicates whether or not the get command could be dispatched to the communication middleware. This call should return immediately, and will fail only if there are network issues. The callback object will receive the response from the server when the task has been completed. This allows the client to receive and handle event status without blocking the issuing thread. (See the Command-Response documentation for more information.)

Proxy objects are generated and returned to the client during the call to the Connection Service's connect method. This method will find and connect to the remote object specified by the component name, and then attempt to determine the appropriate proxy to create by iterating through the various connectors. When the respective connector has been found the proxy will be generated and returned to the client. The proxy will remain valid for the lifetime of the remote object or until the network connection is broken or client application is shutdown. Depending on the capabilities of the communication middleware, a variety of fault recovery and operation failure techniques may be implemented into the proxies. This can include automatic reconnection, server redundancy and failover, and invocation retry. Additionally, proxies may implement middleware specific efficiency and quality of service strategies to improve throughput and bandwidth utilization, including batching and one-way invocations.

5.0 Stubs

Stubs implement the bridge between the communication middleware and the server components. As with proxies, stubs implement an identical interface to the component they represent. Stubs however, work as the inverse to proxies by forwarding data and commands from the communication middleware to the components. Stubs are essentially a wrapper for component implementations to bridge the communication specific environment with the abstract KCSF middleware. Components implemented in KCSF are completely unaware of how their methods are invoked, and do not need to take any specific action to format data for one stub implementation over another. The logic to transform and send data is handled entirely by the stub.

There are five classes of stubs:

- ILocalStub
- IComponentStub
- IControllerStub
- IContainerStub
- IContainerManagerStub

Each of these stubs implements the server-side bridge between the middleware and the ILocal, IComponent, IController, IContainer, and IContainerManager classes. Stub implementations react to events from the communication middleware by processing and pushing data to components. The stub will receive information from the client proxy to determine the operation to be executed in the component, and invoke the corresponding method.

When the operation has completed, any data returned by the component method call will be transformed and returned to the invoking client. From the perspective of the component, all calls appear to be performed by local clients. Data is processed and returned as usual, without required knowledge of whether the client is local or remote.

6.0 Connection Service

The Connection Service provides developers with the tools to deploy and connect to remote components, and is available to both servers and clients. A connection service instance is typically created by a container and shared between each of its components. As such there is at most one connection service per process. Although multiple connection service implementations could be used within a single project, there will typically only be one.

The connection service exposes a simple interface to developers, and hides the middleware specific details from the users. The actual process of finding and deploying components, opening network connections, and creating stubs and proxies is all performed as a black-box. Clients and servers only need to hold a reference to a connection service implementation to have access to these capabilities.

```
classs IConnectionServiceTool {
      /**
       * Makes a component available to the network.
       */
      void bind(IToolBoxAdmin, String name, ILocal object);

      /**
       * Removes a component from the network.
       */
      void unbind(IToolBoxAdmin, String name);

      /**
       * Connects to a remote object and returns a proxy.
       */
      IRemote connect(IToolBoxAdmin, String target);

      /**
       * Disconnects a proxy from a remote object.
       */
      void disconnect(IToolBoxAdmin, IRemote proxy);

      /**
       * Returns a list of all bound objects of the specified type.
       */
      String[] allRegistered(IToolBoxAdmin, String Type);
}
```

All connection service implementations inherit from this base class. The connection service's two main methods are *bind* and *connect*. The bind method accepts an instance of a component object and its unique name. The bind will create an appropriate stub for the component, publish the existence of the object over the network, and map the name to the object reference. The connect method accepts a string representing the unique name of a bound object. A corresponding proxy will be determined for the remote object and returned to the client.

7.0 ICE Implementation

The ICE middleware distributed design closely resembles the proxy-stub pattern used by the KCSF. An Interface Definition Language (IDL) skeleton is created and used to generate client and server side proxies and stubs. As with the KCSF proxies, ICE proxies can be treated as local instances of remote objects. ICE stubs essentially define a class skeleton for the developer to define the specific functionality of the server. A stub implementation in ICE is known as a servant. The ICE middleware is responsible for transmitting data between the proxies and dispatchers.

In order for ICE objects to find and open connections to remote objects, the location endpoints need to be defined and known by clients. Endpoints essentially boil down to an address and port for a machine on a network. Part of the incarnation process of server objects is to define the endpoint(s) where it will listen for remote invocations by clients. There are many ways to do this, but the most common is to create an ICE object adapter (which bridges the ICE run-time with the application), and hang ICE objects from it. The adapter is responsible for delegating calls between the run-time and ICE servants, and vice-versa. With this technique a single adapter (endpoint) can be used to communicate with ICE servants within an application. The address assigned to the adapter can be provided to clients to open network connections between proxies and servants. Although this nicely ties servants to endpoints for each application, it does not solve how the clients will be informed of the address. *IceGrid* provides a simple and effective solution to publishing endpoint information for servants.

7.1 IceGrid

IceGrid is a common tool provided by the ICE middleware that implements a simple object location and connection service. An ICE location service is a network application that maps servant endpoints to unique names. Clients can connect to IceGrid and request an object reference to a servant based on a user friendly name (e.g. "Camera1"). IceGrid will lookup the endpoint information for the associated name, retrieve type and object information from the servant, and return an ICE proxy to the client. After the initial connection, the client will communicate directly with the server – IceGrid is not involved in any further interaction between the components.

The advantage of using IceGrid is two fold:

- Servants can be moved to new endpoints without impacting clients.
- Clients only need to know the unique name of an object and the address of the IceGrid application to make remote connections.

Since the name of a component is likely to never change during the lifetime of a project, clients can expect servant names to persist. Developers can explicitly hardcode the name of a remote component into a client and be confident that IceGrid will be able to find the reference, even if the host for the servant has changed. Since only a single IceGrid application is needed for a project a dedicate server

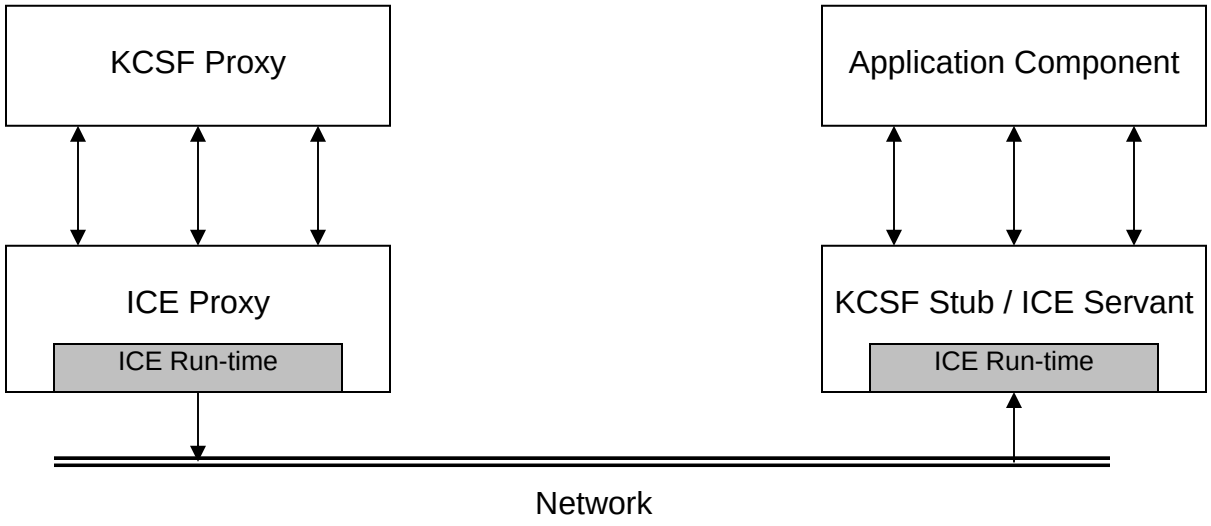
can be defined to host the service. Clients only need to be configured with the endpoint information for IceGrid to find and make connections to distributed applications. The Ice Connection Service utilizes IceGrid to perform object location retrieval and binding.

7.2 ICE Component Stubs

KCSF components and their subclasses do not directly implement or know about the underlying communication middleware. It is the stub implementations discussed earlier that bridge the application and transport layers. The KCSF stubs developed for ICE are responsible for wrapping a component implementation in an ICE servant. When the servant/stub is invoked by the ICE run-time it will perform the necessary operations to format the incoming data and pass it on to the component. Specifically, for the get/set/execute methods of a component, this involves converting the ICE wire representation of the data (a dictionary of String-AttributeValue objects) into an IAttributeList. The data returned by the component is converted back into the ICE wire format, and sent across the network to the client.

7.3 ICE Component Proxies

Since KCSF components are not themselves ICE servants, the ICE proxies do not actually communicate directly with them. The ICE proxies are actually proxy implementations for the ICE stubs that wrap components. When a method on a proxy is invoked, the KCSF proxy will convert any data to the ICE wire format, and invoke the associated method on the actual ICE proxy. The ICE run-time will transport the data to the ICE servant (the KCSF stub) which will handle the delegation to the component.



8.0 DDS Implementation