Java RMI
Middleware Project

Nathan Balon
CIS 578
Advanced Operating Systems
December 7, 2004

Introduction

The semester project was to implement a middleware similar to Java RMI or CORBA. The purpose of the middleware is to connect two separate applications together. Middleware is an additional layer that is added on top of a network operating system, which allows a program to call a method in a remote object. Middleware provides added functionality to an application and it is placed between the network operating system and an application. The implementation for my project reflected Sun Microsystems implementation of Java RMI more so than CORBA.

1.0 Main Component's

A number of components were used to create the middleware application. The components used can be divided into four categories. First, there are the components that make up the registry. Second, a number of components are used on the server. Second, a number of components are used on the client. Last the there is an active replication module. Tying all of these components together creates a functional middleware application. Each of the components will be examined in detail.

1.1 IDL Compiler

Before discussing the main components of the middleware program, an IDL compiler was developed. The IDL compiler is used by the application to generate the stubs and skeleton components that are used on the client and server. The compiler reads both implementation and interface class files and generates stubs and skeletons to be used by both the client and server applications. Although this wasn't required for the project, I decided to implement it any ways since it would be pointless to design a middleware program the only worked for a single Java class. The class StubSkeletonGenerator uses Java reflection to construct both the stub that is used on the client and the skeleton that is used on the server.

1.2 Registry

The registry is used to locate objects in the system. When a Server creates an object that will be shared with other applications it send a message to the registry to bind the object. When the registry is binding an object it creates a remote object reference that is used to uniquely locate a service that is provided by the server. So, when a server binds an object to the registry, the registry returns the remote object reference for that object. The registry is also used by client application to locate the available services. The client application sends a lookup message to the registry, which returns a remote object reference for an object. The benefit of using a registry is that the application only needs to know the location of the registry. The client application is unaware where the actual remote object they are using is located at. The registry also supports the use of active replication. If Active replication is used registry will give the client the location of the front end to the active replication system. The client will then communicate with the front end which will forward the clients request to the replicated object. Again the location of the replicated objects is transparent to user of the application.

The client and server use the Naming to lookup and bind remote objects. If the client wishes to lookup an object they supply the location of the registry and the name of the service that they wish to use. If the object exists in the registry, the client is returns a remote object reference to a remote object. The server on the other hand uses the rebind method to bind a remote object. The server supplies the location of the registry they wish to contact and the name they wish to use to have the service looked up by. After a service is bound in the registry clients can use the services provided by the object.

1.3 Server

The server component is used to host the remote objects that are accessed remotely by the client application. The server component has a number of subcomponents. The components that are used by the server are a communication module, a dispatcher, a skeleton, a remote reference table and the actual object that will be called remotely by the client application. These five components make up the server for the middleware application.

The communication module is used by the server to send messages back and fourth to the client. The communication module uses a request reply protocol. It composed of two classes they are ServerCommunication and ServerCommunicationHandler. The Server uses multi-threading, so it can handle multiple connections simultaneously. ServerCommunicationHandler objects are the threads that are created to handle the communication with the client. When a client sends a request to the server, the communication module receives the request and passes the message along to the dispatcher. When the request has completed the dispatcher returns the message to the communication module and it sends a reply message back to the client.

The dispatcher class is responsible for finding a skeleton that can handle the method call that was requested by the client. The dispatcher locates the skeleton for a remote object reference in the remote reference table on the server. The dispatcher then gets the name of the method to be invoked and the arguments of the method from the message sent from the client. The dispatcher invokes the method on the skeleton object. When the skeleton completes, it returns the results back to the dispatcher, which then returns the results back to the server.

The skeleton is used to unmarshal and then marshal a message. When the dispatcher invokes a method on the skeleton it passes it the message that was received from the communication module. The skeleton then unmarshals the arguments that are to be passed to the remote object. Once the arguments have been unmarshalled, the skeleton invokes the method on the remote object and if any results are returned from the invocation, it marshals these values and returns them back to the dispatcher.

Any class can be used as a remote object in the application provided that the class implements an interface and uses StubSkeletonGenerator to compile the stub and skeleton classes that are needed. In the case of the middleware application StudentListImpl is used. This is just a typical Java class that can have it methods invoked remotely. The StudentListImpl provides a number of methods that the clients use to manipulate a list of student objects. Some of the methods provided by the remote object are to add a student

and to list all students that are contained in the list. Once a method is invoked on a remote object, the results are returned to the skeleton, which eventually returns the results to the client application.

The final component that is held on the server is a remote object reference module. The module is used to store the remote object references and skeleton for a remote object. The main use of the remote object reference module on the server is for the dispatcher to locate a reference to a skeleton object from a remote object reference.

1.4 Client

The client application uses the services that are provided by a remote object on the server. What distinguishes the RMI middleware from sending messages to a server through a socket connection is the user is basically unaware of the underlying implementation. Users of a remote object invoke methods on an object and are basically unaware that the method is being invoked on another computer. The only difference is before an object can be used for the first time the Naming service must be used to locate a reference for a remote object. After that, the client is able to use the object as if it were an object located locally.

A stub object is created for the client when they use the Naming service to lookup an object. If the remote object is successfully located, the Naming service will create a stub to use on the client. A stub object is invoked by using the interface that is defined for the remote object. To the user of the application the method calls appear to be performed locally. The stub is used to marshal and unmarshal the values that are sent as messages. The stub acts as a proxy for the local object. The stub is implemented by using the proxy design pattern. The local object actually invokes methods on the stub. The stub then marshals the arguments that were passed to it and passes the arguments to the communication module. When the communication module receives a reply message from the server it passes the message to the stub which unmarshals the results and returns the result to the local object.

The communication module on the client is very similar to the communication module on the server. About the only difference between the two is the server communication module is multi-threaded while the client's communication module uses a single thread of execution. The client's communication module is used to send and receive messages to the server. The communication module sends a request message to the server and then waits for the server to reply. The communication module is also used to create Lamport timestamps so messages can be ordered by a "happens before" relationship.

The last module that is used by the client is the remote reference module. The remote reference module is used to store remote object reference objects and local references. This module is used to locate local references.

1.5 Active Replication

The middleware system uses active replication for fault tolerance. By providing active replication, if one of the servers that hold a remote object fails or is terminated, it is still

possible for a client to use the services provided that another server that is still active. The active replication is provided by the use a front end manager. When a method is called on a remote object the request is sent to the front end. The front end then sends the request to all of the remote objects in the system registered under the same name. So, all objects are updated at the same time. In the case of a remote object failing the system is able to continue by using the other remote objects available.

These are the main components that make up the middleware application. By combining all of these components it is possible to create a distributed system.

2.0 Interfaces between Components

The components of the RMI system interact by sending messages to other components. In the middleware system a number of different types of messages are sent. The following is a description of some of the interaction between components and the message that are sent between them.

2.1 Binding an Object

The first thing that must be done in the middleware application is to create a remote object and to bind the object with the registry. When an object is first created on the server that is to be remotely referenced a number of interactions between components take place.

The first thing that takes place is a new remote object is created on the server. In the case of the student list application a new ServerListImpl object is created and is assigned to a reference of the type StudentList which is the interface it implements. This object is the passed as an argument to the Naming service using the rebind method. Along with the ServerList object that is passed to the rebind method so are the address and port number of the registry, with the port number that the server will listen on for request for the object and the name that the object is to be bound to. The registry then creates a RemoteObjectReference object for the remote object and returns it to the server. In the process rebind method call, reflection is used to determine the class of the remote object. A StudentListImplSkeleton object is then created which is the skeleton object that is used on the server. When the created StudentListImplSkeleton, it initializes and creates a number of additional objects that are need. First the skeleton constructor adds the reference to the local object to be called by the skeleton. Then the skeleton creates the communication module so that communication can be accepted by the remote object. Next, it adds the remote reference to the remote reference module. After this is complete the skeleton is able to be used by the application. After the skeleton has been constructed control is returned to the rebind method of the Naming class which then returns a reference to the remote object and the application is ready to use. The figure 1 below shows the basic interaction that takes place when a remote object is created on the server.

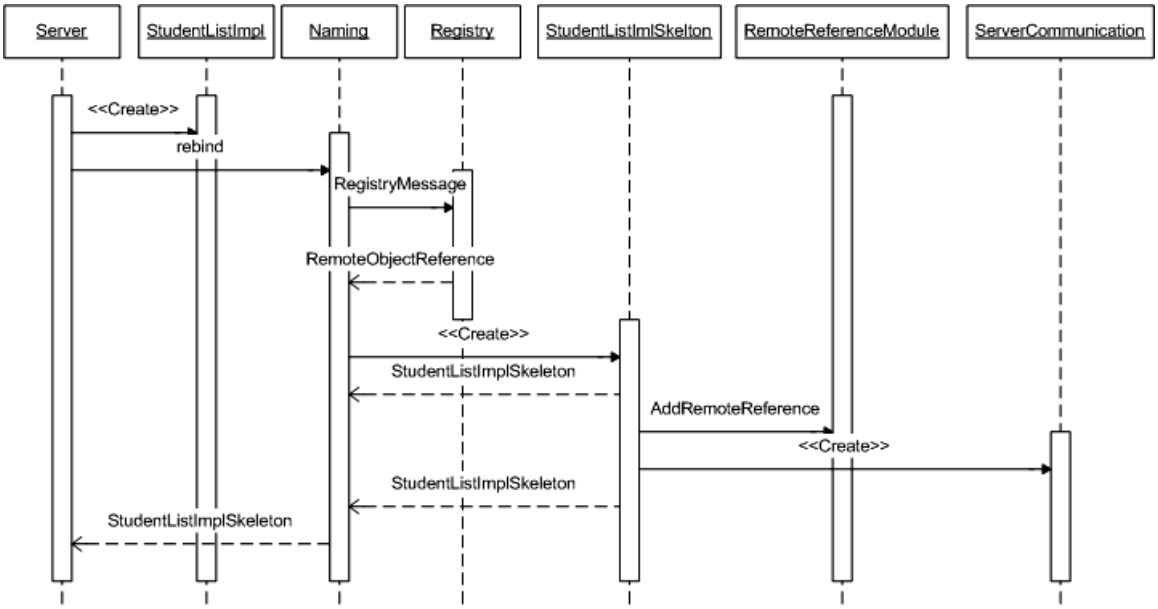

Figure 1

2.2 Looking up an Object

A similar set of events takes place when a client application wants to use a remote object. Instead of having the skeleton created by the rebind method as in the case of the server, the stub invokes the lookup method of Naming which creates the stub object. The lookup method of Naming creates a message that is sent to the Registry to lookup a remote object reference for a remote object. Once the remote object reference has found it is returned to the client which adds it to the remote object table. The stub also creates the communication module in the process. Once all of this complete the client is then able to call methods on a remote object. Figure 2 below shows the interaction between components while looking up a remote object.

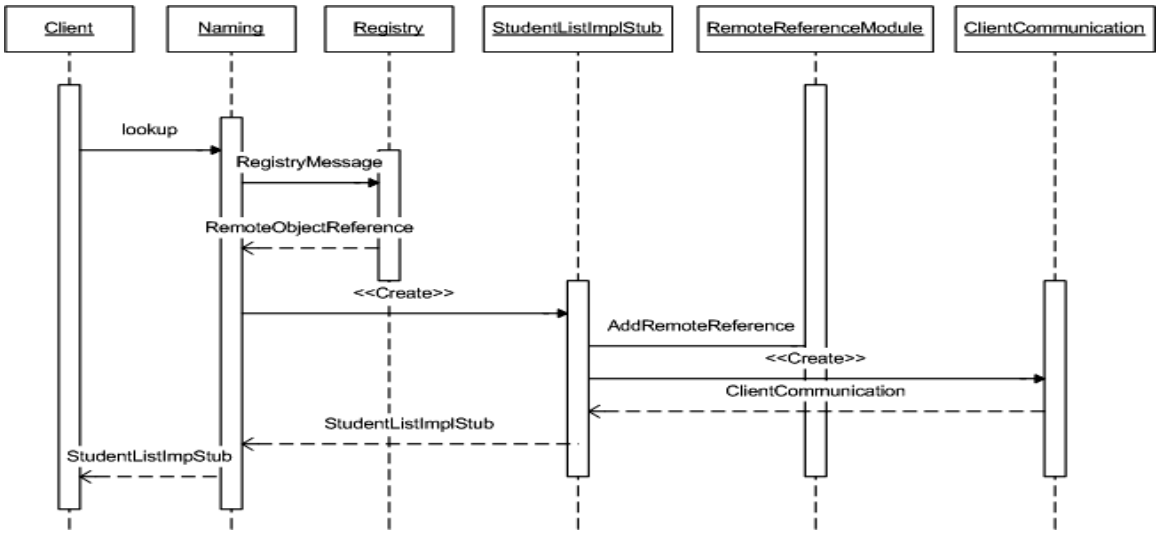

Figure 2

2.3 Invoking a Remote Method

After both the server has created an object and registered it in the registry and the client has looked up an object reference and created a stub, the client is then able to invoke a remote method on a remote object. Below in figure 4, the interaction between components that is necessary to make remote invocation on an object is shown.

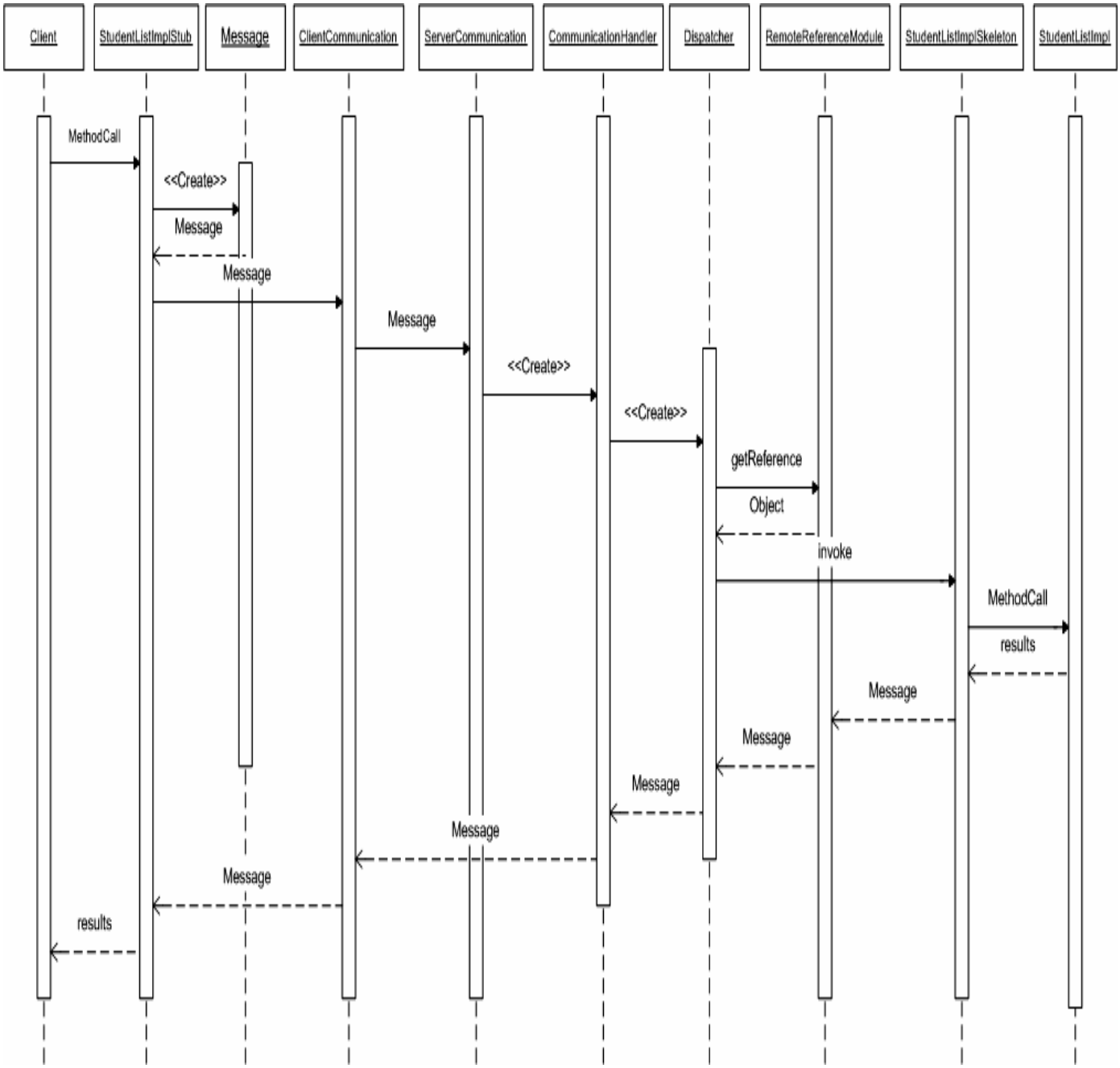

Figure 3

The client initiates the process by making a method call on the interface. In terms of the university application, the client will be able to call any of the methods that are available in the StudentList interface. The arguments to the method will then be passed to the stub to be marshaled. The stub then creates a new Message and then adds the marshaled arguments to the message. The stub then passes the message to the communication module. The remote object reference is used to locate the server to which the message should be sent. The communication then creates a new socket to the server where the remote object resides. The communication module then increments the logical clock and then adds the timestamp to the message. Then the message is sent to the server's communication module.

Upon the server's communication module receiving a message from the client, the server creates a new Dispatch object. Once the dispatch object has been created, it is passed the message received from the communication module. The dispatcher then looks up the skeleton object for a remote object reference by calling the getReference method on a RemoteRefernceModuleServer object. Once the dispatcher has the correct object, the dispatcher uses reflection to invoke the correct method on the skeleton. When the skeleton is invoked it is passed the message from the dispatcher. The skeleton then unmarshals the arguments in the message and calls the desired method on the remote object. After the remote object completes the method call, it returns the results to the skeleton. The skeleton then marshals the results into a reply message. A number of exchanges then take place between the components, until the reply message is received by the client's communication module. The client's communication module then returns the reply to the stub which unmarshals the results and returns them to the client application. At this time, the remote invocation is complete and the results of the invocation are displayed to the user.

2.4 Message Formats

The program uses two different message formats. First, there are RegistryMessage objects that are designed to be used for registry message. Second, there are Message objects that are used by the client and the server to communicate.

The first type of message used is of the type RegistryMessage. The registry message is used to bind and look up objects in the registry. The registry message is only used when communicating with the registry. Figure 4 depicts the fields that are contained in a registry message. In the cases of binding an object all of the fields are used. When an object is looked up it is only necessary to supply the name of the object and the message type.

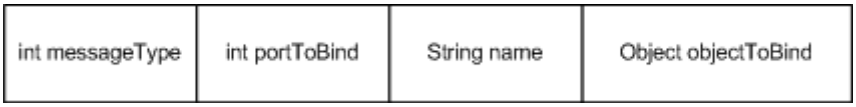

Figure 4

The second type of message that is used is a Message object, which is used for the client to communicate with the server and to invoke methods on remote objects. Figure 5

contains the fields that are present in a message object.

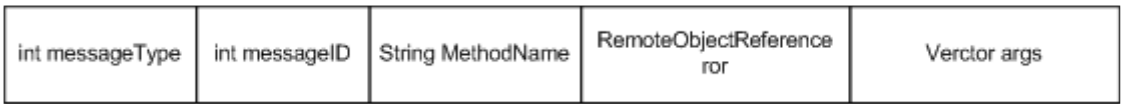

Figure 5

The RemoteObjectReference objects are important objects in the distributed system. The remote object references are the primary means of locating a remote object. Figure 6 shows the fields that are contained in remote reference.

Figure 6

2.5 Active Replication

In order to provide a fault tolerant application the middleware uses active replication. The use of active replication greatly increased the complexity of the application. In order to use active replication a front end of the replication manger is started which all request to the objects it manages will go through. The active replication manager is started as separate application that can reside on any computer. The class FrontEnd provides the active replication. When a FrontEnd object is created it uses the Naming method createFrontEnd. This sends a message to the registry which is used create an object of the type RegistryObjectMapping that holds the location of the replication manager and also holds the list of replicated object. When a new remote object is created with the same name as the replication manager it is add by the registry to the list of replicas. When an client looks up a remote object it is given the address of the replication manager so it is all transparent to the end user.

When the client invokes a method on a remote object the message is forwarded to the front end, which keeps a list of all the replicated objects that it is managing. Before sending a message the front end checks that none of the objects it is managing have died and then forwards the message to each of the replicated objects. The front end then receives the results from all of the replicated objects and then sends the results back to the client.

A number of issues have to be considered when using replication. The remote objects need to remain in a consistent state. So, if a new remote object is created after the other objects have been in use for a while the new object should be brought up to date. It is important that all remote objects have the same state. One way to keep a consistent state between objects is when a new replicated object is add to the registry, the registry sends a message to the front end. The front end will then coordinate the transfer of state to the new object. Some other concerns are that it must be detected when remote object have terminated and are no longer available. The front end periodically sends messages to the remote objects to determine if the are still available. By using active replication it allows

the system to continue to function in the case that one of the objects has failed.

3.0 Design Issues

A number of issues had to be considered when designing the middleware application. A few of the issues that had to be considered are the invocation semantics used, how mutual exclusion would be implemented and the type of fault tolerance used.

3.1 Invocation Semantics

The invocation semantics the middleware uses is "at most once" the implementation relies on TCP ability to eliminate duplicates and it ability to retransmit a message if it is lost. UPD would potentially offer better performance as compared to TCP, but it would make implementing the program much harder. For instance, if UDP was used algorithms would have to be implemented to detect that a message was lost in transit and duplicate messages would have to be filtered. Using TCP, on the other hand, takes care of all of these implementation problems.

3.2 Mutual Exclusion

It is important that any program that allows access to data by more than one thread at implement some form of mutual exclusion. For the middleware application, Java's synchronization mechanism was used to implement mutual exclusion. Anywhere that a possible race condition existed synchronization was used.

3.3 Fault Tolerance

Fault tolerance is another important issue to consider. Application can fail in many ways, so some form of fault tolerance is needed. For the project, active replication was used to provide fault tolerance. This allows physical redundancy to be used, by allowing an object to exist on multiple computers. Active replication was chosen over passive replication because it is easier to implement and it can deal with more types of failures as compared to passive replication. One benefit of active replication is it possible to use it to detect Byzantine failures. A front end is used which is a separate process that can be run in any location. The front end coordinates the sending of messages to all replicated objects. When a remote invocation is to take place the message is first sent to the front end, instead of being sent directly to the remote object. After the front end has received a message, it forwards the message to all of the replicated remote objects. The remote object then invokes the correct method and returns the results to the front end. The registry is used to determine which objects are replicated or not. When a front end object is created it makes a call to the registry to create announcing it will handle the active replication of an object. If the client uses a replicated object it will be transparent to the user it will appear as if the message was sent to a single process. One design issue that was considered is that the front end must be started before any of the replicated objects are added to the system. If this was not done there would need to be someway of determining what the correct state of an object is when the front end is started.

3.4 IDL Compiler

Another design decision was to create an IDL compiler for the stub and skeleton classes. Although it was not necessary, the class StubSkeletonGenerator was developed so the middleware would work with more than one class, it would be of limited use to have a middleware application that only worked with one class and if a new class was to be added the programmer would have to write the stub and skeletons for that class. The StubSkeletonGenerator is passed the name of the class and the interface name it implements the generator then uses Java reflection to create all of the methods that were declared in the interface. The generator then creates all of the methods and data members that are necessary for the stub and skeleton and these writes to two files for Java implementation one for the stub and another for the skeleton. The .java files are then complied by the program producing .class file for both the skeleton and stub. One problem that I came across is it may be necessary to move the compiled files after the StubSkeletonGenerator is ran to the correct location on in the file system. On a UNIX system everything worked correctly and it will replace the files if they already exist, but when using Windows it may not move the files to the correct directory if they already exist.

3.5 Use of Reflection

Another issue that I came across is reflection needs to be used in other class in the application such as the Dispatcher and the Naming class. The Dispatcher uses reflection to invoke a method on a skeleton object. The Dispatcher looks up in the remote reference module the skeleton for a remote object reference and then uses reflection to invoke the method. The Naming class also uses reflection so that the program can be ran with any type of remote object. By using reflection any class can by dynamically created by using Java reflection to invoke the constructor. The use of reflection will also allow the middleware to be used by any class to be used as a remote object that has it stub and skeletons compiled with the StubSkeletonGenerator.

3.6 Message Format

Another design issue is the format of messages to be used. Two of possible message designs exist. One is sending the message as raw byte or a second is sending objects as messages. I chose to implement the messages as serialized objects. There are different objects for the various types of messages, for instance there is a message object which is used to send the remote invocation messages and there is a registry message which is used to bind and lookup objects. The one draw back of sending the messages as object is there is a higher overhead associated with them since the must also send the class information also, but if someone else were to use the program it would be easier to understand the format of the message by looking at the interface provided by the class. It is also easier to manipulate the messages by sending them as objects.

3.7 Components not Implemented

A number of additional components could be implemented to improve the application.

Some features that are necessary for a middleware implementation were left out because of the limited amount of time given to complete the project. Some features that could later be added to the project are support for transactions by implementing a two-phase commit protocol. Also, the middleware doesn't implement any election or group communication algorithms this is another area the middleware could be improved on. Another feature that could have been implemented was the ability to recover from Byzantine failures. I decided to leave this out since to get an agreement, I would need to have a number of replicated objects, at least 3m + 1 active replicas to reach agreement on the correct message. The algorithm would not work correctly if only a couple of replicated objects were present.

There are some problems with the way that the fault tolerance is implemented. The object replicas should implement either the two-phase or three-phase commit protocol. If a message is only delivered to some of the replicated objects and it is not able to be delivered to all, there should be someway to roll back the changes. The front end simply checks that it is able to deliver a message to all of the replicated objects before it sends the message. There is no way for it to recover if an object fails after the front end has begun to send the message to all of the replicas. If the middleware was to be used in an environment that had guarantee that no failure was possible additional features would need to be implemented. Another, thing that should be implemented is a distributed snapshot algorithm. If the snapshot algorithm is developed, check pointing could be used so that objects can recover to a previous state. Another problem with the implementation is that only a single front end to the replication managers is used so there is a single point of failure. It would have greatly increased the complexity of the application, so I chose to use only one. Overall the middleware application works well, but it would have been nice to implement some of these additional features.

4.0 User Interface

To run the university application there are a number of steps that must be taken in order for the program to function correctly. First, when starting the program a choice must be made whether to the application is to use active replication or not. If active replication is used the FrontEnd object must also be started in the process of starting the application. All classes in the program are in the package nathan.middleware so to start any of the middleware applications the package must be included with the class name.

4.1 Starting the Components

The first thing that should be done is to start the registry. The registry is start by supplying the command line argument the port number. To start the registry the command below should be issued.

```
java nathan/middleware/Registry [PortNumber]
```

After the registry is started the next thing to do is to start the front end of the active replication manager. If fault tolerance is not desired this step can be skipped. To start the front end the command below should be issued. The name of the replicated object is the name that will be used to locate a remote object by in the registry. If replication is used

the front end to the replication managers should be started before any remote objects are created. The registry port and IP address are used to make a connection to the registry and the front end port is the port that the front end will run on.

java nathan/middleware/FrontEnd [Name of Replicated Object] [IP Address] [Registry Port] [Front End Port]

Once the front end has been started it is now necessary to start the servers of the remote objects. In the case that the front end is start the user may start multiple servers for an object using the same name. To start the server the following command should be issued. The server port is the port number that the server will run on.

java nathan/middleware/Server [Name of Object] [Registry IP] [Registry Port] [Server Port]

After these three components have been started all that is left to do is to start the client application. The client application can be started by enter the following command.

java nathan/middleware/Client [Name of Object] [Registry IP] [Registry Port]

4.2 Running the Client Application

Once the client has been started the user can use the program to invoke the remote methods on the student list. The University application uses console to run the application as opposed to using a GUI. When the university application starts the user has a number of choices available to them. To begin, the user can add and remove students from the remote list. Also, the user can get or set a students grade. Another option available to the user is to list all students and to find a specific student. Finally, a user of the can also lookup an additional StudentList object to access. Below figure 7 shows the menu of available options to the user when the program is started

Student List Menu

1 – Add new student
2 – Delete a student
3 – List all students
4 – Find a student
5 – Set a students grade
6 – Get a students grade
7 – Lookup object
8 – Exit application

Figure 7

First, “Add a new student” creates a new student object locally and then sends it to the server to be added to the student list. When the add student option is chosen the user enters in the required information such as the user name and age. Second, the “Delete a student” option allows the user to enter in the id of the student to delete and then the results are returned to the user, if the student was able to be deleted or not. If an invalid id is entered it will be displayed to the user that it was unable to delete a student. Third, the “List all students” command returns to the user a list of all the students that are stored remotely on the server. Forth, “Find a student” option allows the user to enter in the id

number of the student they wish to find. If the student is contained in the remote list the student information will be displayed on the client. Fifth, "Set a students grade" allows the user to get and set the grade for a particular student. Set grade allows the user to set the grade for the grades that are stored for the students. When the user has completed setting the grade to the grades is sent to the server and the student information is updated. Sixth, "Get a students grade" is executed by the user entering in the student id of the student that they wish to have grade displayed. Get grade then returns all the students grades and there grade point average. Seventh, lookup object locates an additional remote object to use in the application. The user then can access multiple remote objects if the lookup succeeds. When the user has multiple references to remote objects the user is then prompt a message to select the object that they wish to access. This is done by the user typing in the name of the object from the list displayed. Finally, "Exit application" performs any necessary clean up need by the program such as closing connections and then terminates the application.

5.0 Task List

I chose to work on the project individually, so I completed all of the work on the project myself. I chose to do it myself, because I would learn more from the experience as opposed to only do half of the work. Overall, this was a good assignment. Before the class, I always want to learn to use RMI or CORBA and this gave me a chance to gain an understanding of how RMI works and to implement a scaled down version of Java RMI.

6.0 References

Arnold, Ken, and James Gosling, and David Holmes. The Java Programming Language 3rd ed. Palo Alto, CA: Addison-Wesley, 2000.

Coulouris, George, and Jean Dollimore, and Tim Kindberg. Distributed Systems: Concepts and Design. 3rd ed. Wokingham: Addison-Wesley, 2001.

Darby, Chad, et al. Beginning Java Networking. Birmingham, UK: Wrox Press Ltd, 2001.

Farely, Jim. Java Distributed Computing. Sebastopol, CA: O'Reilly & Associates, 1997.

Grand, Mark. Java Enterprise Design Patterns: Patterns in Java Volume 3. New York: John Wiley & Sons, Inc., 2002.

Grosso, William. Java RMI : Designing & Building Distributed Applications. Sebastopol, CA: O'Reilly & Associates, 2002.

Horstmann, Cay, S. and Gary Connell Core Java Volume II-Advanced Features. 4th ed. Palo Alto, CA: Prentice Hall, 2002.

Pitt, Esmond, and Kathleen McNiff. Java.rmi: The Remote Method Invocation Guide. Great Britain: Addison-Wesley, 2001.