Peer-to-peer Implemented

Transparent Access to Remote Services

Jeson Martajaya (jeson@cs.utexas.edu)

CS379HW Honor Thesis Spring 2003

Supervisor : Dr. James C. Browne

Co Supervisor: Dr. Shyamal Mitra

May 16, 2003

http://www.cs.utexas.edu/users/jeson/JRDL

ABSTRACT

The emerging trend of distributed computing gives birth to various mechanisms of providing access to remote services. These services may be in form of dynamic libraries or custom-tailored applications. This paper suggests a simple and transparent mechanism to access these services which requires no central point that allows it to be incorporated into peer-to-peer systems. This mechanism integrates discovery and invocation of the services into one solution. The solution is implemented as an application suite named Java Remote Dynamic Loader (JRDL). This application may also be used to accommodate other remote services as well.

1. INTRODUCTION

Transparent remote procedure calls for specialized services such as remote file systems have created fundamental paradigm shifts in the use of desktop computers. Transparent access to general remotely hosted services is fundamental to the future development of the Internet. The future model for development of application programs is that a developer, instead of either developing custom implementations of common services or locally installing and configuring existing implementations of common services, will access those common services through the Internet.

There are several approaches to the implementation of access to remotely hosted services. Broker systems such as CORBA [1], DCOM [2] and Enterprise Java [3] are one approach. Programming systems such as Java provide Remote Method Invocation [4] systems as a part of their runtime systems. Web services [5] is an emerging industry standard for packaging common services for delivery over the Internet and for accessing services which are packaged in conformance to Web Services standards. Web Services is a standard means of specification for remote procedure call systems of general semantics. Grid Services, which are specialized Web Services [6,7], are another approach to standardizing access to remote facilities and services.

All of these systems require modifications to programs and require application developers to acquire significant skills not related to application development and are relatively "heavyweight" procedures. Services must be encapsulated with rather complex "wrappers" and registered in a central database of available services. Those wishing to use these services must utilize rather complex procedures for discovery of services and for accessing of services. While tools for assisting with both enabling a service for web access and accessing a service using these standards are emerging, the overhead is still non-trivial and impedes utilization of services over the Internet.

There is a need for a simple transparent mechanism for making services available over the Internet and for discovering and accessing these services. This paper presents one simple transparent mechanisms for these functions. It is implemented in Java and uses mechanisms provided by Java but encapsulates these mechanisms to render them transparent to the application developer. The implementation integrates discovery of general services based on a symmetric "peer to peer" broadcast mechanism [8].

2. APPROACH

The solution involves a symmetric architecture which renders an agent node requesting a service be able to provide a different service for another requestors. However, we separate the role “client agent”, “server agent”, and “other agent” to better illustrate our approach. “Client agent” represents a node in a network that requests a service, “server agent” represents a node that provides the service, and “other agent” represents a node that does not provide the service. All agents are connected in a virtual broadcast network.

Our approach involves two clearly defined stages a) the discovery stage and b) the invocation stage.

2.1 Discovery stage

Discovery happens when a client agent does a query requesting for an arbitrary service over a public channel. Then, all agents receive the query. Each of these agents makes an attempt to find the requested service on its physical node. When an agent finds the requested service, it becomes a server agent and sends a reply back to the client agent through a private channel. When the search for the requested service fails, the agent becomes “other agent” and does not respond.

We are following a peer-to-peer approach to satisfy this stage. This approach has no central point of failure. Contrasting to the existing Web Services model, we use a registry-free mechanism. A server agent searches for a service at runtime, right after the service is requested. This mechanism enables a service be published anytime, as long as it is placed in an accessible location. It also eliminates the overhead of the service registration process.

The query routing for this stage is performed at application level. This mechanism allows the query to be routed beyond boundaries of local area network. Associative Broadcast [8] merges this application level routing and reliability through Lightweight Reliable Multicast Protocol (LRMP [13]). It encapsulates low-level multicast routing mechanism into a high-level application layer. However, other Java-based implementation of distributed hash tables such as FreePastry Scribe could also be used to implement the query router.

2.2 Invocation stage

After the client agent receives the reply sent by the server agent, it prepares itself to invoke remote procedure calls on the server agent. The calls are initiated by the client agent, but the computations are performed by the server agent. Network traffic in this stage is no longer through application level routing, but instead through direct socket-to-socket connection.

The invocation stage begins when a developer program on the client agent invokes the requested service. Before this stage begins, the program must prepare a description of the service. Then, service search will be performed based the description. The program cannot invoke unprepared service.

Following the object-oriented model for remote procedure invocation, the instantiation of the requested service is recorded both in the client and server domain. The image of the requested service object resides in both domains. Each call that modifies the instantiated service object is recorded in both domains as well.

Calls are mediated through a transparent mechanism. A developer only needs to have a knowledge of the service she*) requests, such as the class name of the service, the constructor information of the service, and the method names and their input types. She need not be aware the calls that she invoked are remote procedure calls that need special handling.

To satisfy the transparency requirement, a uniform interface to load services is needed. This interface has to be general enough to accommodate different services, regardless of their structure. Moreover, this interface has to follow a static structure that is understandable by the user. The only mutable part of this interface should be its data value. Therefore, the structure information of a requested service must be passed as data values.

**) For conciseness "he/she" is referred as "she" throughout the paper*

3. IMPLEMENTATION

The Java programming language is well known for its standard structure that is easy to comprehend. Moreover, Java supports a Class Loading feature that enables a class to be loaded at runtime instead of at compile time. This feature, contained in the Java.lang.reflect package, is able to treat structure information of a particular class as data values. It is a perfect match to fulfill the uniform interface for transparency requirement mentioned above.

Java also has a standard feature[1] to perform remote procedure calls, Remote Method Invocation (RMI) [4]. This mechanism involves two proxy classes, RMI skel and RMI stub, that act as representatives of remote agents.

An application suite named Java Remote Dynamic Loader (JRDL) which implements the approach and satisfies the requirements given in Chapter Two has been developed. JRDL includes two components: a) a runtime daemon named JRDLCarrier and b) a service interface named JRDLObject. The runtime daemon serves as i) query router/replier, and ii) service loader. The service interface serves as a bridge between the developer and the remote service that she requested.

3.1 Design of JRDL

The design of JRDL application works as follows.

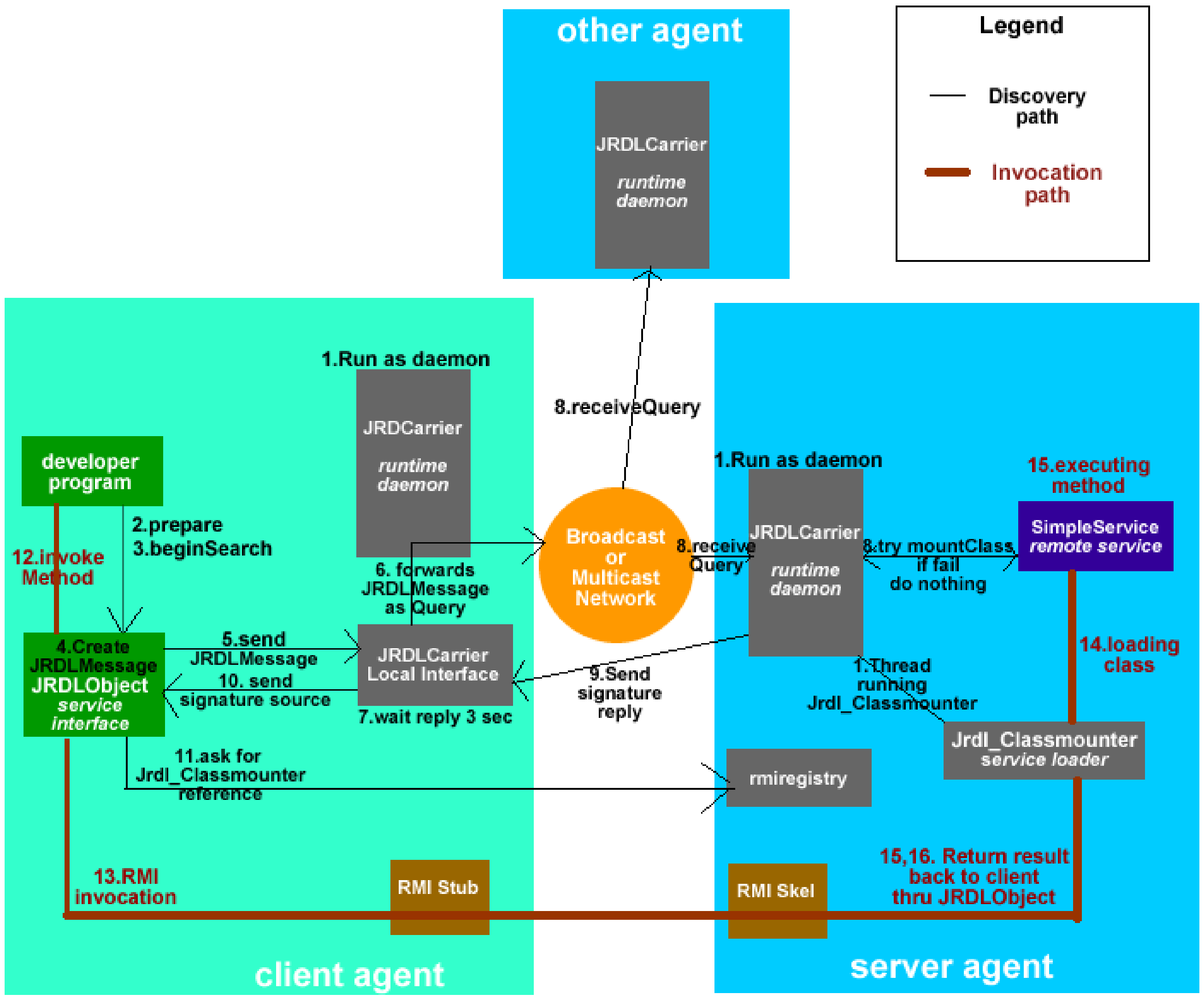
We have the runtime daemon running on a client agent. The developer prepares a service interface containing the information of the remote service she requested in her source code. The service interface then packages the information in a customized serializable data type. The developer then invokes a built-in method in her algorithm to perform the search process, which marks the beginning of the discovery stage. After this method is invoked, the packaged specification for the desired service is sent to the runtime daemon through a specific local port to be handled as a query request. The runtime daemon sends this query to the network, opens a socket to accept incoming reply, and waits within a certain time period. If no reply is received, the runtime daemon signals the developer program telling her that the requested service is not available.

A runtime daemon on the other peers receives the query and tries to load the requested service. If it does not find the service, it does not reply back to the requester. If the runtime daemon on some peers receives the query and is able to load the requested service, this agent becomes a server agent and sends a reply back to the client agent directly to the socket that the client agent has prepared. The signature contained in the reply is matched to the signature stored in the runtime daemon. If matched, the source address of the reply packet is then passed to the service interface as the server address for Remote Method Invocation. This step marks the final step of the discovery stage.

Every time the developer program invokes a remote procedure call through the service interface, the invocation stage occurs. This stage follows the standard practice of Java RMI calls. The name of the service method is passed along with its parameters. The service interface then forwards this information to the local RMI stub. The RMI stub forwards this information to the remote RMI Skel, which forwards it to the remote service loader. The service loader then loads the image of the service, performs requested computation, and returns the result value back to the RMI Skel. The RMI Skel forwards the return value to the RMI Stub, which forwards it to the service interface, and finally ends at the developer program.

3.2 Workflow of the Design

To better illustrate the design narrative above, consider the following scenario. Suppose there is a trivial remote service named “SimpleService” that performs a simple computation by multiplying and dividing an input number. Steps below give the workflow of how the discovery and invocation processes are executed. Underlined word or phrase represents a single component.



JRDL Workflow Schema

1. The runtime daemon JRDLCarrier runs initially on each agent.
2. The developer program prepares a service interface JRDLObject that contains requested service class named "SimpleService", and its corresponding constructor parameter types and constructor parameter values, and also the method names and their parameter types.
3. The client invokes beginSearch method. Discovery stage begins.

4. JRDLObject class bundles the service information into a JRDLMessage query request.
5. JRDLObject sends the packaged query request to the local interface of the runtime daemon, JRDLCarrierLocalInterface (port 11507-11607).
6. JRDLCarrierLocalInterface forwards the received JRDLMessage as a query to the network.
7. JRDLCarrier's Local Interface opens a socket for incoming reply and waits for 3 seconds.
8. JRDLCarrier on other agents receives the query, tries to load the service specified in the incoming JRDLMessage, fails, and does nothing. On the other side,

8. JRDLCarrier on one agent receives the query, tries to load the service specified in the incoming JRDLMessage, succeeds, sends a reply containing a signature string to JRDLCarrierLocalInterface socket, and becomes a server agent.

9. JRDLCarrierLocalInterface receives the reply and matches the signature string.
10. JRDLCarrierLocalInterface detects the source address of the signature, and passes it to the service interface JRDLObject.
11. JRDLObject connects to the rmiregistry daemon running on the server agent to establish an RMI connection to the service loader Jrdl_Classmounter.

This concludes the discovery stage. Below are the steps for invocation stage.

12. The client invokes a pre-prepared remote service method.
13. The service interface JRDLObjects treats the method as a Remote Method Invocation, and connects it to the established service loader Jrdl_Classmounter through RMI Stub and RMI Skel proxy classes.
14. The Jrdl_Classmounter receives the RMI call and loads the remote service SimpleService.
15. SimpleService, executes the remote service's requested method and performs necessary computation, and returns the result back to service loader Jrdl_Classmounter.
16. Jrdl_Classmounter returns the result back to the service interface JRDLObject
17. JRDLObject returns the result back to the developer program
18. The developer program receives the result.

4. TESTING

The simple scenario given in workflow design above is implemented in two small Java programs a)client.java, which serves as the developer program, and b)SimpleService.java, which serves as the remote service.

4.1 Remote Service : SimpleService

The content of the remote service follows a regular Java program. This SimpleService service is designed to reflect object instantiation, object modification, and method computation at a very simple level. It has a constructor, a mutable field, an accessor method, a compute method, and an extra method that is not utilized by the developer program. Words with bolded fonts are the service information that is used for service discovery.

Here is the annotated source code of SimpleService.java

```
//no need to import anything here

public class SimpleService
{
        //mutable field
        Integer result = null ;

        //One parameter constructor
        //gives initial value of the mutable fields
        public SimpleService(Integer r){
                result = r ;
        }

        //Accessor method
        public Integer getResult(){
                return result ;
        }

        //compute method
        public Integer compute(Integer i){

                //performs simple computation
                int num = i.intValue()*916432/2634*result.intValue();
                Integer number = new Integer(num) ;

                //modifies this instantiation
                this.result = number ;

                //returns result of computation
                return number ;
        }

        //extra unused method. To show not all the methods
        //in the service has to be specified in developer program.
        //Instead, need to specify used ones only.
        public String notCalled(){
                return "Don't call me" ;
        }
}
```

4.2 Developer Program

4.2.1 client.java (non-JRDL version)

The developer program utilizes the remote service. It initially instantiates the remote service with value 7, and performs the computation with value 19. The invoked service methods are getResult and compute, while notCalled will never be called in the program. The numbers are chosen arbitrarily for simplicity.

In non-JRDL version, here is the source code:

```
public class regclient
{
  public static void main(String args[]) throws Exception{
    //instantiate service with 7
    SimpleService service = new SimpleService(new Integer (7)) ;

    //show the result
    Integer result = (Integer) service.getResult() ;
    System.out.println("value before compute="+result.intValue()) ;

    //perform computation with 19
    service.compute(new Integer(19)) ;

    //show the result again
    result = (Integer) service.getResult() ;
    System.out.println("value after compute="+result.intValue()) ;
  }
}
```

4.2.2 JRDL Client Usage Model

In JRDL version, client.java needs to be modified. All tokens that resemble structure of SimpleService have to be translated into data values.

A general practice for JRDL clients includes two sections, a) preparation, and b) invocation. In the preparation section, the developer prepares all service information including the method names she will invoke later in her program. This information will then be used to search the desired service.

Below is an outline that illustrates the general practice of a JRDL client program.

```
JRDLobject.Constructor(String classname, String[] param_types,
                       Object[] param_values)
JRDLobject.prepareMethod(String methodname, String[] param_types)
JRDLobject.prepareMethod(String methodname, String[] param_types)
```

Preparation Section

```
JRDLobject.beginSearch(); //Discovery stage begins here
```

```
JRDLobject.invoke(String methodname, Object[] param_values) ;
JRDLobject.invoke(String methodname, Object[] param_values) ;
JRDLobject.invoke(String methodname, Object[] param_values) ;
```

Invocation Section

4.2.3 client.java (JRDL version)

Back to client.java, here is the annotated source code following the usage model above.

```
//requiring the remote service is mediated by service interface
import UTCS.JRDL.JRDLObject;
import UTCS.JRDL.LibraryNotFoundException;
import UTCS.JRDL.InvocationErrorException;

public class client{
public static void main(String args[])//handles service not found signal
  throws LibraryNotFoundException, InvocationErrorException{
    /*Begin preparing for requesting the service*/
    //Preparing two-parameter constructor
    String[] cp = {"Integer"} ;
    Object[] cv = {new Integer(7)} ;
    JRDLObject jrdlobj = new JRDLObject("SimpleService", cp, cv) ;

    //Preparing method getResult()
    jrdlobj.prepareMethod("getResult", null) ;

    //Preparing method compute(Integer)
    String[] methodpartypes = {"Integer"} ;
    jrdlobj.prepareMethod("compute", methodpartypes) ;

    /*end preparation section*/

    /*begin discovery stage*/
    jrdlobj.beginSearch() ;
    /*end discovery stage*/

    //invoking getResult()
    Integer result = (Integer) jrdlobj.invoke("getResult",null) ;
    System.out.println("value before compute="+ result) ;

    //invoking compute(new Integer(19))
    Object[] parameter = {new Integer(19)} ;
    jrdlobj.invoke("compute", parameter) ;

    //invoking getResult() again
    result = (Integer) jrdlobj.invoke("getResult",null) ;
    System.out.println("value before after="+ result) ;
}
}
```

4.3 Result

The results of these two different versions of client.java are compared. They show identical outputs. The SimpleService service is located at owlbreath.cs.utexas.edu, and the client is located at oban.cs.utexas.edu

Here is the result of running client.java, non-JRDL version

```
oban.cs.utexas.edu> java client
value before compute=7
value after compute=46270
```

Here is the result of running client.java, JRDL version

```
oban.cs.utexas.edu> java -classpath ./:JRDL.jar client
LOCAL=Library Found from:owlbreath.cs.utexas.edu
SERVERHOST=owlbreath.cs.utexas.edu
```

```
URL=rmi://owlbreath.cs.utexas.edu/JRDL
value before compute=7
value after compute=46270
```

We can see the value before and after compute are the same. The developer does not need to specify the location of the service. She does not need to even aware that Remote Method Invocation is occurred underneath. These processes are completely transparent.

4.4 Limitation

The given remote service example, SimpleService, uses Integer wrapper to pass int values. This is one limitation of the current implementation of JRDL. The signature of remote service methods cannot have primitive data types such as int or double. Instead they must use Object-based primitive wrapper (e.g java.lang.Integer, java.lang.Double, and java.lang.Long).

This limitation arises because in low-level, JRDL exploits the Object Stream transfer. One main requirement of the Object Stream is that, all parameter objects that are passed must implement the serializable interface (java.io.Serializable). Primitive data types do not satisfy this requirement. However, primitive arrays such as double[] or int[] do not fall into this limitation because they are Object.

This limitation also applies for custom data types. The developer must implement the serializable interface for her data types.

5. TYPICAL USAGE SCENARIO

JRDL may be used to support a scientific work. We have implemented a Data Fitter application that runs using JRDL as its service seeker. This application requires three services, given a series of X and Y values: a) LeastSquare, that performs a least square fit, b) Lagrange, that performs Lagrange fit, and c) NatCubic, that performs Natural Cubic Fit. These services also require Jama[28,29] and Jamlab[29], two publicly available Java math libraries. We placed the services together with the required libraries in a physical node named owlbreath.cs.utexas.edu. We placed the client program in another node named oban.cs.utexas.edu. We also ran JRDL in other nodes as well, without the services.

5.1 Client source code

Our client source code works as follows. First, it has some X and Y values stored in two arrays as doubles. These arrays are then passed to each of these services. The resulted values are formatted and printed to the screen. There are three search processes occurred throughout the program. Each search process is for an instantiated service.

Here is the annotated source code.

```
import DataFit.Cubic ;
import java.text.DecimalFormat ;
import UTCS.JRDL.JRDLObject ;
import UTCS.JRDL.LibraryNotFoundException ;
import UTCS.JRDL.InvocationErrorException ;
/**
 * This class demonstrates a typical usage scenario of JRDL
 * @version 1.0, May 16, 2003
 * @author, Jeson Martajaya
 */
public class DataFitClientJRDL
{
    static String format(double value)
    {
        DecimalFormat myFormatter = new DecimalFormat("000.000000") ;
        return myFormatter.format(value) ;
    }

    public static void main(String[] args) throws LibraryNotFoundException,
InvocationErrorException
    {
        double[] x = {127.32, 308.28, 521.76, 475.92, 267.01} ;
        double[] y = {228.02, 404.32, 687.12, 512.10, 328.01} ;
        Integer order = new Integer(2) ;
        String[] paramtypes = {"double[]","double[]","Integer"} ;
        Object[] paramvalues = {x,y,order} ;

        //Preparing LeastSquare service
        //LeastSquare has a default constructor, and a method that takes
        //two double arrays and one integer object
        JRDLObject LS = new JRDLObject("DataFit.LeastSquares",null,null) ;
        LS.prepareMethod("processRequest",paramtypes) ;
        LS.beginSearch() ;

        //Invoking LeastSquare service
        double[] ls = (double[]) LS.invoke("processRequest",paramvalues) ;

        //Preparing Lagrange service
        //Lagrange has a default constructor, and a method that takes
        //two double arrays and one integer object
        JRDLObject L = new JRDLObject("DataFit.Lagrange",null,null) ;
        L.prepareMethod("processRequest",paramtypes) ;
        L.beginSearch() ;

        //Invoking Lagrage service
        double[] l = (double[]) L.invoke("processRequest",paramvalues) ;

        System.out.println("      x        y          LeastSquare   Lagrange");
        for (int i=0; i<ls.length; ++i)
        {
            System.out.println(format(x[i])+"\t"+format(y[i])+"\t"+
            format(ls[i])+"\t"+ format(l[i])) ;
        }

        //Preparing NatCubic service
        //NatCubic has a default constructor, and a method that takes
        //two double arrays and one integer object
        JRDLObject NC = new JRDLObject("DataFit.NatCubicLib",null,null) ;
        NC.prepareMethod("processRequest",paramtypes) ;
        NC.beginSearch() ;

        //Invoking NatCubic service
        Cubic[][] C = (Cubic[][]) NC.invoke("processRequest",paramvalues) ;

        System.out.println("\nNatural Cubic Fit") ;
        for (int j=0; j<C.length; ++j)
        {
            for (int k=0; k<C[j].length; ++k)
                System.out.println(C[j][k]) ;
        }
    }
}//END class DataFitClientJRDL
```

5.2 Limitation

In developing the real application, we have found further limitation. We found Associative Broadcast limits the packet to be sent to the multicast network. In finding the service, JRDL includes an array of objects to be the service's constructor parameter. Because these objects are instantiated objects, their size may be large enough that could not be accommodated by Associative Broadcast.

6. CONCLUSION

This paper gives one solution to implement a peer-to-peer based transparent access to remote service. The solution, implemented as an application suite named Java Remote Dynamic Loader (JRDL) is now in working and usable condition. It represents an approach of having the integration of discovery and invocation of remote service into a programming language at a very simple and developer-friendly level.

We conclude that transparent access to remote services boils down to remote procedure call and dynamic class loading. These two functions are the core backbone that becomes the foundational base in building the solution. Both Java and C have these capabilities, but we found at the given time of this thesis, building with Java is much simpler. Dynamic class loading in C is much more difficult because it involves modifying the internals of dlfcn library[14] in handling ELF binaries[15]. We hope our solution will be brought into the C world as well, because there are countless dynamic libraries available in UNIX systems for potential remote services.

6.1 Future work

There are numerous possibilities in extending this solution. Some of the known ones are:

6.1.1 Security

As of the date when this paper is published, there is no security implemented in JRDL. The runtime daemon runs with Java security policy disabled. It allows an attacker to exploit the RMI server running on the agents to invoke malicious code. Also, there is no signature checking in accepting server agent’s reply, thus an adversary may mimic a server agent and provide a malicious service by replying to a client agent.

6.1.2 Lifting limitation

JRDL bases its network transaction on object exchange. One limitation of Java Language is that, primitives are not objects, although they may have object wrappers. Therefore, one potential project would be embedding the automatic Object-to-primitive conversion to JRDL, so the user may include primitives in her service.

7. RELATED WORK

There are several other efforts today to make transparent access to remote services available. In this chapter, we examine Web services, GridRPC, and JavaCog and contrast them to JRDL in fulfilling the purpose.

Web services [5]

A web service is an application that is self-describing and invokable from the web. It is geared toward business applications that involve some accounting for usage of service. Therefore, its remote services are language-independent, unlike JRDL whose client and remote service are dependent on the Java language.

Web service requires its remote services to register themselves to a central directory service using a standard protocol [9]. This requirement introduces three weaknesses a) central point of failure, b) overhead in publishing a service, and c) overhead in maintaining a published service.

Web service's single point of failure lies on its central directory. There would be multiple service providers registering their services to a public directory. A client first searches its desired service in this directory. When it finds the desired service, the client goes to the provider and starts using the service. However, when a failure to this directory occurs, the client is no longer able to find a service, rendering the published services unsearchable.

There is also an overhead in publishing a service. First, the provider needs to create the service. Second, it needs to write a description of the service using a standard language named Web Service Description Language (WSDL). Third, the provider needs to write a registry entry following a standard named Universal Description, Discovery, and Integration (UDDI) [9]. And finally, publish this registry entry to the central server. The overhead of learning and writing two extra standards impedes the publishing process.

Maintaining the service also includes overhead. The directory typically has stale data. It needs to be frequently updated. Every time a remote service is published, modified or removed, the registry needs to be informed. Otherwise, a registry entry in the central directory may not reflect the actual service.

Grid RPC [10]

GridRPC is a standard API designed for conducting remote procedure calls for grid computing. It is similar to MPI [16], the standardized message passing interface for parallel programming. This standard also supports asynchronous coarse-grained parallel tasking. Its standardized API includes a) initializing and finalizing functions b) remote function handle management functions c) GridRPC call functions d) asynchronous GridRPC control functions e) asynchronous GridRPC wait functions f) error reporting functions, and g) argument stack functions. Contrasting to JRDL, GridRPC is an implementation-independent standard that is not specifically aimed to integrate discovery and invocation. Two evaluated implementations of GridRPC are Netsolve and Ninf-G.

Netsolve involves a central Netsolve agent for service address resolution. The agent serves as the central registry for remote services, network monitor, load balancer, and authentication gateway. It incorporates the Network Weather Service [17] and the Globus Heart Beat Monitor [18] for network monitoring, its own load-balancing algorithm to select the most suitable server to execute a service, and Kerberos [19] for authentication. This implementation has a central point of failure.

Ninf-G on the other hand, does not explicitly have a central agent. Ninf-G is designed with simplicity as its focus. Therefore, it does not provide fault detection, recovery, nor load balancing. Ninf-G relies heavily on Condor [24] platform for providing these functionalities. It also utilizes Globus Security Infrastructure (GSI [25]) for its authentication. It utilizes Globus Resource Allocation Manager (GRAM [26]), Globus Monitoring and Discovering Service (MDS [27]), and Globus-I/O for low-level remote procedure calls. Therefore, it has some considerable infrastructure overhead.

Java Commodity Grid Kit [11]

The Java Commodity Grid Kit, abbreviated as Java CoG, is a middleware for accessing a Globus-based [12] grid from the Java framework. It has extensive

API libraries to utilize various Globus capabilities. The combination of Java Cog and Globus may emulate the purpose of JRDL in integrating discovery and invocation of remote services. Java CoG has an interface to utilize Globus MDS for discovering the remote service. It also has an interface to utilize globusrun to invoke a remote application.

Nevertheless, Globus positions itself as the general tool for authentication, connectivity, data transfer, and resource management for grid computing. Therefore, using Globus to emulate JRDL purpose may involve extensive overhead such as establishing certificate credentials, having daemon that needs super-user intevention running in the system, and having separate cache directory.

8. ACKNOWLEDGEMENT

I would like to deliver my utmost gratitude for Dr. James C. Browne and Dr. Shyamal Mitra, who become mentors, parents, and friends while working on this project. Also for Kevin Kane, who gives me the permission to use his Associative Broadcast project and other unexplainable numerous help. For Peter Druschel, who give me insights on the possibility of incorporating FreePastry Scribe to JRDL.

9. REFERENCES

[1] S. Vinoski, “CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments”, *IEEE Communications Magazine*, Vol. 14, No. 2, February, 1997

[2] Microsoft Corporation, “MSDN DCOM Technical Overview”, November 1996. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_ dcomtec.asp

[3] Sun Microsystems, “Developing Java TM 2 Platform, Enterprise Edition (J2EE) Compatible Applications”, January 2001, URL: http://java.sun.com/j2ee/white/j2ee.pdf

[4] Sun Microsystems, “Java RMI white paper”, URL: http://java.sun.com/products/javaspaces/whitepapers/rmipaper.pdf

[5] B.Alexander, “Oncoming Wave with Web Services”, *Macronimous.com*, URL: http://www.macronimous.com/resources/web_services_and_standards.asp

[6] I.Foster, C.Kesselman, J.M.Nick, S.Tuecke, “The Physiology of the Grid”, July 6, 2002

[7] J.S.Brown, J.Hagel, “Service Grids: The Missing Link of Web Services”, 2002

[8] J.C.Browne, K.Kane, H.Tian, "An Associative Broadcast Based Coordination Model for Distributed Processes"

[9] “UDDI Faqs” URL: http://www.uddi.org/faqs.html

[10] H.Nakada, S.Matsuoka, K.Seymour, J.Dongarra, C.Lee, H.Casanova, “GridRPC: A Remote Procedure Call API for Grid Computing”, July 2002

[11] G.Laszewski, J.Gawor, P.Lane, N.Rehn, M.Russel “Features of the Java Commodity Grid Kit”, July 23 2001.

[12] “The Globus Project” URL: http://www.globus.org/

[13] “Light-weight Multicast Reliable Protocol”, URL: http://webcanal.inria.fr/lrmp/

[14] “dlfcn.h – Dynamic Linking, the Single UNIX Specification”, URL: http://www.opengroup.org/onlinepubs/007908799/xsh/dlfcn.h.html

[15] K.Boldyshev, “Startup State of a Linux/i386 ELF Binary”, 2000, URL: http://linuxassembly.org/articles/startup.html

[16]”MPI – The Message Passing Interface (MPI) Standard”, URL: http://www-unix.mcs.anl.gov/mpi/

[17] R.Wolski, "Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service". In *6th High-Performance Distributed Computing Conference*, August 1997.

[18] P.Stelling, I.Foster, C.Kesselman, C.Lee, and G. von Laszewski, "A Fault Detection Service for Wide Area Distributed Computation*."* In *7th IEEE Symp on High Performance Distributed Computing*, pages 268-278, 1998.

[19] "Kerberos – The Network Authentication Protocol", URL: http://web.mit.edu/kerberos/www/

[20] I.Stoica, R.Morris, D.L.Nowell, D.R.Karger, M.F.Kaashoek, F.Dabek, H.Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications." To Appear in *IEEE/ACM Transactions on Networking.*

[21] D.Mazières, "Self-Certifying File System", Ph.D Thesis, May 2000, URL: http://www.fs.net/sfs/@amsterdam.lcs.mit.edu,bkfce6jdbmdbzfbct36qgvmpfwzs8exu/pub/pubs.html

[22] "Pastry - A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications" URL: http://www.cs.rice.edu/CS/Systems/Pastry/

[23] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, "SCRIBE: A large-scale and decentralised application-level multicast infrastructure", In *IEEE Journal on Selected Areas in Communication (JSAC)*, Vol. 20, No, 8, October 2002.

[24] R.Raman, M.Livny, and M.Solomon. "Matchmaking: Distributed resource management for high throughput computing", In *Proc. Of HPDC-7*, 1998

[25] "Globus – Grid Security Infrastucture", URL: http://www.globus.org/security/

[26] "Globus Resource Allocation Manager", URL: http://www-unix.globus.org/api/c-globus-2.2/globus_gram_documentation/html/index.html

[27] "Globus Monitoring and Discovery Service", URL: http://www.globus.org/mds/

[28] "JAMA: A Java Matrix Package", URL: http://math.nist.gov/javanumerics/jama/

[29] S.Palu "Java in Science: Data Interpolation and Extrapolation Using Numerical Methods of Polynomial Fittings", *Developer.com*, URL: http://www.developer.com/java/article.php/762441

APPENDIX A - TUTORIAL

This tutorial runs the test given in Chapter 4. It also gives the reader a feel on how JRDL works. It makes JRDL system that has at least three peers, the "client agent", the "server agent", and the "other agent". The server agent provides a remote service "SimpleService". As of JRDL-1.0, these agents need to be connected in a multicast network, because the query router for this version utilizes Associative Broadcast, which requires multicast network.

The tokens after '$' sign are the command that need to be executed. The token before '$' sign indicates a UNIX prompt, which specifies the location and the account needed to execute the command.

Making the "other agent"

```
user@other> tar zxvf JRDL-1.0.tar.gz
user@other> cd jrdl_runtime
user@other> make other
user@other> cd ../jrdl_runtime-other
user@other> make run
```

These steps create a source directory jrdl_runtime and build directory jrdl_runtime-other. Jrdl_runtime-other does not contain the remote service "SimpleService". This other agent may be created in several physical computers.

Making the "server agent"

```
user@server> tar zxvf JRDL-1.0.tar.gz
user@server> cd jrdl_runtime
user@server> make server
user@server> cd ../jrdl_runtime-server
user@server> make run
```

These steps create a source directory jrdl_runtime and build directory jrdl_runtime-server. Jrdl_runtime-server contains the remote service "SimpleService". This server agent may also be created in several physical computers.

Making the "client agent"

To see the complete interaction of the runtime daemon and the client program, two terminals are needed. One is for running the runtime daemon, and the other one is for invoking the program.

In terminal for **runtime daemon**, execute these steps:

```
user@client> tar zxvf JRDL-1.0.tar.gz
user@client> cd jrdl_runtime
user@client> make run
```

In terminal for **client program**, execute these steps:

```
user@client> cd jrdl_app/SimpleClient
user@client> javac client.java
user@client> java client
```

These steps build the source code in directory jrdl_runtime. Only one physical computer is needed to run the client.

APPENDIX B – REPORT ON CHORD AND FREEPASTRY SCRIBE

When we develop the query router for JRDL, we explored two other systems before we decide to use Associative Broadcast. These systems are Chord and FreePastry.

Chord [20]

Chord is a Distributed Hash Table (DHT) system designed to connect many different nodes into a peer-to-peer ring-like network. Chord's routing mechanism utilizes finger index table, which enables efficient routing based on node id. Chord was implemented in C++.

During the development of JRDL, The Chord developers did not have an official release. Instead, we have to download the source code directly from their CVS repository. Chord source code was having an intensive development at that time which resulting an unstable code.

We managed to get Chord working in our test bed and successfully ran the simulation given in its installation website. However it was not usable. There was no documentation that explains about writing an application that uses Chord functionalities. It also requires a global file system named Self-Certifying File System[21], which was only able to access read-only files. Before we decided to implement JRDL in Java, we found that Chord was theoretically suitable to be the

JRDL's query router. However, because of these practical findings, we finally concluded otherwise.

FreePastry Scribe [22, 23]

FreePastry[22] is an implementation of Pastry [22] Distributed Hash Table. Scribe[23] is a scalable group communication level that builds on top of Pastry. Scribe enables a Pastry node to send a notification to the entire network, which resembles multicast protocol in application level.

FreePastry is implemented in Java. As of today, its latest official release is version 1.2, which runs under Java JDK 1.4. The source code also includes some simple applications which may be used as examples to build a custom one. This system seems to be a potential system for JRDL's query router.