HPCERC2001–001
8 February 2001

UESA Project Architecture
UHF Electronic Scanning Array

Authors

Dennis Lucero, Rahul Kulkami,
Vinod Ramdas, and Robert Ballance

The University of New Mexico
Albuquerque High Performance Computing Center

HPCERC TECHNICAL REPORT

Technical Report on the UESA Architecture

January 2001

Authors:

Dennis Lucero	strider@ahpcc.unm.edu
Rahul Kulkarni	rahulak@ahpcc.unm.edu
Vinod Ramdas	alomaram@ahpcc.unm.edu
Dr. Robert Ballance	ballance@ahpcc.unm.edu

Abstract

This report describes the UESA project architecture. UESA is the UHF Electronic Scanning Array. Our customer is the United States Navy, specifically the office of Naval Research (ONR) and the Navy research Laboratory (NRL). The goal of the project is to facilitate a user who wants to view naval radar data. To accomplish this the system will take data from radar sensors and make it available for processing and viewing on multiple computing environments. The system must operate in a dynamic environment, be an open system, deal with varying production rates, perform filtering and processing, and handle bandwidth management issues. The architecture accomplishes these goals using Java, Jini, XML, uniform data transport, a Portal program, and Web technologies.

Table of Contents

Overview

Introduction

The project vision is to facilitate a user who wants to view radar data. To accomplish this the user would simply have to start a display application and select the type of data they were interested in from a menu of available data streams. The display application would go out and find the desired type of data, retrieve it, and display it. The user could decide what type of data is wanted, filter the data, and have it preprocessed to their liking before receiving it. In this same way the radar data will be made available to computers at any location. Any computer at any location that has this display application installed on it could retrieve and display radar data without needing any prior configuration with the sources of data. In short the system will take data from radar sensors and make it available for processing and viewing on multiple computing environments.

Problems and Issues to be solved

Dynamic environment

The data producing and data displaying programs must be able to locate and use each other without human intervention. This must work if they are located near each other (locally) or they are separated by large distances (remotely). When data producing or data displaying programs start they must make themselves available on the network automatically. When data producing or data displaying programs shutdown they must remove themselves from the network automatically.

Open system – new kinds of sensors, devices, data streams, etc

The types of sensors will likely change over time, as will the types of data produced by these sensors. Because of this the format and content of the data streams can be expected to be changing over time. The system must be able to function properly with these new types of devices, data, and data streams. The system should not only be able to function with these changes but must do so without needing changes made to the system itself. In this way the system can continue to run effectively without recompilation even though the data, sensors types and data streams may change.

Production rates, buffering

The data producing programs and data receiving programs may not produce or receive data at the same rate. The data produced by the data-producing program should be saved (buffered) for some period of time in case a data-receiving program becomes available that needs it. The data-producing program should not be encumbered with this saving (buffering) operation and should not need to be aware of the number of data receiving programs available.

Filtering and processing

The data must be available for filtering and processing regardless of its point of origin or its destination. The user must be able to filter the data so as to only get the desired information. The user must be able to process the data in any way available (compression, formatting, etc) before receiving it for display or as a goal in itself.

Bandwidth Management

The data should be sent as pure as possible. This means that information needed to utilize the data should not be sent with every data packet. The information needed to utilize the data should be sent only once, at the beginning of transmission. In this way the "pure" data is all that must be sent, reducing the bandwidth overhead incurred by using the system.

Architecture

Solution Techniques

Java

Java facilitates the transmission of classes. Classes needed at runtime can be retrieved using Java web technologies such as serialization, class loader, and reflection.

Serialization provides the ability to store and retrieve Java objects.

A class loader is an object that is responsible for loading classes. It loads remote classes. Given the name of a class, it should attempt to locate or generate data that constitutes a definition for the class.

Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions.

Jini

Jini is a set of specifications that enables services to discover each other on a network and that provides a framework that allows those services to participate in certain types of operations. Jini allows applications to interact; it allows this interaction to happen in a dynamic, robust way. There are three basic features of this interaction.[1]

The Jini environment requires no user intervention when services are brought on or offline.[1]

The Jini community is self-healing: it can adapt when services, and consumers, come and go.[1]

Consumers of Jini services do not need prior knowledge of the services implementation. Instead the consumer loads the service implementation dynamically, with no configuration or user- intervention required.[1]

These features make Jini ideal for a dynamic environment, where software-based services come and go, and where administration of these services must be as simple (or non-existent) as possible.

XML

XML is a document-processing standard. XML allows you to create and format your own document markups.[3] There are numerous already written and tested parsing utilities available for XML. The XML format allows for flexibility in document content allowing a parser to easily search a document for the desired information. The XML document utilized in this system could also contain information about the data being sent from Sensor to Display by the system.

Uniform data transport (XML + frames)

To enable the sensors and data types to change independently of the system a Frame object was designed and is used to encapsulate the data for transport. Regardless of the form the data may take it is written to and read from this Frame object. This Frame is what is passed by the system from the data-producing programs to the data-receiving programs. Frames are analogous to packets, minus fragmentation issues. In this way the data type can change without necessitating any changes whatsoever to our system. An XML document is used to provide the data needed to properly use the frame.

Data Stream Management

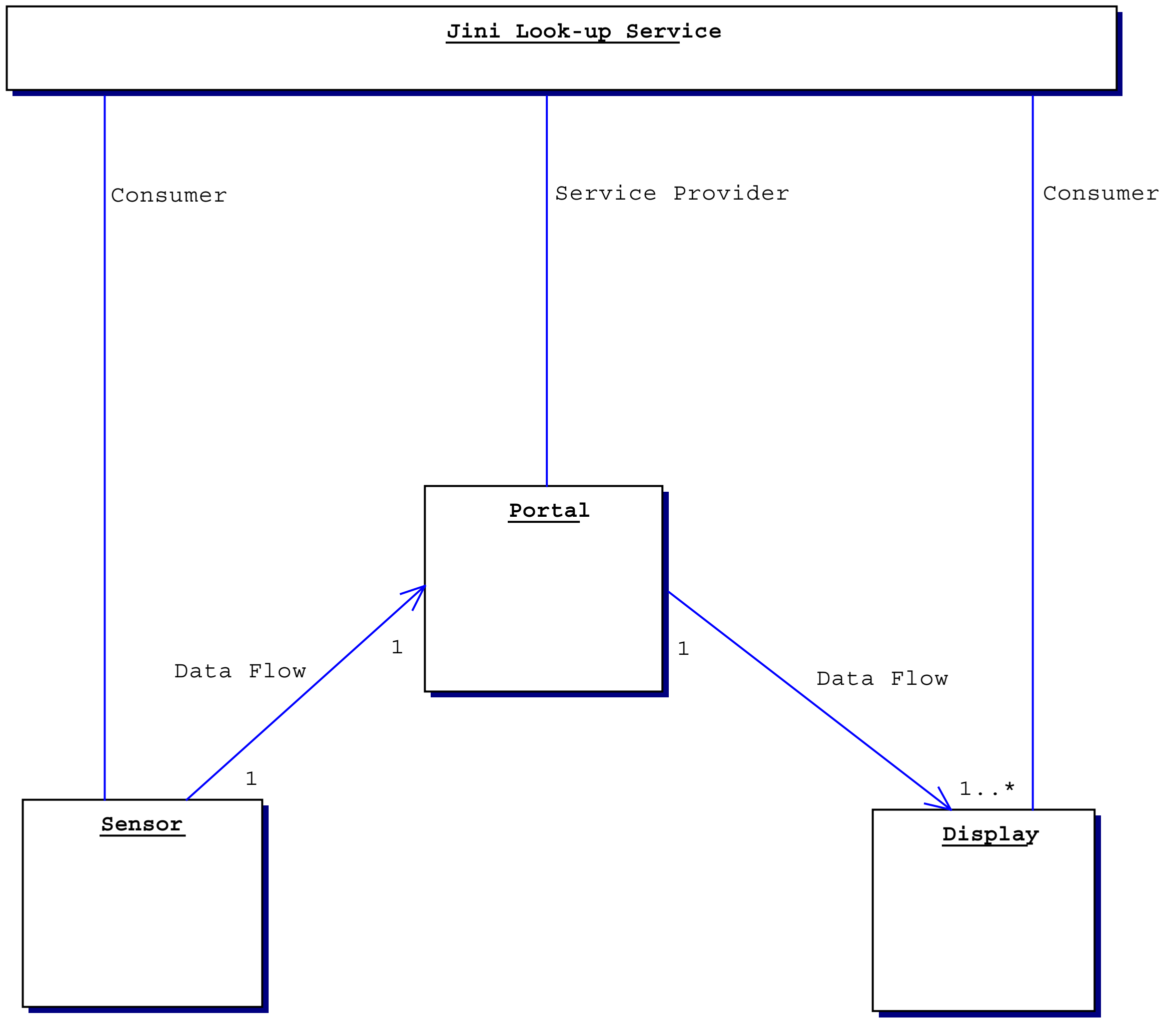
Programs will be needed to allow the data-producing programs and the data-receiving programs to produce or receive at their “natural” rates. The program acts as a buffer between the data-producing programs and the data receiving programs, saving the data until it is needed or is deemed no longer viable. The program handles the complexity of multiple data-receiving programs so the data-producing programs do not have this complexity added to them. The program is also used to provide a mechanism for filtering and processing. The program can provide data to a data-filtering program or a data-processing program and then have the filtered or processed data available for a display program.

Web Technologies

A web server makes the classes that need to be retrieved at runtime available. Using java technologies a query to a web server returns the data needed to build the class in the local java virtual machine. The query contains the locations and names of the classes to be retrieved.

Overview of the architecture

The data originates in the Sensor, moves to a Portal where it is buffered and copied if necessary, and is sent from there to one or more Displays. A Sensor can send to multiple Portals and a Portal can send to multiple Displays. The only one-to-one relationship is that any given Portal can receive from only one Sensor. A Portal can provide data to a data-filtering program or a data-processing program and then have the filtered or processed data available for a display program.



Sensor

The Sensor is the program that produces data. This is the data that is transported by our system to various destinations. The Sensor program is a wrapper that will go around ReadiNet (or an actual sensor) allowing it to use the Jini model. The Sensor can send data to multiple Portals. The Sensor produces data that is used to test transmission of data. This is necessary because ReadiNet is unavailable to us. The Sensor registers with the Jini look-up service's as a consumer. The service it is looking for is that of a data-consumer, a program that accepts data of the type produced by this Sensor. The Sensor is a typical Jini client utilizing, discovery, registration, leasing, and helper services to facilitate and administrate the network connection. See High-Level-UML for a class diagram.

Portal

The Portal is a data management program. It allows the Sensor and Display to produce or receive at their "natural" rates. The portal is the buffer between the Sensor and the Display. It takes data from the Sensor and gives it to the Display. The Portal receives from only one Sensor but sends to multiple Displays. The Portal handles the complexity of multiple Displays, simplifying the Sensor, and provides a mechanism for filtering and processing the data. The portal registers as a service provider with the Jini look-up services. The portal provides two Jini services: (1)- it takes data from a Sensor; (2)- it provides data to a Display. The portal is a typical Jini client utilizing, discovery, registration, leasing and helper services to facilitate and administrate the network connection. See High-Level-UML for a class diagram

Display

The Display is the program that is the end destination for the data. The Display is a wrapper that will go around SIMDIS and allow it to use the Jini model. The Display outputs the data it receives directly to the screen. This is necessary because SIMDIS is unavailable to us. For instance we cannot create a Display that wraps SIMDIS. The Display registers with the Jini look-up services as a consumer. The service it is looking for is that of a data producer, a program that can provide data of the type wanted by this Display. The Display is a typical Jini client utilizing, discovery, registration, leasing and helper services to facilitate and administrate the network connection. See High-Level-UML for a class diagram

Sample interaction and UML

Sensor-to-Portal

See diagram   1.2   Sensor-to-Portal

1. The Portal contacts the Jini look-up service and registers its receive port Proxy. This Proxy is what the Sensor uses to send data to the Portal.

2. The Sensor contacts the Jini look-up service seeking a receive port proxy. If none is available the Jini look-up service will notify the Sensor when one becomes available.

3. The Jini look-up sends the receive port proxy to the Sensor.

4. The first call made by the Sensor to the Portal is a call to the function “bind” this function passes the XML string to the Portal and alerts the Portal that it is about to receive data. The Portal at this point is “bound” to this Sensor.

5. The Portal stops advertising the receive data service by discontinuing its receive port proxy from the Jini look-up service.

6. The Sensor begins and continues to send data to the Portal.

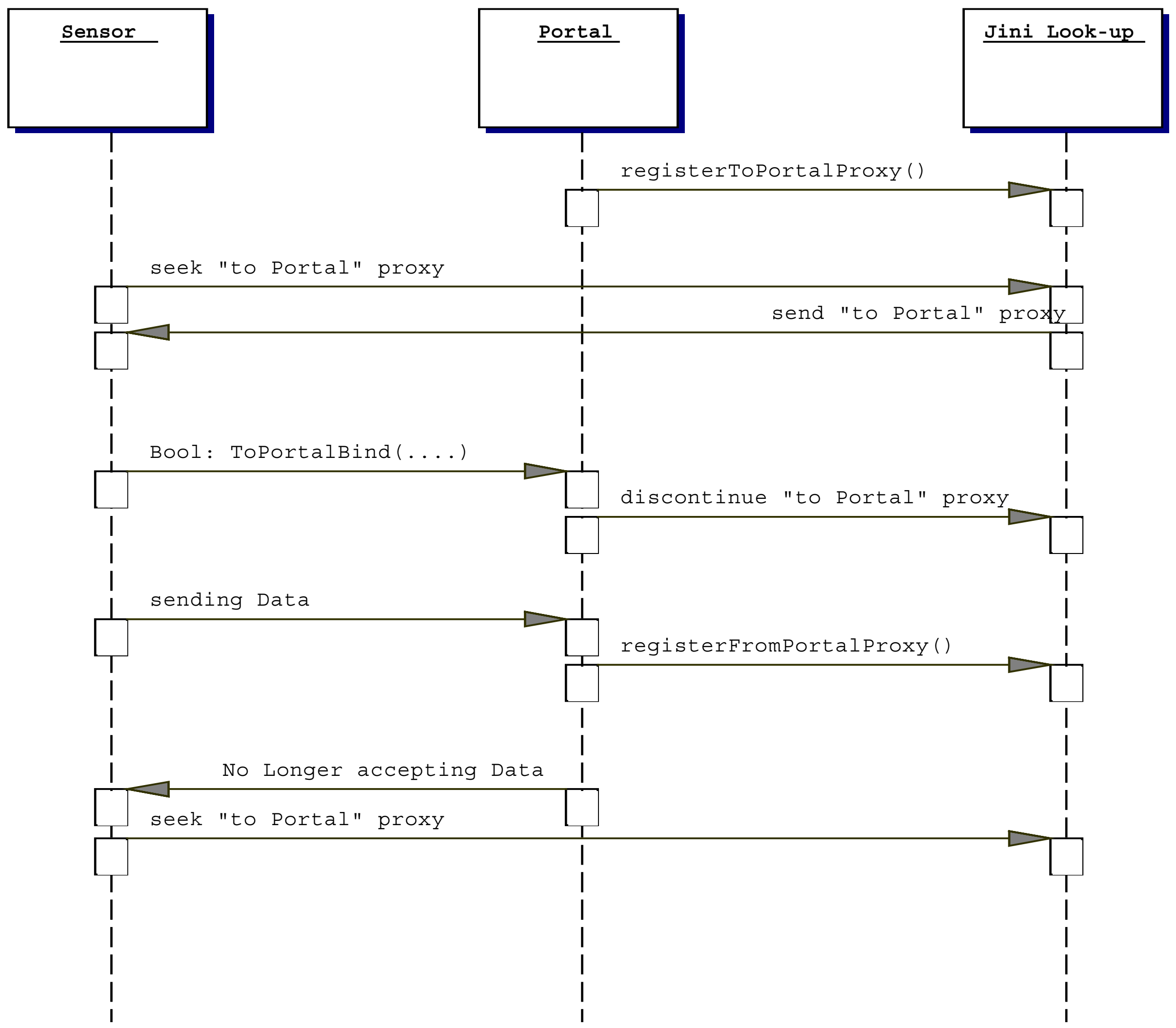
7. The Portal, now that it has data, advertises its send data service by registering its “FromPortalProxy”.

8. The Sensor is notified the Portal is no longer accepting data. (For whatever reason)

9. The Sensor starts the process over by actively seeking a “to Portal” proxy from the Jini look-up service.

(Although this sample interaction shows only one Jini look-up service. A Sensor would interact with multiple look-up services in the same way.)

Diagram 1.2 Sensor-to-Portal



Display-to-Portal

See diagram 1.3 Display-to-Portal

1. The Portal contacts the Jini look-up service and registers its “send port” Proxy ‘ This Proxy is what the Display uses to retrieve data from the Portal.

2. The Display contacts the Jini look-up service seeking a “send port” proxy. If none is available the Jini look-up service will notify the Display when one becomes available.

3. The Jini look-up sends the “send port” proxy to the Display.

4. The first call made by the Display to the Portal is a call to the function “bind” this function passes the buffer to be used with this Display to the Portal and receives a UID in return.

5. The Display begins and continues to retrieve data from the Portal.

6. The Display is notified the Portal no longer has data.

7. The Portal stops advertising its “data send” service by discontinuing its “send port” proxy.

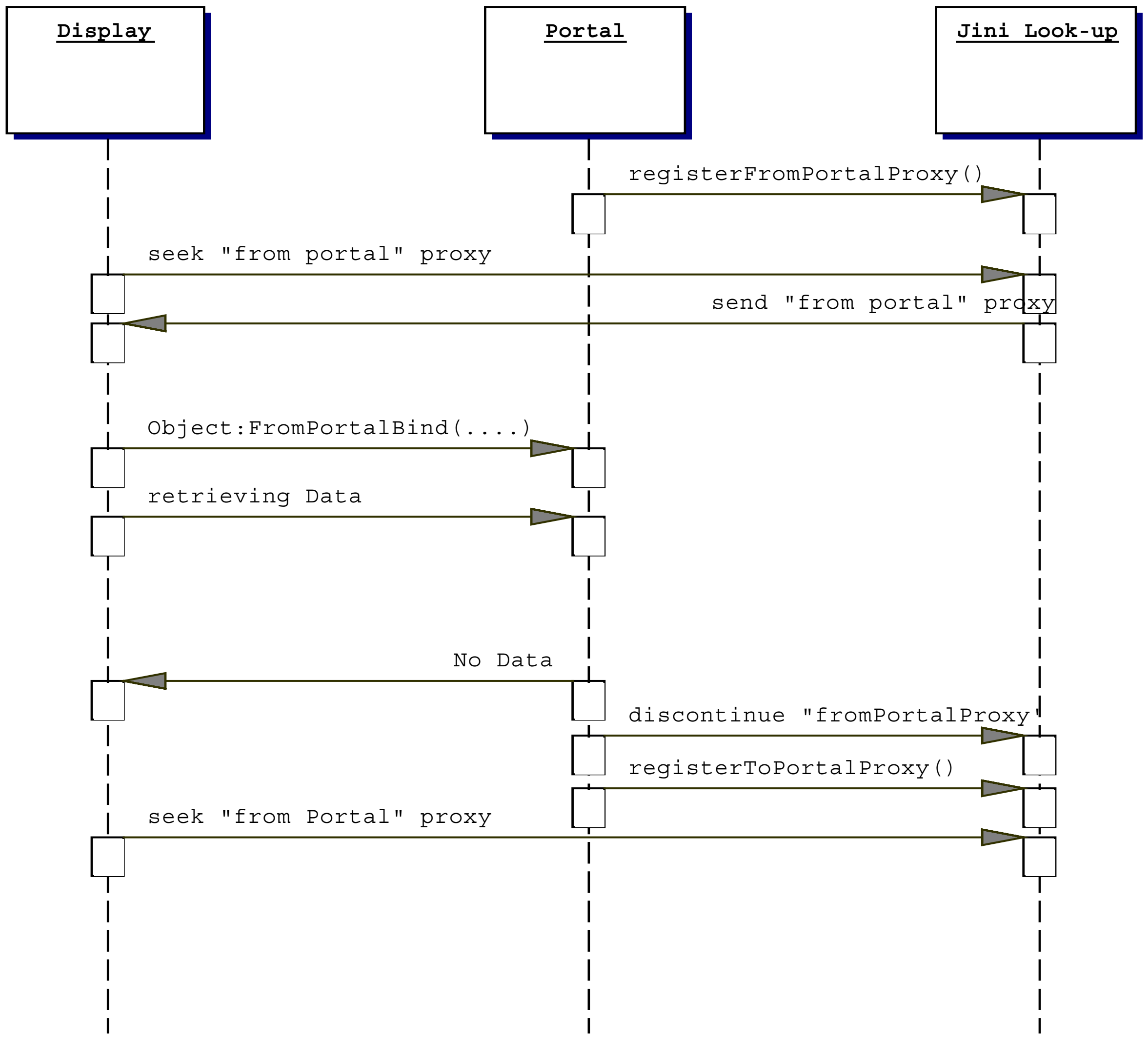
8. The Portal advertises its “receive port” service by registering its “receive port” proxy with the Jini look-up service.

9. The Display starts the process over by contacting the Jini look-up service and seeking a “send port” proxy.

(Although this sample interaction shows only one Jini look-up service. A Display would interact with multiple look-up services in the same way.)

(Although this sample interaction shows only one Display, multiple Displays’ would interact with the Portal in the same way)

Diagram 1.3 Display-to-Portal



IMPLEMENTATION

Goals

The goals of NALU System are

1. To provide a dynamic User Interface to information from different sensors of radar. Also to any amount of computations required, in-between, the final user and source.

2. To make the system is independent of the type of data and data processing (thanks to jini and XML). Hence, the application is not just restricted to the radar sensors, but, to a large number of devices.

3. To provide a plug n' work application to different devices, which also include wireless devices. For the wireless devices, the NALU system uses Surrogate Architecture, wherein, the wireless devices contact the nearest surrogate which has all the capabilities to use the NALU system.

4. To provide subject authentication and authorization or remote calls.

The Original milestones we started with

1. Develop an extensible framework for sensor/display interaction and data management.

2. To submit a job to supercomputer and then transmit the processed data around.

3. To design a surrogate for wireless devices.

4. To run the application with real data.

A summary of code

There are different types of data depending of different sensors. If these are to be passed around, the programs should be aware of all the types. This leads to lot of complexities. However, classes Sensor, Display and Portal are kept independent of the type of data by passing around Frames. These classes only need to have the knowledge of Frames. The Frame Class contains a single data variable, a byte array, in which the corresponding data is added. This is done using IDataHandler interface. This interface has methods to put data into the frame and get data from the frame. Thus any class implementing this Interface has the intelligence to put the data, in to the Frame, and out of Frame. The class implementing IDataHandler, will write out its individual data variables into the byte array, thus serializing itself into the frame. The concept of Frame was a significant step because it not only freed the Classes of the type of data but also allowed any type of data to be shipped across. Just a simple program implementing the IDataHandler interface is all that is needed.

The Portal is a ‘Jini Service’. Sensor and Display is ‘Jini Client’. Whenever a Sensor has some data to ship across, it registers with Jini Lookup service as a client, requesting for a ‘free’ Portal. If available, the Lookup service hands over a ‘proxy’ of this portal. The first thing that the sensor will do is calling the BIND method in the portal, which will make the portal available only for this sensor. Also, this method will change the attributes of the portal, so as to reflect the type of data it holds. This will avoid the ambiguity of information from different Sensors. The Portal has an ‘activatable Backend’ which then talks to the Sensor, getting the frames from sensor and buffering it. The activation framework provides a way for a remote object to consume no memory or CPU resources until it is needed. Activation provides a nice way to have very long-lived remote objects that don’t have to be active when there are not being used.

If a Portal is not available with the Lookup service, the Sensor uses Jini’s Event Listener. This will then ensure the Sensor a Portal as soon as one is available.

Similarly, Display registers with Jini Lookup service as a client requesting for a bounded portal with attributes specifying respective data. If available, it gets the proxy of this Portal. In case such a portal is not available, the Display too uses Event Listeners.

The programs are self-healing, i.e. to say that, if at all a portal goes off line for some reason, the sensor sets the event listener to get next free portal. The portal uses persistent storage, wherein it stores its service ID and its attributes. So that whenever it comes online it uses its own unique features.

How does the Display program retrieve the data from the Frame obtained? The class implementing the IDataHandler actually can be retrieved at runtime from local or remote sites. The sensor program takes an XML document as a runtime argument. This document contains all the information (location, class names, etc) needed for the Display programs to retrieve and utilize the class (or classes) that implement the IDataHandler interface. The sensor before sending any Frames sends out an XML format string stating where its class, that implements IDataHandler can be found. The Display program first parses this string to get the location of this class and then downloads it in its JVM using ClassLoader (a Java class to load a remote class into the local JVM).

Furthermore, Display sends a buffer to be used with their particular data in this way providing their own strategy for data buffering, expiration, retrieval etc. Thus we can have one-to-many relationship between the Portal and Displays.

One of the latest additions in the current design/development iteration is the processing of the data by a supercomputer one of the clients sends a buffer that keeps copies of all data passing through it and when a flag amount of data has been gathered a job is submitted to Blackbear, one of the supercomputers at AHPCC, in this design iteration it is a “toy” job submission to learn how to best perform this sort of activity.

Sample classes

Frame Class This is the most general and most important class. The types of data may vary depending on different sensors. However, we need a system that should work with any data type. Here comes the concept of Frame. Instead of having all the programs know all data types, they now, need to know only the Frame class. The Frame class contains a single data variable, a byte array, and has methods to write the byte array and read the byte array. A byte array is most general data type, i.e. any data type can be converted to a series of bytes. Thus, we just need to get our data into this byte array.

```
Class Frame implements Serializable {
     byte [] bytearray;

     public Object getObject() {
           return bytearray;
     }

     public Object setObject(byte[] input) {
           bytearray = input;
     }
}
```

IDataHandler Interface This interface defines methods that will read the proper data from the frame and write the data into the frame.

```
public interface IDataHandler {
     public void readFromFrame(Frame input);
     public void writeToFrame(Frame input);
}
```

DataHandler Class This class implements the IDataHandler Interface and hence defines the above two methods. Presently this class has “toy” data, which is serialized (or written in byte form) into the Frame. This Class serializes its data in its own way. On the other side (Display), this class is downloaded at runtime to extract the information using readFromFrame(Frame input) method.

```
public class DataHandler implements IDataHandler, Serializable {
     <data variables>
           …
     public void writeToFrame(Frame input) {
                 …
           <writes the serialized information into the Frame's byte array>
                 …
     }
     public void readFromFrame(Frame input) {
                       …
           <reads the serialized information from the Frame's byte array>
                       …
     }
```

```
        public byte[] writeObject() {
                        …
        <serializes its data into a byte array. this method is called in
        writeToFrame method>
                        …
        }
        public void readObject(Object ob) {
                        …
        <reads out the contents from the byte array into the data variables
        of this class>
                        …
        }
  }
```

Backend Class This is an inner class of Class Portal. The Portal class is actually a wrapper over Backend class. The actual process of buffering the data is done in this class. While the Portal class keeps track of lookup services, leasing, discovery etc. This class has Sensor only methods, and Display only methods. An important point here is that this class is activatable. That is to say, it comes alive only when called. This framework provides a way for remote object to consume no memory or CPU resources until it is needed.

```
 public static class Backend extends Activatable implements BackendProtocol
 {
        public Boolean toPortalBind(String input) {
                  …
        <method called by Sensor. This will bind the portal to the particular
        Sensor and also the XML string is passed>
              …
        }
 public boolean toPortal(Object input) throws RemoteException {
                        …
        <this method is called by Sensor. This method stores the Frames in a
        buffer>
                        …
        }
        public Boolean fromPortalBind(Object buffer) {
                  …
        <called by Display. The display is given a unique Identity number(ID)
        and based on its ID its type of buffer is stored>
                  …
        }
        public Object fromPortal() throws RemoteException {
                  …
        <this method is called by Display. This method sends the next frame
        from the buffer to the display>
                  …
        }
        public Object fromPortal(int UID) throws RemoteException {
                  …
        <this method is called when the display send a buffer specifying the
        way it wants the data. The UID implies unique identity given by
        portal to the particular Display>
                  …
        }
  }
```

There are many classes providing different functionalities. These were few fundamental classes. The modules Sensor, Display, and Portal are wrapper classes which keep track of lookup services, discovery, leasing, etc.

Capabilities of existing code

The code is in complete agreement with Jini specifications and uses powers of jini Technology.

The Portal, a Jini service, uses an Activatable Back-end process, which implies that the backend comes alive only when called, otherwise it remains deactivated. This framework provides a way for remote object to consume no memory or CPU resources until it is needed.

All the programs need to know only the Frame class. Thus the data is decoupled from the system. Hence, changes to the data do not require changes to/recompilation of the system. This keeps the process simple and software more robust.

The IDataHandler interface has the methods to write data into the Frame, and read data from the frame. Only the class that implements IDataHandler knows how the data is written. The display needs to know ONLY the IDataHandler interface and it downloads the class (es) that implements IDataHandler interface, at runtime, and loads it in its JVM.

This downloading of classes saves a lot of bandwidth. Since transferring the class information with *each frame,* adds up a significant amount of overhead. This is especially useful for wireless devices.

XML provides platform independent data. XML format allows incredible flexibility because of the META information it contains describing itself.

The code is superbly dynamic. Once the data is available with the sensor, it sets an event listener, whereby, once a free portal is available it gets that Portal and the data is transmitted without human intervention. For any data, we simply have to write a class implementing the IDataHandler interface.

The portal uses persistent data storage. This implies that it stores its attributes, serviceID in a file. So that if for any reason it goes off line, upon restarting it will use the same information rather than starting the process again.

Once a sensor gets a Portal, it first binds the Portal to it. Thus the portal will be available to this sensor only. This will avoid ambiguity of information from different Sensors.

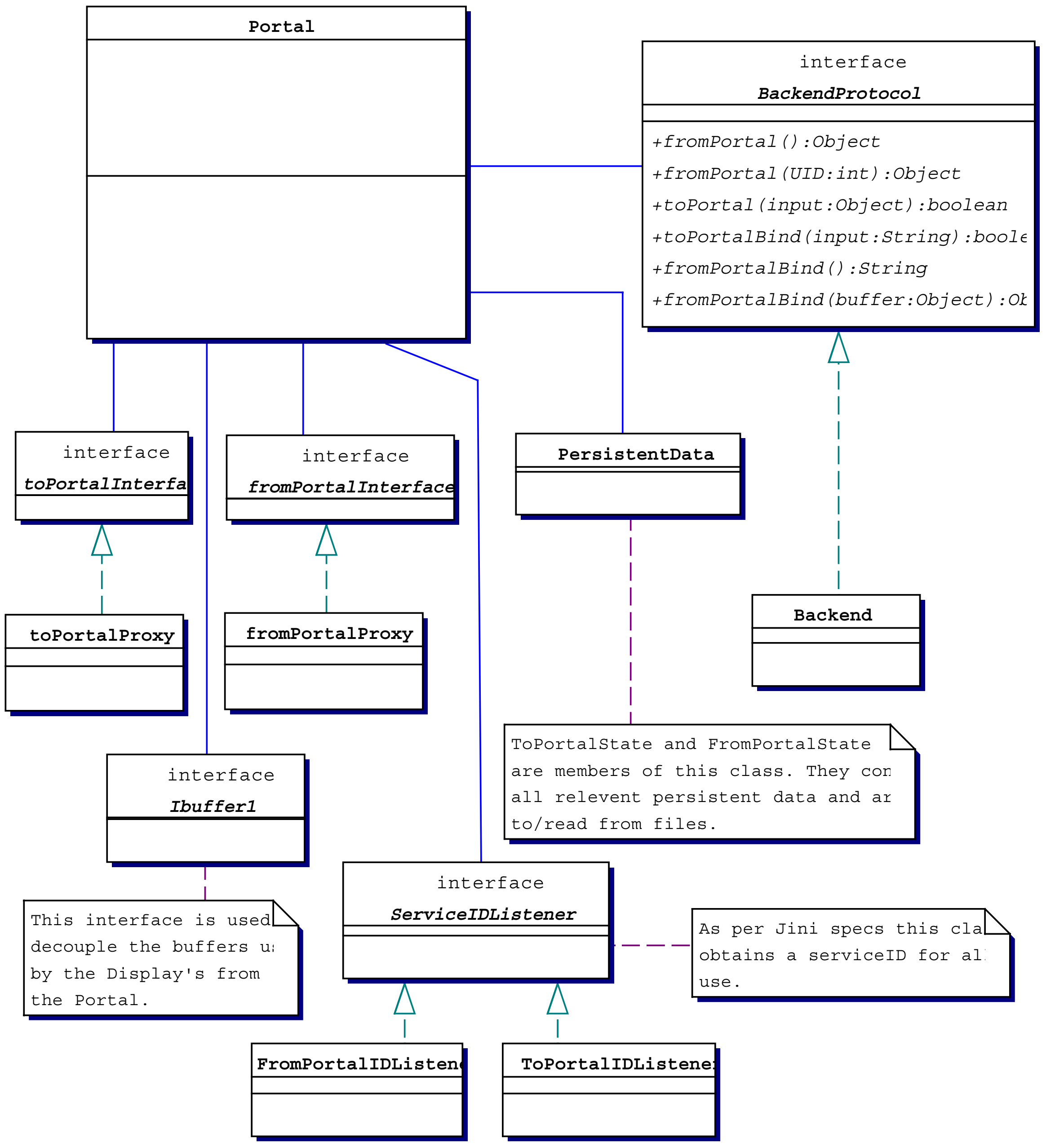
Once the Portal is bound to a Sensor, it registers with Lookup so that it is then available for Displays requiring that data. Furthermore, each Display sends a buffer to be used with their particular data in this way providing their own strategy for data buffering, expiration, retrieval, etc.

All three modules, Portal, Sensor, and Display are robust and well behaved. They correctly handle a shutdown or crash by any or all of the other modules. They also, actively pursue new services on the network when they are freed from their current connection due to job termination, shutdown, or a crash.

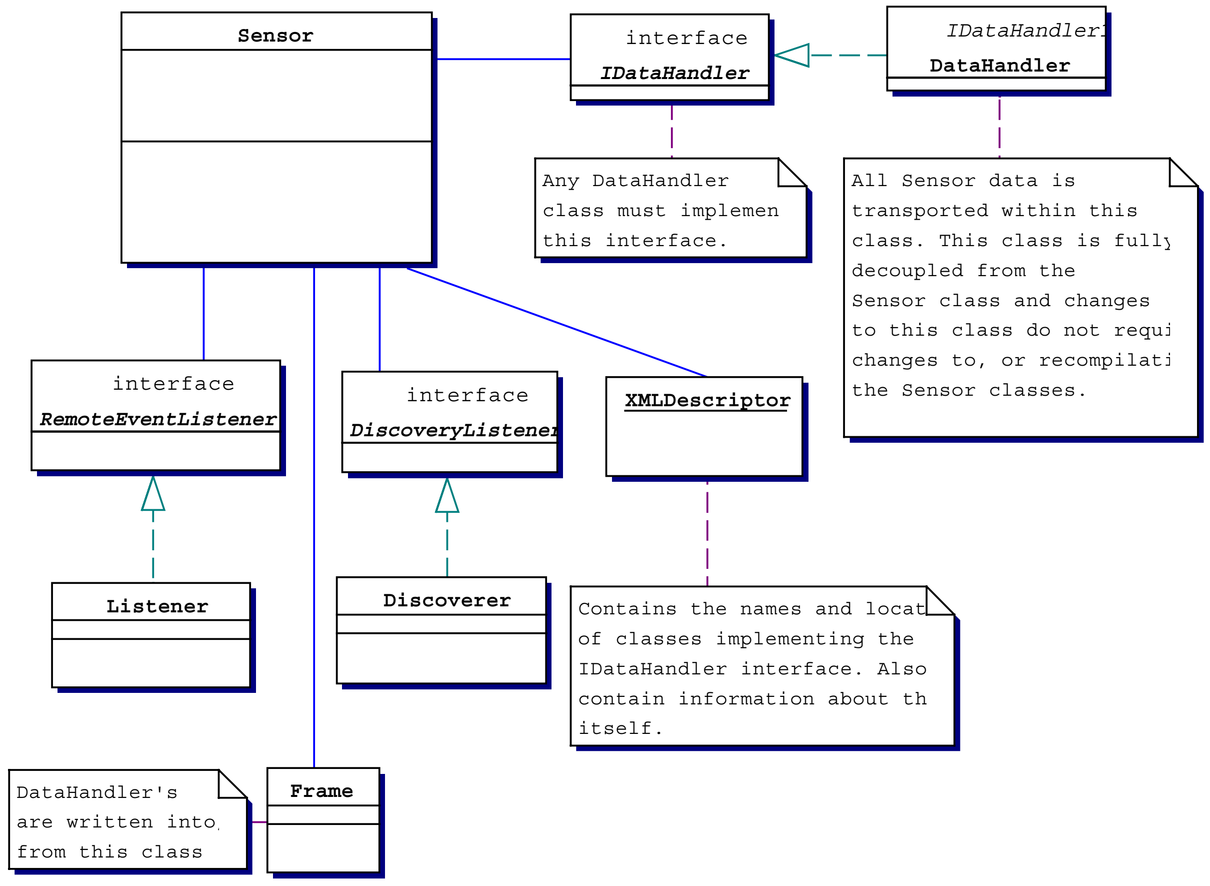
The code supports, any number of data processing between the Portal and Display. This was one of the latest additions in the current design/development. For example, one of the Displays sends a buffer that keeps copies of all data passing through it and when a flag amount of data has been gathered a job is submitted to the respective supercomputer. The result is then passed to display.

High-Level UML

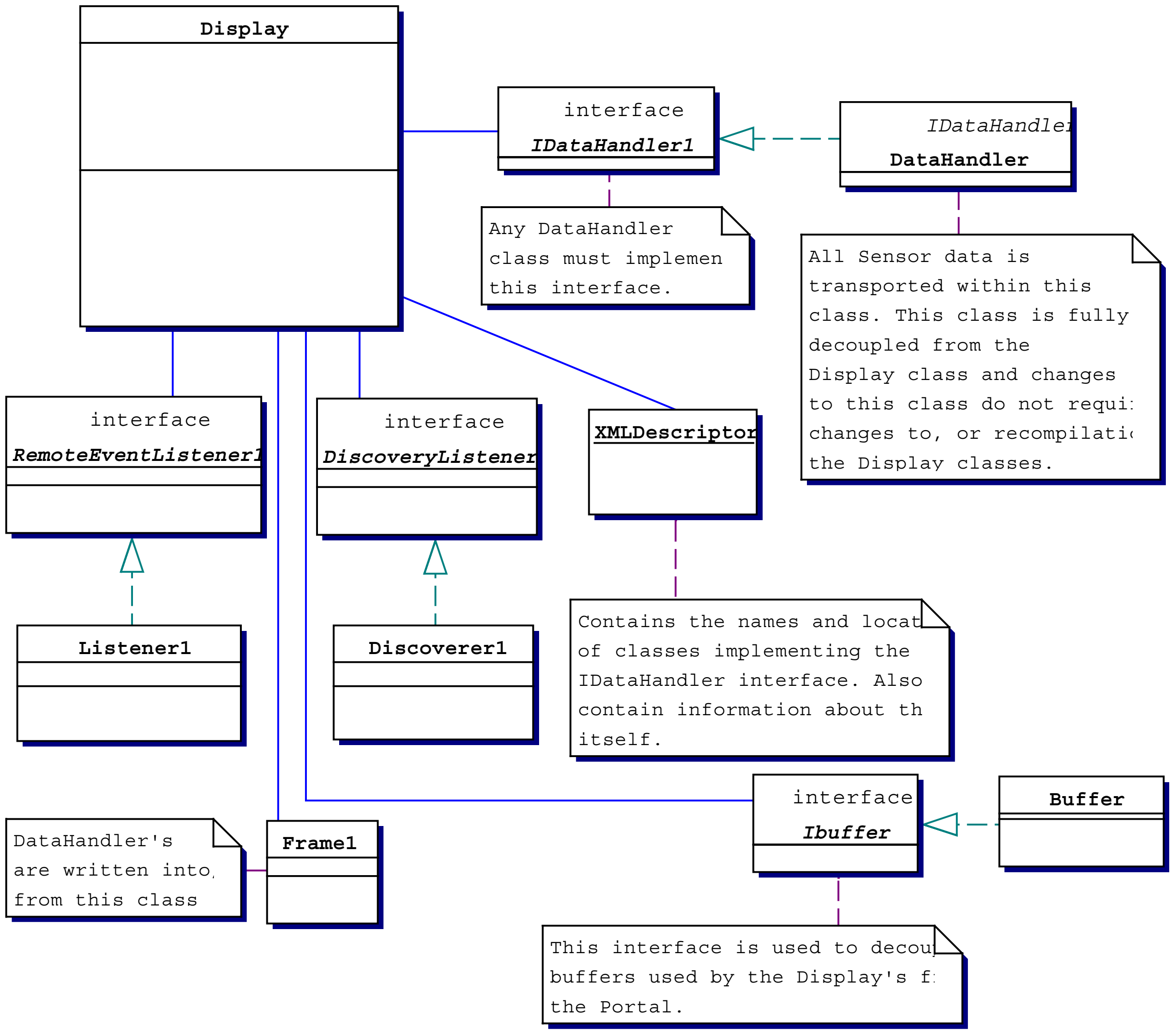
Portal


Portal
interface
BackendProtocol
+fromPortal():Object
+fromPortal(UID:int):Object
+toPortal(input:Object):boolean
+toPortalBind(input:String):boole
+fromPortalBind():String
+fromPortalBind(buffer:Object):Ob
interface
toPortalInterfa
interface
fromPortalInterface
PersistentData
toPortalProxy
fromPortalProxy
Backend
ToPortalState and FromPortalState are members of this class. They con all relevent persistent data and ar to/read from files.
interface
Ibuffer1
interface
ServiceIDListener
This interface is used decouple the buffers u: by the Display's from the Portal.
As per Jini specs this cla obtains a serviceID for al use.
FromPortalIDListen
ToPortalIDListene

Sensor


Sensor
interface
IDataHandler
IDataHandlerI
DataHandler
Any DataHandler class must implemen this interface.
All Sensor data is transported within this class. This class is fully decoupled from the Sensor class and changes to this class do not requi changes to, or recompilati the Sensor classes.
interface
RemoteEventListener
interface
DiscoveryListener
XMLDescriptor
Listener
Discoverer
Contains the names and locat of classes implementing the IDataHandler interface. Also contain information about th itself.
DataHandler's are written into from this class
Frame

Display



Display
interface
IDataHandler1
IDataHandler
DataHandler
Any DataHandler class must implemen this interface.
All Sensor data is transported within this class. This class is fully decoupled from the Display class and changes to this class do not requi: changes to, or recompilati(the Display classes.
interface
RemoteEventListener1
interface
DiscoveryListener
XMLDescriptor
Listener1
Discoverer1
Contains the names and locat of classes implementing the IDataHandler interface. Also contain information about th itself.
interface
Ibuffer
Buffer
DataHandler's are written into, from this class
Frame1
This interface is used to decou buffers used by the Display's f: the Portal.

Status and Future

Accomplishments

Phase I Designed a robust system to transfer toy data from Sensor to Display using the middleman Portal. All the three modules were robust, dynamic and well behaved. All three modules, Sensor, Display, and Portal were independent of type of data being transferred. All they need to know was Frame Class. The data from a sensor was serialized into the byte array of Frame Class. And a Frame object was shipped. The IDataHandler interface was enough to deserialize the data from the byte array of Frame.

Phase II The code was further extended to be robust and well behaved. With introduction of Frame Class and IDataHandler interface, we simply have to write a DataHandler class (default name, the name of this class can be anything, maybe related to the type of data it handles) for the type of data from sensor. The DataHandler class will then implement the methods in IDataHandler interface. This DataHandler class was then kept at a known location. And instead of shipping class information each time an XML format string is shipped first. This string holds the class name and location information. XML provides incredible flexibility because of the META information it contains. The Portal does not need to have any information of DataHandler class or its location. Its just buffers the string and the frames. Upon request from a Display, it sends the string across. The Display then parses the string and obtains the class name and location. It downloads it in its JVM using the CLASSLOADER class. Once in the JVM, the Display can then know all the methods and data variables in it by using REFLECTION. 'Reflection' is the ability of software to analyze itself. It allows you to analyze a software component and describe its capabilities dynamically, at run time rather than at compile time. Display then extracts the information from DataHandler class and uses the data in the way it is designed to display.

Further, the Portal can support multiple Displays. The Portal provides each display with a unique identity number, and uses this to transfer the frames. Furthermore, each Display sends a buffer to the Portal specifying the way it wants the data. For example a display program may request only the data from past 5 minutes!

Also, the Display can send a buffer that keeps copies of all data passing through it and when a flag amount of data has been gathered a job can be submitted to a supercomputer. Thus we can obtain any type of data processing in between to obtain desired information.

Table of classes

Frame class has a byte array and holds the information to be shipped

DataHandler Class implements the IDataHandler interface, and has methods to put data into the Frame object and also to get the data from Frame.

Sensor Class is a wrapper class for sending data (Frames), and participates in Jini discovery, Leasing, keeps track of available lookup services, and available Portals.

Display Class is a wrapper class for display of data, and participates in Jini discovery, Leasing, keeps track of available lookup services and the Portals holding the required data.

Portal Class is a Jini service. Registers with lookup services and provides an interaction between Sensor and Display.

DefaultBuffer Class A buffer in which the frames will be stored by Portal and distributed to different Displays depending on the buffer sent by them.

BindInfo Class has methods to set and get the XML string containing the class name(s) and location(s) that have intelligence to extract data from the Frame. Also holds information of the unique identity provided to Display by Portal.

ClassInfo Class encapsulates the class name and class location (of DataHandler class).

StringParser Class used by Display class to parse the XML string obtained from Portal containing the class name(s) and location(s) of DataHandler class.

BufferTwo Class implements IBuffer interface and is used by Display class to specify the type of buffer it wants from the Portal.

IDataHandler Interface has methods to insert and extract data from the Frame.

ToPortalInterface used by Sensor class to ship frames to Portal.

FromPortalInterface used by Display to get frames from Portal.

IBuffer Interface has methods which are used by buffer as required by Display from Portal.

Future work

1. To extend the capability of this code to wireless devices. However, the wireless devices may not have enough memory. Hence to design a surrogate which can then use the existing code itself. A surrogate will have just enough computational resources to be able to talk to both the devices and the NALU system. The surrogate architecture is discussed below.

Surrogate Architecture

For a device to participate in a Jini network, it should be able to perform discovery, download and execute classes written in the Java programming language, and also may be needed to export classes written in Java. Many devices, for one reason or another, cannot participate directly in a Jini network.

Surrogate architecture exactly addresses this problem by defining a means by which these devices, with the aid of third party, can participate in a Jini network while still maintaining the plug-and-work model of Jini technology. The surrogate architecture is explained in detail after this section.

2. To use real sensor data and to improve upon the DataHandler class to read directly the sensor data and serialize it into the Frame.

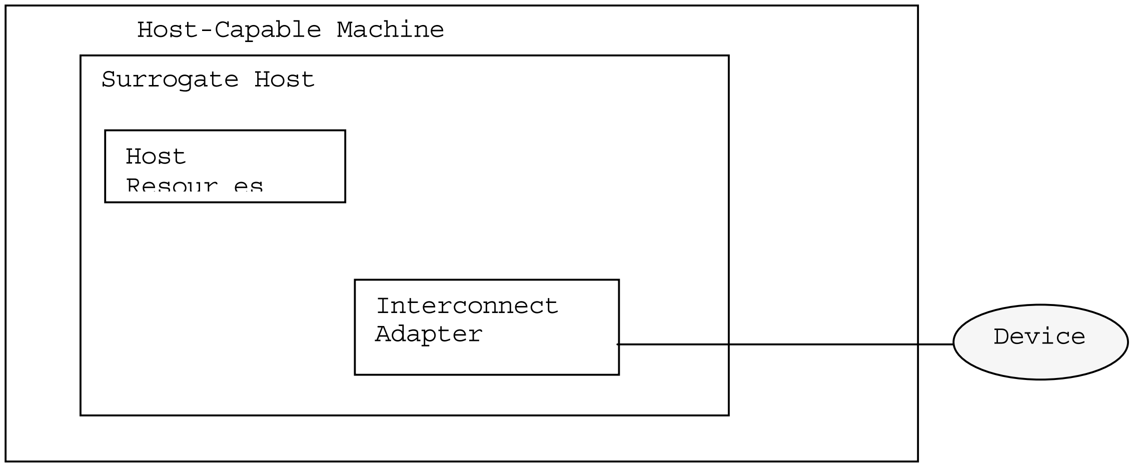
3. The submission of the job to the super computer is now just a test job submission. The archived data must be transported to the super computer so that real processing can occur.

4. Integrate the complete code and have a web GUI (Graphic User Interface), so that the user simply has to choose from a menu.

Surrogate Architecture

For a device to participate in a Jini network, it should be able to perform discovery, download and execute classes written in the Java programming language, and also may be needed to export classes written in Java. Many devices, for one reason or another, cannot participate directly in a Jini network. Surrogate architecture exactly addresses this problem by defining a means by which these devices, with the aid of third party, can participate in a Jini network while still maintaining the plug-and-work model of Jini technology.

The surrogate acts as a communicator between the device and the Jini network. The machine (called Host capable machine) which runs a surrogate should have computational resources to execute code written in Java on behalf of the device and also provide the necessary resources that such code may need. The surrogate architecture is targeted for devices that are not capable of downloading coded, because of either computational resource or network connectivity limitations.



(figure from Jini™ Technology Surrogate Architecture Specification[2] version 0.6)

The figure shows a surrogate host present on a host-capable machine. The Host-capable machine has the surrogate host, and the required resources. An interconnect adapter monitors the device interconnect. The first part of surrogate protocol is discovery, where, the device and the surrogate host find each other. Once discovery has been performed the device's surrogate is retrieved. Once the surrogate is loaded, the reachability of device is checked. Reachability means that a usable communication path exists between the surrogate and the device that it represents. Once the surrogate is executing, it may perform any task necessary for the device, including accessing the Jini network. If the device becomes unreachable for some reason (like the device being switched off, out of range, surrogate failure…) the surrogate must be stopped and resources restored/ device must search for an available surrogate depending on the reason for loss of communication.

In Detail A surrogate is a Java Object that represents a device. It may be retrieved from a device in an interconnect-specific manner or from some other appropriate source and loaded into the surrogate host. The host executes the code within the surrogate to operate on behalf of the device. The surrogate host should provide for a Java 2 platform runtime environment, a Jini runtime environment, export server and surrogate API's. The minimal Java 2 platform is Java 2 standard edition v1.2.2. Each surrogate is given its own thread and its own thread group. The surrogate may be executed in a separate thread group within the surrogate host's virtual machine or a thread group in a separate virtual machine. The Jini API classes must be available through the surrogate's class loader before a surrogate is loaded. The *export server* is the means by which surrogate resources are exported to remote clients. The export of resources particularly class files are necessary for many Jini operations. The surrogate specifies which resources are to be exported in a file that is packaged along with other surrogate information. Each surrogate has its own export codebase annotation. All of the URLs generated by the export server that refer to JAR resources are used by the surrogate host to set the export codebase annotation of the surrogate's classes. The surrogate is packaged in a JAR file containing

A manifest file containing the following elements:

a `SurrogateActivator header` specifying the name of surrogate activator class and

`SurrogateExport` headers specifying resources (if any) to be exported by the surrogate host on behalf of the surrogate.

The resources that implement the surrogate. These resources may be classes written in Java, as well as any other data such as HTML files, icons.

A class that implements the `SurrogateActivator` interface.

SurrogateActivator is an interface defined in surrogate APIs net.jini.surrogate package. This interface defines the methods used to control the execution of the surrogate namely activate and deactivate. The activate method is called by the surrogate host to activate the surrogate. This method is used to allocate resources and start any threads that the surrogate needs. This method must return in a timely manner. If this method throws an exception, surrogate host assumes that the surrogate is not active and discards the surrogate. The deactivate method is called to deactivate the surrogate. All the threads started by this surrogate must be deactivated i.e. this method should undo the work done by the activate method. The method activate takes instances of HostContext interface (hostContext) and Object (called here interconnectContext). The type of interconnectContext Object depends on the interconnect utilized by the associated device.

```
Package net.jini.surrogate;
Public interface SurrogateActivator {
     void activate (HostContext hostContext, Object interconnectContext)
                                 throws Exception;
     void deactivate ();
}
```

The HostContext interface is also defined in net.jini.surrogate package and provides methods to access the execution environment of surrogate host.

```
Package net.jini.surrogate;
Import net.jini.discovery.DiscoveryManagement;
Public interface HostContext {
     DiscoveryManagement getDefaultDiscoveryManager ();
     void cancelActivation ();
     java.net.URL[] getExportURLs ();
}
```

The getDefaultDiscoveryManager method returns an object that implements the DiscoveryManagement interface. This object defines the default discovery management policy for the surrogate host.

The method informs the surrogate host that the surrogate is to be deactivated. The host does not call the surrogate activator’s deactivate method as a result of a call to cancelActivation.

The getExportURLs method returns an array of URLs representing the exported resources specified by the surrogate and being handled by surrogate host’s export server. The number of elements in the array correspond to the number of Surrogate Export headers in the surrogate JAR file.

Along with above interfaces defined in package net.jini.surrogate is also defined GetCodebase interface. Now, each surrogate has its own export codebase annotation and by default the surrogate host sets the export codebase annotation to be the list of export JAR file resources defined in the Surrogate Export headers. Using this interface, the surrogate can overwrite this list by a new export codebase.

```
 Package net.jini.surrogate;
 Public interface GetCodebase {
 java.net.URL[] getCodebase (HostContext hostContext,
                                 Object interconnectContext)
                          throws Exception;
 }
```
The above interfaces are independent of the type of interconnect. In version 0.6 (the current version) of Surrogate Architecture defines two more interfaces (`keepAliveManager` interface and `keepAliveHandler` interface) that are interconnect dependent. Their use is dependent on how reachability is defined for an interconnect.

An interconnect is defined as the logical and physical connection between the surrogate host and a device. The *interconnect specification*[4] defines protocol and model for IP-capable interconnects. IP can be implemented on various physical layers such as Ethernet or wireless networks. The IP surrogate protocol (defined in Interconnect specifications) is applicable to any physical layer that supports IP. The first thing done is Discovery. By this, the surrogate host and the device locate each other.

The surrogate host uses a *multicast host announcement protocol* to announce its presence on the interconnect and indicate that it is available to serve as a host. For this, it creates a listener to listen for registering requests, constructs an announcement message containing its IP address and port used and multicasts at regular intervals. A device using this *protocol* creates a listener for host announcement messages and upon receiving a host announcement message, sends a registration request to the host using the IP address and port contained in the message.

However, the discovery can also be initiated by device. The device uses *multicast host request protocol* to find a surrogate host at an unknown location. For this, the device creates a UDP listener to receive host response message, constructs a host request message that contains the IP address the port of the host response listener and multicasts this at regular intervals. Upon receiving a host response message, it sends a unicast UDP registration request to the IP address and port contained in the host response message. A surrogate host performing this *protocol* creates a listener for registration requests, creates a listener for host request messages and upon receiving such a message, sends a host response message providing its IP address and port for registration to the address in the request message.

The type of discovery is dependent on whether the device can perform discovery or not. However, once a device has obtained the address and port of surrogate host it can send a registration request either as unicast UDP message or open a TCP connection to surrogate host and send the message across using this connection. If the registration is sent as a unicast UDP packet the message size is limited to 512 bytes. The registration request message[3] should have the following format:

```
int                              protocol version

int                              length of surrogate URL
```

`byte[]`	`surrogate URL`
`int`	`length of initialization data`
`byte[]`	`initialization data`
`int`	`length of surrogate`
`byte[]`	`surrogate`

the surrogate field is a byte array that surrogate host interprets as a JAR file containing the device's surrogate. The contents of the JAR file were described above in the first paragraph of ***In Detail.*** Upon receipt of registration request the surrogate host, checks for surrogate URL, and if present, retrieves the surrogate JAR file from that URL. If not present (length of surrogate URL field is zero), attempts to extract the surrogate JAR file from the registration request message itself. Once the surrogate is executing, it may perform any task necessary for the device, including accessing the Jini network.

Acknowledgments

The Office of Naval Research through federal contract with the Maui High Performance Computing Center made this research possible, contract # 29601-96-D-0128.

Brian Sayrs was the project head at the Maui High Performance Computing center. The authors gratefully acknowledge his support.

Robert Ballance was the project head at the Albuquerque High Performance Computing center. The authors gratefully acknowledge his support.

We thank Brian Bowers a former employee of the Albuquerque High Performance Computing center Brian was on the project in the early design stages until he left to attend Graduate school at the University of Wisconsin at Madison.

References

1. W. Keith Edwards, *Core JINI (1999 Prentice hall PTR Prentice-Hall Inc. Upper Saddle River, NJ 07485)*

2. http://developer.jini.org/exchange/projects/surrogate/surrogate.pdf

3. Didier Martin, etal. *Professional XML* (2000 Wrox Press LTD., Birmingham, UK.

4. http://developer.jini.org/exchange/projects/surrogate/IPInterconnect.pdf