Providing Easier Access to Remote Objects in Client-Server Systems

Jonathan Aldrich, James Dooley, Scott Mandelsohn, and Adam Rifkin
California Institute of Technology
{jonal, jdooley, scott, adam}@cs.caltech.edu

Abstract

The Java Environment for Distributed Invocation (JEDI) is efficient, dynamic, and easier to use than alternative communication systems for distributed Java objects. Existing state-of-the-art mechanisms for remote method calls on Java objects, such as RMI, require users to perform a complicated series of steps. Furthermore, the compiled static interfaces these systems use limit their functionality. This paper presents the design and implementation of JEDI's simpler approach utilizing dynamic proxies. We discuss a means of integrating JEDI with a publicly available CORBA ORB, followed by the tests used to ensure the robustness of the JEDI system. Comparing this system's performance with that of other communication facilities such as UDP, TCP, and RMI demonstrates the efficiency of JEDI. A calendar-scheduling application illustrates the flexibility and usability tradeoffs of employing JEDI in distributed client-server applications.

1. Introduction

Java programs can use the Internet for distributed computations in many different ways [8]. One such technique involves message passing [9] between objects on different machines, as exemplified by Caltech's Infospheres [4], IBM's Aglets [14], and the iBus multicast system [16]. Another technique involves accessing remote objects through a request broker active on a remote machine using CORBA [17] or DCOM [6]. Some systems communicate with remote objects through a gateway to a Web server using HTTP and CGI [1]. Method calls on remote objects may be made using Open Network Computing Remote Procedure Calls [29] or Java's Remote Method Invocation [12]. With each of these techniques, the programmer must deal with creating extra interfaces (often in a different language) and must do other setup work to handle low-level communication details.

This paper explores remote method calling facilities that automatically handle some of the more cumbersome communication and synchronization responsibilities [21]. Many existing systems, including Java's RMI, require a programmer to run interface code through a preprocessor to create stub and skeleton objects. We have developed an alternative system for remote method calling, offering the programmer complete control over communication while simplifying the model of distributed computing. Since our system uses Java's serialization capabilities, a programmer can automatically send any object to a remote machine.

First, we will discuss and evaluate several existing systems, including RPC, RMI, CORBA and IIOP, DCOM, and Infospheres. These systems motivate the design and implementation of JEDI, a system that allows dynamic method invocation (the ability to call any method of a remote object at run-time without relying on statically compiled interfaces) and requires fewer development steps than other existing systems. Then we describe how a developer would use JEDI. Next, we discuss several experiments in client-server computing to determine the flexibility, scaleability, and ease-of-use of the JEDI system. These experiments include studying the integration of JEDI with CORBA, testing the performance and reliability of JEDI, and using JEDI to develop an application from scratch. We conclude with a short summary of our JEDI findings.

2. Existing Systems

Until recently, Java lacked a native client-server method invocation paradigm [24], but several supplementary systems are available to provide this functionality.

In this section, we explore and compare some of the current communication mechanisms in client-server systems: RPC, RMI, CORBA and IIOP, DCOM, and the Infospheres Infrastructure.

2.1. ONC Remote Procedure Calls

For years, programmers have used ONC's Remote Procedure Call (RPC) system [29] to automate some of the communication tasks between client programs and their servers. Although RPC was one of the first systems to simplify the development of distributed applications over the use of plain socket connections, RPC does not handle remote procedure calls automatically. The programmer must first design the interfaces (step 1 in figure 1) on both the client side and server side so they will connect properly when the distributed system is invoked.

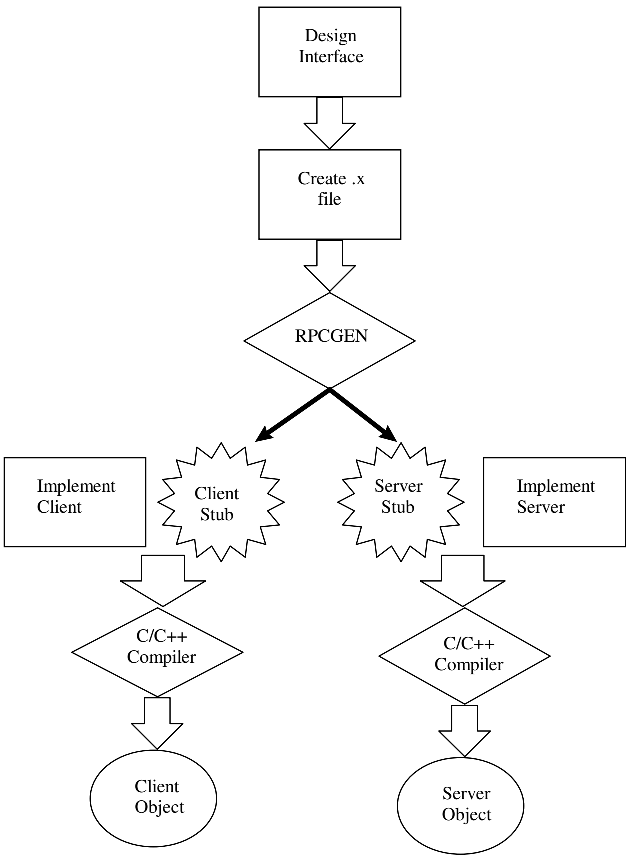

Figure 1. Although the Remote Procedure Call system somewhat simplified the job of a client-server system developer, it still requires running separate interface files through a preprocessor and filling in the resulting stub files.

Next, the user creates a `.x` file (step 2) that specifies the designed interface. This file is then run through `rpcgen` (step 3) to construct client and server stubs. The programmer then fills in the client and server stubs produced by `rpcgen` with code that implements the desired behaviors (step 4). This code is then compiled for execution (step 5). Some consider this system to be more straightforward than standalone sockets or message passing, because the client can invoke operations on the server by using what looks like a local procedure call. Making network communications into procedure calls meshes well with the top-down design techniques of procedural programming. However, the RPC system requires that the programmer tediously set up all the remote methods in the interface description file, run the preprocessor, and implement the resulting stub files.

2.2. Java's Remote Method Invocation

The Remote Method Invocation (RMI) system furnished by Java 1.1 allows RPC-like access to remote objects [12]. RMI includes support for data serialization, remote class loading, and socket manipulation.

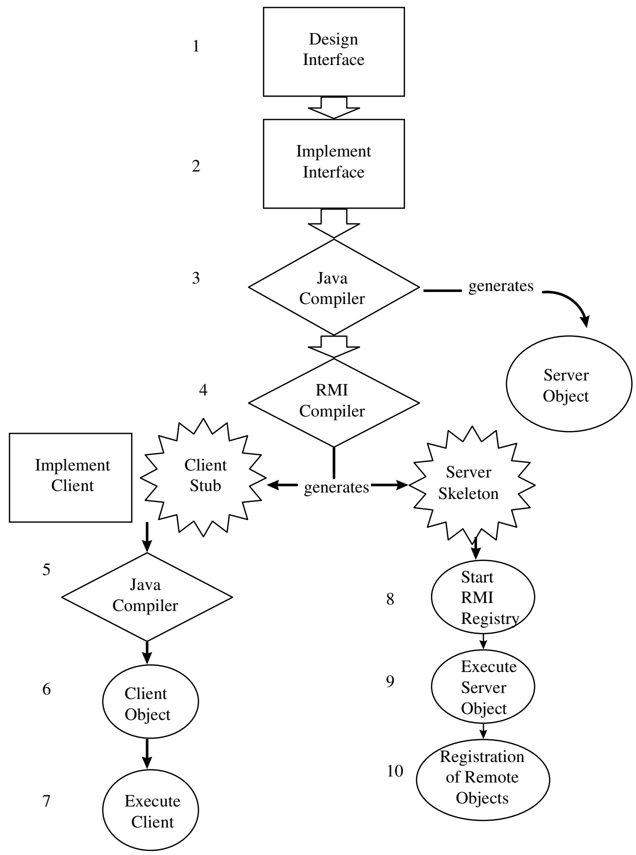

Figure 2. RMI improves the RPC-like access for remote Java objects, but adds several extra tasks for the developer.

To use RMI, an application developer creates a Java interface for the object to be accessed remotely (step 1 of figure 2), and then writes a server class to implement this interface (step 2). The Java compiler compiles the server code (step 3), and the RMI preprocessor `rmic` uses the resulting class to create the server skeleton and client-side stub (step 4). The client may be written and compiled any time after the server interface is written (steps 5 and 6). Before the client can access a remote object on a server (step 7), the RMI registry must be run (step 8), the server object must be created in a Java VM (step 9), and the server must register itself in the RMI registry (step 10).

We believe that the extra preprocessing steps introduce unnecessary complexity. They add several additional object files to the compilation process, and restrict the methods that can be run on remote objects to those that are described in a static interface. If a server object is updated to include new functionality and a new interface, the client will be unable to use the new interface without resorting to Java's somewhat awkward reflection capabilities.

RMI is difficult for developers to use, because it does not allow client programs to use new functionality in server programs without first recompiling (and redistributing) the client programs or using Java's reflection to get around this static limitation of RMI. Another cumbersome aspect of RMI is its requirement that programmers must use the rmic preprocessor to generate code for the server skeletons and client stubs. RMI can use only a few wire protocols (currently TCP/IP and HTTP), but some applications would benefit from the use of custom transport protocols available through a generic message infrastructure. The ACE framework [25][10] and the iBus project [16] both provide a layered component-based Java communication system that allows plug-in custom transport protocols to provide different quality of service facilities to applications.

ObjectSpace Voyager [18] provides remote method invocation facilities much like RMI's, but makes the development process much simpler and provides additional features. Developers run an existing class through the Voyager preprocessor to create a stub class with all the methods the original class had. This saves them the work of writing an remote interface file and changing their code to implement the interface. Although Voyager allows dynamic method calls, it requires developers to specify methods with the unintelligible method signature syntax used by the Java virtual machine.

2.3. CORBA

The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) allows the development of distributed applications with component objects [30]. CORBA's language-independence allows objects written in different languages to communicate with one another. All object interaction is routed through intermediary Object Request Brokers (ORBs) which communicate through the industry-standard IIOP protocol (see figure 3). CORBA uses client and server stubs created from an interface definition written in the ISO Interface Definition Language (IDL).

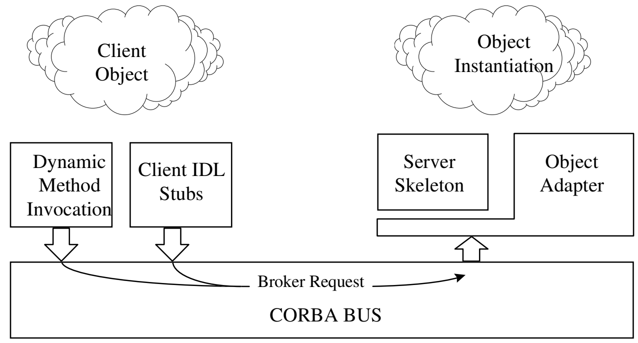

Figure 3. Through the CORBA bus, client objects send requests for method invocations to the remote ORB, which routes the request through the object adapter and server skeleton to the server object.

To create a Java-based client-server application in CORBA, the programmer first writes a IDL file defining the signatures of the methods that need to be called remotely. The IDL file is then run through a Java preprocessor which creates an interface for the server object and a stub class that will forward method calls to that server object. Then the programmer writes the client program and an object that implements the server's interface. The client and server can then be compiled and run.

CORBA has support for clients to discover and use interfaces dynamically through its Dynamic Invocation Interface (DII) [17]. When using DII, the client creates its method calls at run-time, rather than calling methods in the stub. A CORBA specification for the Dynamic Skeleton Interface (DSI), allows server objects to update their interfaces at run-time [17]. Any method invoked through the DSI is passed through a single upcall method (written by the programmer) that is responsible for checking the method name and forwarding it to the correct implementation method.

We looked at two particular ORB implementations. Xerox PARC's Inter-Language Unification (ILU) system [11] interoperates with other CORBA ORBs using the Internet Inter-ORB Protocol (IIOP). Although ILU does not implement many features of the commercial ORBs [7], it provides DII and is freely available. As described in our section "Experiments in Client-Server Computing", we have worked on an ILU interface that allows CORBA objects to call methods on JEDI objects.

VisiBroker, Visigenic's implementation of CORBA, has similar features to ILU but includes more complete functionality. Caffeine, a part of VisiBroker, includes a compiler that generates IDL code from a Java interface, making the CORBA development process in Java much like that of RMI.

2.4. DCOM

Microsoft has recently developed a Java interface to their Distributed Component Object Model (DCOM) [6]. DCOM is another system that allows RPC-like calls on remote objects; it uses a DCE-like IDL language to define interfaces. Note that Microsoft's IDL (MIDL) is compliant with neither CORBA IDL nor DCE IDL.

Like CORBA, RMI, and RPC, DCOM requires compiling interfaces written in its IDL into stub objects. However, DCOM has the added complexity of requiring that a type library for the object be created as well. The server object must then be written to implement the defined interface. The client code is fairly straightforward, but DCOM objects can only be accessed through an interface, not directly. Also, both the client and the server must register the DCOM object with the operating system before the client may access it. To use a reference to a local DCOM object in Java, a program must first cast the object to its corresponding interface before using the class. Although several ports are planned for the future, DCOM is presently available only on Windows 95 or NT systems.

DCOM Automation allows clients to make dynamic method calls. By exposing one or more *dispinterfaces* (a set of methods that can be called dynamically), an object can make methods available to clients that were not compiled with DCOM interface stubs. The client packages up the arguments to the call in a *variant* data type and combines this with an integer ID denoting which method to call. These parameters are passed to the `Invoke` method of the DCOM interface `IDispatch`. When the `Invoke` call reaches the server, the destination object must check the ID in order to discover which method is being called before unpacking the variant parameters and implementing the call.

2.5. Comparing RMI, CORBA, and DCOM

Comparing each of these distributed object communication mechanisms [19], we note that RMI, CORBA, and DCOM all offer somewhat seamless Java integration, typed parameter support, and reasonable performance. However, all three approaches suffer from high setup costs due to programming complexity, lack of configuration ease, evolving wire-level security, and limited dynamic discovery and dynamic dispatch when compared with systems such as NeXT's Portable Distributed Objects [22]. Furthermore, although CORBA was designed to scale to accommodate communication among many objects, neither DCOM nor RMI presently seems suitable for communication among more than a handful of objects [19].

The Infospheres Infrastructure [4] offers a solution to the scaling problem by providing mailboxes that can send and receive typed messages. With these mechanisms, developers can set up sessions of persistent communicating objects [3]. JEDI was originally constructed as an invocation layer built on top of the Infospheres message-passing communication layer. As the package evolved, the Infospheres plumbing was replaced by a more performant communication layer using UDP.

Work is proceeding on the design and implementation of the second generation of the Infospheres Infrastructure, which integrates JEDI's invocation facilities. With this design, we hope to make JEDI even easier to use and provide features like security, authentication, and an even more flexible and performant communication layer.

2.6. The Evolution of JEDI

Originally, JEDI was designed to make access to remote objects completely transparent to the programmer, handling all of the networking and synchronization details. As the package developed, we realized that complete transparency is not always desirable in distributed systems. Several unique characteristics of distributed systems, including uncertain delays and distributed failure, must be considered when designing such a system [31]. Unfortunately, when the distributed nature of a system is not hidden, programmers must often deal with low-level coding issues and with complicated development tools such as RMI. JEDI's focus therefore shifted to making robust distributed systems easier to implement.

2.7. A Simple, Dynamic, and Global Vision for Distributed Computing

We envision a distributed computing model with billions of objects scattered over the globe, interacting with each other via the Internet [4]. Because objects in one Java VM usually communicate with method calls, we believe that a communications system based on remote method calls is conceptually more simple than a message-passing system. Although message-passing is a more general communication framework, developers are more comfortable reasoning about and using method calls to communicate between objects. Unfortunately, many existing remote method call systems are quite complex and have many steps to learn and repeat. JEDI's primary goal is to simplify or eliminate as many of these steps as possible.

JEDI provides a simple and flexible dynamic invocation service. Conventional RPCs, for the most part, are based on static interface definition files. RPC requires a static, compile-time interface, and standard CORBA requires the same. Through the Dynamic Invocation Interface facilities, CORBA clients can discover resources dynamically. However, DII is somewhat difficult to use—many steps are required to construct a dynamic request object [19]. If the client must query the server for the interface of the method it wants to call, performance may decrease dramatically. DCOM Automation is likewise complicated and difficult to use. With JEDI, calling a method dynamically is a simple process: a client must bind to the object it is calling, and then it can invoke any method by name. Querying the interface of a remote object is as simple as calling `getClass()` on the object (using JEDI) and then finding its public interface using Java's reflection capabilities.

As the number of objects running on the Internet increases, truly dynamic RPC interfaces will be a necessity, because it will become impossible to take tightly-integrated object systems off-line so that they can be recompiled to produce new static interfaces. With massive distribution, knowledge of precompiled stubs of every object in the network universe just is not practical, because massive distribution requires dynamic typing and construction of messages at run-time. Just as latency and partial failure are inherent aspects of distributed computing, coping with dynamically changing and unknown interfaces is an inherent aspect of massively distributed computing. Therefore, JEDI was developed to allow the programmer to dynamically call any method on any Java object at run-time.

3. An Overview of JEDI

We now describe the design of JEDI, and its use in distributed Java systems.

3.1. The Design of JEDI

The simplicity of using the JEDI system is illustrated in figure 4. Instead of involving a stub compiler, JEDI provides a software library for making remote method calls; as a result, any method can be called remotely at run-time. The development process is identical to writing any non-distributed Java program: classes are written, compiled and run. Any object can be called from another machine simply by giving it a name. A client can bind to a remote object by creating a proxy with the remote machine, port, and object name. The client can then call any public method of the object with one simple line of source code.

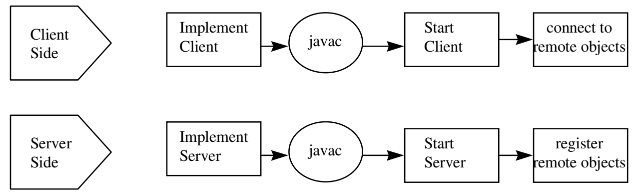

Figure 4. The JEDI package simplifies a programmer's task considerably. The user needs only to implement, compile, and start up the client and server programs. Servers name their objects in a registry service so clients can call methods on them.

Unlike most other RPC schemes, no source code changes must be made to an object before its methods can be called globally. This means that methods of any object, including the core library objects like String, can be called remotely—even if the source code for the object's class is not available. Thus, "legacy" Java objects not designed for distribution can nevertheless be integrated into distributed systems without writing the wrapper classes necessary with other RPC systems.

Since JEDI does not require a precompiler, the program does not need to know the signature of any method it will call until the call is actually made. An interesting consequence of this is that JEDI does not require the shutting down of a distributed object to update its interface for new remote calls. Instead, that object can be replaced on-the-fly by an updated version providing expanded features and clients will be able to access the additional functionality immediately. By using a dynamic system like JEDI for remote method calls, distributed systems can scale up to more objects.

Because JEDI is a library-based system, it fits more naturally into the usual program-development cycle than precompiler-based RPC systems. In general it is easier to learn to use a software library than to use a new command-line tool. Also, it is easier to understand what is happening inside a software library than to understand the black-box code generated by a precompiler—especially if the library's source is available.

A potential disadvantage of a dynamic scheme like JEDI is that there is no static type-checking. A method call can fail at run-time if the programmer makes a mistake and misspells the method or passes the wrong parameters. However, in a distributed computing system the programmer must be aware that any method may fail because of a failure in the network or in the remote machine. No distributed system can mask the failure of an arbitrary method call. Furthermore, a system like RMI that provides interfaces to remote objects still cannot ensure that the programmer does not try to make an illegal method call—it just reports the error as an illegal cast rather than a nonexistent method. Thus, the dynamic typing system of JEDI is not a significant potential point of failure for the application programmer.

Java's serialization mechanisms simplify programming for the JEDI system. Any object that implements `java.io.Serializable` may be passed to a method of a remote object:

```
public class MyClass
      implements java.io.Serializable {
      // class definition here
}
```

The JEDI system permits the creation of distributed systems using the intuitive remote method call paradigm without the complexity of many similar schemes. JEDI provides simple but dynamic remote method calls, giving programmers the ability to make run-time modifications.

3.2. Using JEDI

Consider a simple remote method call using JEDI, as illustrated in figure 5. Before a method call can be made, the object on which it acts must be registered under a name on the server. This is accomplished by getting the local repository (use `Repository.local()`) and calling the bind method with the new name and the object to be registered. Then any client can call a method on that object using JEDI.

The client begins this process by creating a Proxy object with the network address (such as "`www.caltech.edu`") and the name of the object. Once a proxy has been created, any method can be called on a remote object by calling `function()` on the proxy. There are several versions of `function()`; the most general accepts a method name and a vector of arguments to that method and returns an arbitrary object. Underneath it all, the JEDI system will send the method call information to the remote machine, which will find the object associated with the proxy. It will then look up the method with the correct name, invoke the method, and pass the return value back to the client. If any exception is thrown in the method, or the method or object cannot be found, or there is a communication error, an exception will be thrown from `function()`. An example demonstrates these concepts.

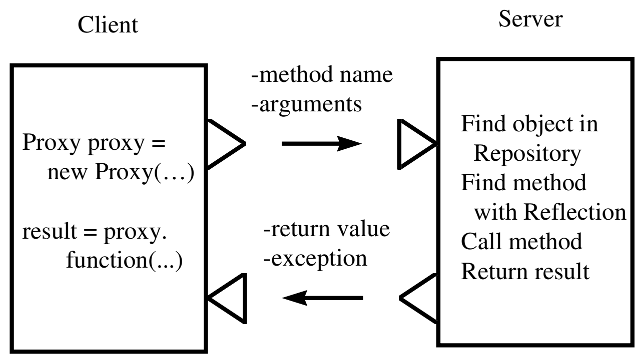

Figure 5. JEDI allows a client to set up a `Proxy` object through which it makes its remote method calls. When calling a method through the proxy, the method name and arguments are sent through JEDI's communication layer to the server, which finds the object in its object repository and finds the proper method using Java reflection. The method is then called on the object on the server machine, with the return result shipped over the JEDI wire back to the client program. New objects on the server can be plugged in on-the-fly, so dynamic methods can be invoked at run-time.

3.3. A Simple Example of Using JEDI

This simple JEDI server allows remote clients to call any method on a String object.

```
import info.jedi.*;

public class ServerTest
{
   public static void
      main(String args[])
   {
     String string = "Hello";
     Repository.local().bind(string,
         "HelloString");
   }
}
```

To expose an object to remote clients, a user needs to create the object and bind it to a name in the repository. In this example, we have created the string "Hello" and bound it to the name "HelloString" in the local `Repository`. Under the hood, our call to `Repository.local()` initialized the JEDI system to listen for incoming remote method calls on the default port. Our server is now ready to accept requests from the following simple client:

```
import info.jedi.*;

public class ClientTest
```

```
{
  public static void main(
      String args[]) throws Exception
  {
     Proxy proxy = new Proxy(
       "harmonica.cs.caltech.edu",
       "HelloString");

     System.out.println(
       "The string's length is "
       + proxy.function("length"));
  }
}
```

This client creates a proxy object to represent the string on the remote machine. In our example, the server is running on `harmonica.cs.caltech.edu`, so this is the network address we pass to the proxy constructor. Since we registered the object under "HelloString," this is the object name we pass to the proxy constructor. Once the proxy has been created, we can call any function on the remote object. In this case, we call the `length()` method of the `String`. We use a convenient version of `Proxy.function()` that does not take a vector of arguments, because the `String.length()` method has no arguments. Other versions of `Proxy.function()` are provided for calling methods with different numbers of arguments. The `length()` method will return an `int`, which will be wrapped in an Integer object, passed back over the network to the client, and finally returned by `proxy.function()`. It will be converted into a string by the concatenation operator, and the output of the client will be:

```
The string's length is 5
```

3.4. A Tour of the JEDI Architecture

JEDI includes a simple but powerful general messaging infrastructure. This infrastructure is designed to be both efficient and modular, and is implemented in several layers with a protocol stack architecture similar to that of iBus [16]. One layer can use the services of another; thus the remote method call facility uses the reliability layer to make robust remote method calls over an unreliable network connections. Because the layers are loosely connected, a separate messaging service layer could make use of the reliability layer to provide robust message passing, as is done with the ACE system [25].

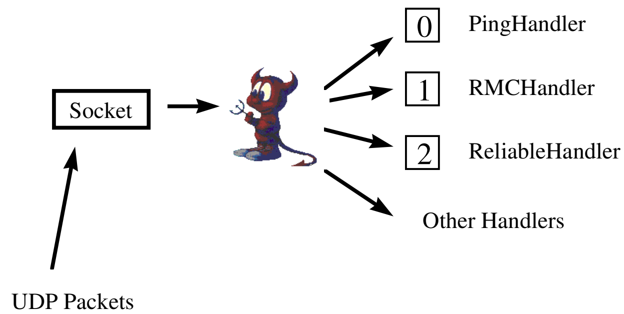

Figure 6. The JEDI `MailDaemon` sits at a socket, waiting for UDP packets. Upon receipt of a UDP packet, it routes the packet accordingly to the `PingHandler`, the `RMCHandler`, the `ReliableHandler`, or some other packet handler. The BSD Daemon is copyright 1988 by Marshall Kirk McKusick.

The center of the JEDI system is the `MailDaemon` class, illustrated in figure 6. Any program needing to communicate uses a `MailDaemon` to forward incoming UDP packets to the appropriate packet handler. When a packet is received, it is converted into an `InputPacket`, which creates a `DataInputStream` for reading the contents of the packet. Then the first byte is examined; this byte indicates which of up to 256 well-known packet handlers will process the packet. Each packet handler must conform to interface `PacketHandler`, which defines a method that takes an `InputPacket` and returns true if processing of that packet is complete. If processing is not complete when the packet handler returns, the next byte is assumed to be the next handler. This provides the layering mechanism in JEDI: a packet can first be processed by a reliability layer and then passed on to a higher-level layer that uses data from the packet for computation. The slots that are not yet used are filled with `DefaultPacketHandlers`, which simply ignore any packets they are passed.

A `ResourceHandler` is created for the `MailDaemon`, and sets up threads of type `ReceiveThread` to handle incoming packets. If there is any possibility of a thread suspending during the handling of a packet, it should call `ResourceHandler.threadBusy()` followed by `ResourceHandler.threadIdle()` when its work is done. These method ensure there are enough threads to handle incoming packets, and create another `ReceiveThread` if there are not. In this way many simultaneous packets can be serviced, up to the thread limit of the virtual machine. For efficiency, the `ResourceHandler` also keeps track of a pool of 64KB packets (which are expensive to create for each incoming packet).

OutputPacket is a convenience class for creating, filling, and sending a JEDI packet. It creates a DataOutputStream that fills up a ByteArrayOutputStream. It also provides methods for sending the packet to another host and for resending it if it gets lost on the way.

As a test for the system, the simplest handler provided is the PingHandler. This class simply sends a packet back to the original host. The PingHandler class has a main() method so a program can test the latency of its network connection through JEDI. It also provides a tool to compare the overhead of the JEDI messaging structure with that of a simple UDP ping.

The reliability layer is implemented through the ReliableHandler class. Clients can call ReliableHandler.addReliability(p) to make packet p reliable. ReliableHandler extends Thread, and the run() method will periodically resend each reliable packet until it times out or is acknowledged by the remote host. No packet ordering is necessary in a remote method call system, so all available packets will be sent immediately without waiting for acknowledgments. When a reliable packet is received, the ReliableHandler is called. It will check to make sure the packet has not been duplicated, and then send an acknowledge packet so that the sending host knows that the packet has been received. If the remote host cannot be contacted within a specified period, a TimeoutException will be thrown to the caller.

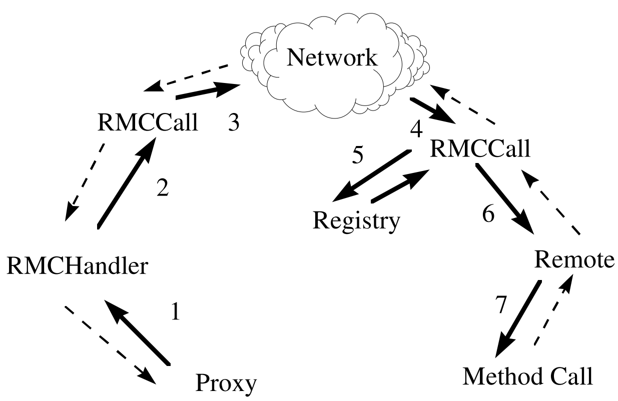

Figure 7. The JEDI package automatically handles the seven steps of a remote method call.

The JEDI remote method call facility is implemented through several classes. The Proxy object asks the RMCHandler object associated with the local MailDaemon to send the method call out over the network (step 1 in figure 7). The RMCHandler object creates an RMCCall object to represent the method call (step 2). The RMCCall object uses the reliability layer to send a reliable packet to the remote machine (step 3), which is intercepted by the RMCHandler object there (step 4). This handler will create an equivalent RMCCall object on that end of the network, look up the object's name in the local Repository object (step 5), and dispatch the method call using a method from class Remote (step 6). Remote.staticCallFunction takes an object, a method name, and a vector of arguments, and uses Java's reflection facility to find and call the appropriate method (step 7). The RMCCall object then sends the return value back to the original client, where it is decoded and passed on to the user.

4. Experiments in Client-Server Computing

To determine the flexibility, reliability, scaleability, and ease-of-use of the JEDI system, four key experiments were performed. We investigated integrating JEDI with CORBA ORBs, and designed a comprehensive test suite to demonstrate the reliability of the JEDI system. Later, we tested and compared the performance of JEDI with several other systems and compared the implementation of a simple scheduling application using JEDI, RMI, and Infospheres.

4.1. Flexibility: Interactions with ILU/CORBA

We experimented with creating a CORBA object that would allow remote invocation of JEDI objects. The testbed chosen was ILU because it is free and openly available from Xerox. To make a JEDI object accessible to CORBA ORBs using IIOP, we created the following CORBA IDL file to expose an interface to the Remote class:

```
module jedi {
  interface ILURMCCall {
    exception JediException{};

    typedef sequence < any > Arguments;

    any RemoteCallFunction (
        in string object_name,
        in string function_name,
        inout Arguments arguments)
            raises (JediException);
  };
};
```

Through this interface, CORBA objects can access JEDI objects through a method similar to the DII interface, as illustrated in figure 8. A CORBA call comes into the ILU system (step 1), where it is decoded and mapped to the ILURMCCall object (step 2+3). ILU then calls the implementation, ILURMCCallImpl (step 4), which accesses the JEDI registry to find the requested object (step 5). We then invoke a function in the JEDI `Remote` library to call the requested method (step 6). `Remote` then calls the method and passes back any result or exception information that was generated (step 7). This information is then returned to the original caller through ILU.

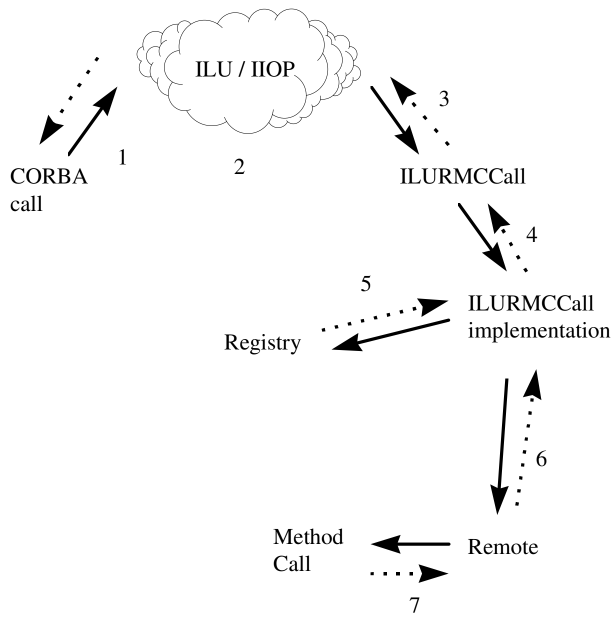

Figure 8. The JEDI system can be integrated with a CORBA-compliant object request broker such as ILU.

4.2. Reliability

To ensure a high-confidence, robust system, we devised a comprehensive test suite for JED to test all portions of the `info.jedi.net` and `info.jedi` packages. The complete source code for the test suite is available with the JEDI distribution at our web site in the info.jedi.testsuite package. In addition to testing JEDI, this code provides many examples of the different ways to use JEDI.

The simplest part of the test suite calls remote methods with different signatures and return types. We call remote methods with no parameters, and with `String`, `int`, and `boolean` argument types. A call to a method with a serializable user-defined tree-like structured data type parameter checked for the proper use of Java serialization. Finally, we call a static method and a method with one argument of each of the above types.

To ensure that performance scales up with the number of method calls, our test suite sends a user-defined number of method calls, reporting the time per message after each 10% of the messages have been sent. This part of the test suite has been tested with 1,000,000 method calls with no observed performance degradation.

To test multiple concurrent JEDI calls, we create 100 threads, each of which called a remote semaphore method. This synchronized method implements a 100-thread barrier. None of the method calls may return until all of them have entered the barrier method.

Next, to make sure that proxies can be sent to remote methods as parameters and used successfully, our test client calls a method on the server, receiving a proxy from the return value of the method. This proxy is then used to call a server method that is passed a count of 100 and another proxy to the client. The server method recursively calls the same method on the client with a count of 99 and proxy to the server. This process continues until recursive method calls had been made 100 levels deep between client and server, after which they all return.

Finally, we test the error-handling capabilities of JEDI. This includes catching exceptions thrown by remote methods and ensuring that they print out stack traces with methods from both the local and remote machines, catching "not serializable" exceptions for parameters and return values that do not implement `Serializable`, catching time-out exceptions when a remote host does not respond within a specified time period, and catching exceptions where the specified remote object or method does not exist.

The successful completion of our test suite gives us confidence that our JEDI infrastructure is reliable. Its reliability has been further demonstrated as we have begun to build the next generation of Caltech's Infospheres Infrastructure on this solid JEDI foundation.

4.3. Performance and Scaleability

We tested the JEDI package on Sun's JDK 1.1.3 virtual machine, running on Solaris 2.5.1 on two 143 MHz UltraSparc 140s with 64M of RAM and 10 Mbit ethernet connections. Although performance numbers can vary from one machine to another, we expect that the relative performance of JEDI to the other Java-based systems will remain approximately the same.

We repeated tests 5 times each and reported the best times achieved for each technology, to filter out the random effects of other users and programs on the two testing machines. As shown in figure 9, each test set up a connection from a client to the test server and sent data back and forth between the machines 100 times, and smaller numbers are better. Notice that both ping and call times are reported for JEDI; the call time includes all of the overhead incurred by passing method names and parameters, looking up the right method, and maintaining call reliability. The JEDI ping time given measures the time required to send an unreliable packet to a server and back.

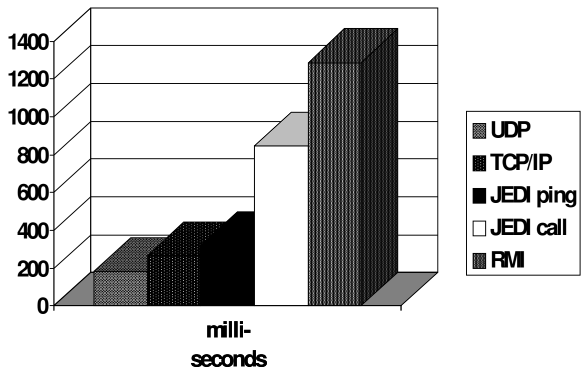

Figure 9. The performance of JEDI pings and actual JEDI remote method calls, in milliseconds, compares well with the performances of pings using UDP, RMI, and TCP/IP.

Although we considered using TCP/IP to send method call data, we were concerned that its scaleability would be restricted by the limited number of sockets the operating system is able to create. Our early (incorrect) tests also implied that the Java VM implementation of TCP/IP sockets was very slow, since the default Java TCP/IP stream does not do any data buffering. While it is still relatively expensive to create a TCP/IP stream for each method call, acceptable performance may be attainable by reusing a single stream for multiple method calls. We plan to provide the option of using this transport in future versions of JEDI.

Because of the Infospheres work being done here at Caltech, we initially used the `info.net` library of the Infospheres Infrastructure [5] as our communications protocol. However, this system provides rich features that slow its performance, such as ensured ordered messages. Performance tuning has not yet begun in earnest for the info.net library, so using it incurs a considerable performance penalty (100 back-and-forth messages took 5.0 seconds). As a result, we elected to build a more simple subsystem for JEDI communication.

This new JEDI messaging system is quite efficient. Sending 100 short UDP ping messages took 182 milliseconds on the machines we tested. Using the underlying JEDI infrastructure directly, a ping took 321 milliseconds, mainly because several convenience objects are created for each ping. On a slow, interpreted system, 1.39 ms per ping (139 ms over 100 pings) represents a fairly low overhead.

JEDI's remote method call facility also compares well with RMI in the performance domain. Testing simple functions that return a string, we found that RMI made 100 remote method calls in 1.29 seconds, while JEDI was able to accomplish the same task in only 0.85 seconds. This is remarkable, considering that RMI uses hard-coded method names and signatures that are fixed at compile time, whereas JEDI can call any method on any object at run-time using reflection. However, JEDI still lacks some functionality provided by RMI, including support for method calls involving more than 64K of data passed as parameters.

When testing JEDI and RMI without including setup time, RMI takes 3.1 milliseconds per call while JEDI takes 3.4 milliseconds per call. Since JEDI is faster when setup time is included, we conclude that connecting to a remote object is an expensive operation under RMI. Thus RMI may be better for extensive communication (more than 100 method calls) with a specific object, while JEDI may perform better when interacting with many different objects at the same time.

Research has shown the importance of measuring not only two-way ping latency in distributed object oriented systems, but also throughput and latency for sending different kinds of data structures [27]. We tested JEDI and RMI by passing a 35 Kb, richly typed, tree-like data structure in a remote method call. In this case RMI sent the structure in 0.85 seconds, while JEDI took 1.56 seconds. Since both figures are significantly longer than the time required for communication, we theorize that the delay is mostly due to Java's serialization. Tests of serialization confirm that just serializing the data structure can explain most of this time delay. Because the communication layer of JEDI sends all of its data in one packet, while the TCP/IP implementation in RMI sends data as it is produced, we believe that RMI is able to begin decoding the serialized data on the remote machine while it is still being encoded on the originating machine. This accounts for RMI's performance advantage in sending large, structured data. We plan to add the capability to send structured data more efficiently in a future release of JEDI.

During our performance experiments, we did not compare our JEDI mechanisms with CORBA invocations, because significant research is being conducted to make CORBA more performant and scaleable over high-speed networks [26], resulting in several techniques for optimizing IIOP performance [28]. We note that using the dynamic capabilities of CORBA comparable to those provided by JEDI can result in a performance degradation of more than an order of magnitude in some ORB implementations [20]. JEDI's niche is in low-end distributed system development as an efficient, easier-to-use alternative to RMI in Java programs. An example of this use is the calendar application we describe next.

4.4. Comparing Implementations of a Simple Distributed Application

As illustrated in figure 10, a calendar scheduling application [4] is an example of distributed resource management [23]. For comparison of the application of different techniques for distributed program development, we implemented this calendar application using Java with simple local method calls, after which we distributed the program using RMI, Infospheres, and JEDI. We used a responsibility-driven design [15] to coordinate the scheduling activities of multiple distributed calendar programs for each port of the system.

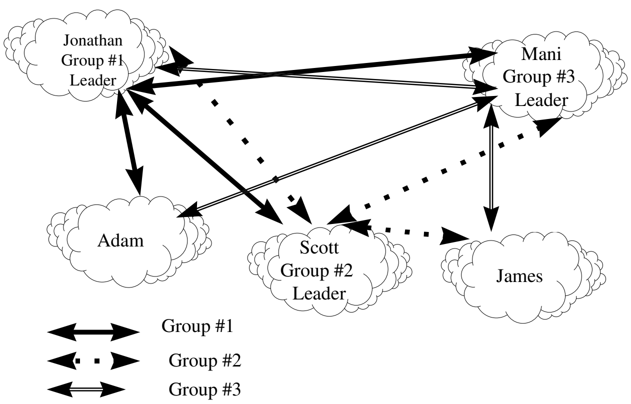

Figure 10. Jonathan, Scott, Mani, and Adam are in group 1; Scott, James, and Mani are in group 2; and Mani, Jonathan, Adam, and James are in group 3. Each person has a single calendar application that handles the scheduling of the social calendar for that person; for example, Mani's calendar application handles the scheduling of his meetings with groups 1, 2, and 3. When a group leader decides to hold a meeting, he queries the other group members in a peer-to-peer session [3] to determine an appropriate meeting time. The calendar application then locks in the appropriate slots for the group meeting in each of the respective group members' schedules.

Because of the request-response nature of the scheduling algorithm we used, this application maps naturally to remote method call semantics. As a result, using the Infospheres Infrastructure's message-passing system required more work than the remote method call systems. For example, locking the calendar objects for each member of a group requires the following code:

```
for (i=0; i < numMembers; i++) {
  sendBox.bind (
    new Place(memberAddresses[i]));
  sendBox.send (
    new CalendarMessage(REQUEST_LOCK));
  CalendarMessage response =
    (CalendarMessage)
      receiveBox.receive();
}
```

On the receiving end, a thread must be specifically set up to wait for incoming messages at a mailbox, attempt to lock the calendar, and send a response back to the process that requested the lock. If the locking operation could block, the user must create another thread to handle other incoming messages while the thread is blocking.

The advantage of a RPC-based system like JEDI or RMI for this application is that many aspects are handled manually. For example, the user does not need to write the `CalendarMessage` class (although in the case of RMI, the user will have to write a new remote interface definition file instead!). Also, the run-time library handles creating enough threads to service incoming requests (in case any method calls block). In addition, the RPC-like syntax is more convenient for the programmer because it matches the method call paradigm common to object-oriented programming systems. Other projects, particularly ones that do not require a synchronous response to every network message, are better suited to the message-passing scheme such as the Infospheres `info.net` package.

Using RMI for communication made the code highly readable (since remote method calls look just like local ones). However, RMI's multiple implementation steps were time-consuming, because an extra interface had to be developed for every object that was accessed remotely. For example, we developed a Semaphore object to guard access to each user's calendar. To allow remote users to lock and unlock the semaphore, we needed to create the following interface:

```
package jedi.calendar.remote;

import java.rmi.*;

public interface SemaphoreInterface
         extends Remote
{
  public void lock()
    throws InterruptedException,
      RemoteException;
  public void unlock()
    throws RemoteException;
}
```

This additional interface was not a useful part of our overall design, as the Semaphore class definition includes a complete specification of the interface to our Semaphore object. Although coding such a simple interface is not difficult, it would be time consuming and error prone to develop an interface for every class in a large collection of distributed objects. In a world with billions of different interacting objects, maintaining a separate remote interface for each one is not a scaleable solution.

Using JEDI did not result in code as pretty as RMI, because remote method calls are made through the generic library facility, rather than to a remote interface with a stub hidden behind it. However, developing the JEDI version was much quicker, simpler, and easier than the RMI version, because no separate interface files had to be layered on top of the existing objects, no preprocessor had to be run, no separate registry program had to started, and we did not need to keep track of stub and skeleton class files.

Locking the semaphore for each group member's calendar provides an example illustrating how RMI calls and JEDI calls are made. In RMI, this process looks like a simple procedure call due to the stub/skeleton system and the remote interface:

```
for (i = 0; i < numMembers; i++) {
  MemberInterface member =
    (MemberInterface)
      members.elementAt(i);
  SemaphoreInterface semaphore =
    member.semaphore()
  semaphore.lock();
}
```

With JEDI, the call is conceptually similar, but syntactically more complex because there is no magic preprocessor to create a Java object with the correct interface:

```
for(i = 0; i < numMembers; i++) {
  Proxy member =
    (Proxy) members.elementAt(i);
  Proxy semaphore = (Proxy)
    member.function("semaphore");
  semaphore.function("lock");
}
```

At the same time, the `Semaphore` class being accessed through JEDI was not modified in any way from a local `Semaphore` class. This demonstrates that JEDI can call methods on objects even when the source code cannot be changed. A protocol such as RMI that depends on changing the source code to implement a remote interface can never be used with libraries that are not designed with distributed computing in mind. In contrast, JEDI allows objects of any class to be fully network-capable.

One weakness in our design became apparent during this implementation: two Java Virtual Machines (VMs) cannot share a JEDI port. In our testing if we wanted two calendars to reside on one machine, we had to set up the calendar application using a different port for each member, rather than looking up the member's name in some sort of machine-global index. This demonstrates that if two JEDI objects are on different Java VMs in one machine, any process that needs to connect to them must keep track of their respective ports. Since RMI depends on a separate registry process running on each machine, we were able to look up calendars in the RMI system by name, rather than by port. In the future, we may add a machine-global directory service so that more than one JEDI VM can be run on one machine without forcing developers to deal with port numbers.

4.5. Comparing JEDI with RMI and CORBA

Table 1

Feature	JEDI	RMI	CORBA
Ease of use	easy	difficult	difficult
Dynamic invocation	yes	no	yes
Pass object by value	yes	yes	no (proposed for future)
Pass object by reference	yes	yes	yes
Steps involved	few	many	many
Inter-language	through CORBA interface	no	yes
Dynamic discovery	yes	limited to Remote interfaces	yes
Forces interface creation	no	yes	yes
Java integration	yes	requires `rmic` stub compiler	requires stub compiler
Security	only native Java mechanisms	special RMI security manager	CORBA services
Transaction capabilities	no	no	CORBA services

This table shows that JEDI is most suitable for projects requiring a Java-based RPC system with conceptual simplicity, ease of use, acceptable performance, and dynamic invocation capabilities. Although JEDI is presently less suitable for applications that require inter-language communication or advanced security and transaction capabilities, these features may be added to the JEDI system in the future.

The JEDI package can be extended in many compelling ways; some of the planned future extensions include:

1. Adding the ability to make method calls with large (more than 64K of data) arguments.
2. Enabling developers to make remote method calls without waiting for a return value (or retrieving the return value later.)
3. Adding a machine-global directory service so multiple JEDI virtual machine servers can exist on one machine without having to remember particular ports.
4. Further integrating JEDI with CORBA.
5. Allowing secure transactions on JEDI objects, including rollback and two-phase commit capabilities.
6. Allowing persistent objects that are woken up when a remote method call is made on them, as is permitted by Infospheres Djinns [5].
7. Providing a security filter mechanism for incoming JEDI method calls, perhaps allowing for trust-signed method invocation chains [13].

As a mechanism for dynamic method invocations, JEDI has become the communication substrate used with Caltech's current work on Infospheres 2.0, allowing the development of location-independent mobile objects with RPCs [2]. In the future, this system will be integrated with both events and the Infospheres mailbox and message packages, creating a JavaBeans-based infrastructure that supports RPCs and messages. New integrated system features will include asynchronous method calls (with the option of receiving a return value later), a general composition framework, fault-tolerant mobile objects, and a server-side thread control library that enables objects to determine when to process incoming method calls.

5. Summary

The JEDI system gives a developer flexibility with its dynamic dispatch of remote method calls and the potential for dynamic discovery of remote object methods through reflection. The JEDI approach is scaleable, in that its communication layer provides efficient communication among many Java objects over the Internet. The ease of using the JEDI package was demonstrated with the rapid conversion of a calendar scheduling application from a single machine application to a robust client-server system. Many possibilities exist for extending the JEDI package to provide a rich but simple and dynamic RPC-like mechanism for Java programmers.

Appendix: JEDI method APIs

Repository methods

- `static LocalRepository local()`
- `Object lookup(String name)`

LocalRepository methods

- `void bind(String newName, Object object)`
- `void unbind(String newName)`
- `MailDaemon mailDaemon()`

Proxy methods

- `Proxy(String machineName, int port, String objectName) throws UnknownHostException`
- `Proxy(String machineName, String objectName) throws UnknownHostException •Proxy(String objectName) throws UnknownHostException`
- `Object function(String methodName, Vector args) throws Exception`
- `Object function(String methodName) throws Exception`
- `Object function(String methodName, Object firstArg) throws Exception •Object function(String methodName, Object firstArg, Object secondArg) throws Exception`

MailDaemon methods

- `MailDaemon() throws SocketException`
- `MailDaemon(int port) throws SocketException`
- `InputPacket receivePacket(DatagramSocket socket, int TYPE_TO_CATCH) throws IOException`
- `void handle(InputPacket packet) throws IOException`
- `void send(DatagramPacket packet) throws IOException`
- `void addHandler(PacketHandler handler, int index)`
- `PacketHandler handlers(int index)`
- `ResourceHandler resourceHandler()`
- `DatagramSocket socket()`
- `PingHandler getPingHandler()`

- `ReliableHandler getReliableHandler()`
- `RMCHandler getRMCHandler()`

InputPacket fields

- `InputPacket(DatagramPacket packet)`
- `DataInputStream stream`
- `DatagramPacket packet`

ResourceHandler methods

- `DatagramPacket getPacket()`
- `void returnPacket(DatagramPacket packet)`
- `void threadBusy()`
- `void threadIdle()`
- `DatagramSocket getSocket() throws SocketException`
- `void returnSocket(DatagramSocket socket)`

ReceiveThread methods

- `ReceiveThread(MailDaemon md)`
- `void run()`

OutputPacket fields

- `OutputPacket(MailDaemon md)`
- `void send(InetAddress address, int port) throws IOException`
- `void resend() throws IOException`
- `DatagramSocket socket`
- `DataOutputStream stream`

PingHandler methods

- `boolean handle(InputPacket packet) throws IOException`
- `void ping(InetAddress address, int port) throws IOException`
- `static void main(String args[]) throws IOException`

ReliableHandler methods

- `void addReliability(OutputPacket packet) throws IOException`
- `boolean handle(InputPacket packet) throws IOException`
- `void run()`

RMCHandler methods

- `boolean handle(InputPacket packet) throws IOException`
- `Object call(InetAddress address, int port, String objectName, String methodName, Vector args) throws Exception`
- `static RMCHandler getHandler()`

RMCCall methods

- `RMCCall(MailDaemon md, InetAddress address, int port, String objectName, String methodName, Vector args)`
- `RMCCall(MailDaemon md, InputPacket packet) throws IOException`
- `void send() throws IOException`
- `Object getResponse() throws Exception`
- `void execute()`
- `void respond() throws IOException`

Remote methods

- `static Object staticCallFunction(Object callee, String methodName, Vector arguments) throws NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException, NullPointerException, IllegalAccessException`

Acknowledgments

This work was supported under the Caltech Infospheres Project. The Infospheres Project is sponsored by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244, by the CISE directorate of the National Science Foundation under Problem Solving Environments grant CCR-9527130, by the NSF Center for Research on Parallel Computation under Cooperative Agreement Number CCR-9120008, and by the Parasoft and Novell Corporations. The BSD Daemon is copyright 1988 by Marshall Kirk McKusick. He has given the daemon a temporary leave from his BSD duties to study this paper on networking in Java. We would also like to thank K. Mani Chandy, Mark Baker, Ron Resnick, Joseph Kiniry, Dan Zimmerman, Rohit Khare, Eve Schooler, Paul Sivilotti, and the anonymous referees for their helpful suggestions to improve this paper. The JEDI packages are available for download from the JEDI home page, at `http://www.ugcs.caltech.edu/~jedi`. A more complete version of this paper is available as California Institute of Technology Computer Science Technical Report 97-20.

References

[1] T. Berners-Lee, R. Cailliau, J. Groff and B. Pollermann, "World Wide Web: The Information Universe", *Electronic Networking: Research, Applications, and Policy*, Volume 1, Number 2, 1992.

[2] K. Mani Chandy, Jonathan Aldrich, and Dan Zimmerman, "The Infospheres Infrastructure 2.0 Specification," California Institute of Technology Computer Science Technical Report, 1997. To appear.

[3] K. Mani Chandy and Adam Rifkin, "Systematic Composition of Objects in Distributed Internet Applications: Processes And Sessions", *Conference Proceedings of the Thirtieth Hawaii International Conference on System Sciences (HICSS)*, Maui, Volume 1, January 1997, pp. 395-404.

[4] K. Mani Chandy, Adam Rifkin, Paolo A.G. Sivilotti, Jacob Mandelson, Matthew Richardson, Wesley Tanaka, and Luke Weisman, "A World-Wide Distributed System Using Java and the Internet", *IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, Syracuse, New York, August 1996.

[5] K. Mani Chandy, Joe Kiniry, Adam Rifkin, and Dan Zimmerman, "A Framework for Structured Distributed Object Computing", *Parallel Computing*, 1998. To appear.

[6] David Chappell, *Understanding ActiveX and OLE*, Microsoft Press, 1996.

[7] Ben Eng, "ORB Core Feature Matrix", `http://www.vex.net/~ben/orbmatrix.html`, 1997.

[8] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*, Addison-Wesley Developers Press, Sunsoft Java Series, 1996.

[9] C.A.R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, Volume 21, Number 8, Pages 666-677, 1978.

[10] Prashant Jain and Douglas C. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java", *The C++ Report*, January 1997.

[11] Bill Janssen and Mike Spreitzer, *ILU Reference Manual*. Xerox PARC, 1997.

[12] Javasoft Java RMI Team, *Java Remote Method Invocation Specification*, Sun Microsystems, 1997.

[13] Rohit Khare and Adam Rifkin, "Weaving a Web of Trust", *World Wide Web Journal* special issue on security, Volume 2, Number 3, pages 77-112, summer 1997.

[14] D.B. Lange and M. Oshima, *Programming Mobile Agents in Java With the Java Aglet API*, IBM Research, 1997

[15] Doug Lea. "Design for Open Systems in Java", *Proceedings of the Second International Conference on Coordination Models and Languages*, Berlin, September 1997.

[16] Silvano Maffeis, "iBus: The Java Internet Software Bus", available at `http://www.olsen.ch/export/ftp/users/maffeis/ibus/ibus_overview.ps.gz`, Olsen & Associates, Zurich, 1997.

[17] Object Management Group, *The Common Object Request Broker: Architecture and Specification (CORBA)*, revision 2.0, 1995.

[18] ObjectSpace Voyager Development Team, "ObjectSpace Voyager Core Package Technical Overview", available at `http://www.objectspace.com/voyager/VoyagerTechOviewNEWTAGLINE.pdf`, ObjectSpace, Dallas, 1997.

[19] Robert Orfali and Dan Harkey, *Client/Server Programming with Java and CORBA*, John Wiley & Sons, Inc., New York, 1997.

[20] Robert Orfali, Dan Harkey, and Jeri Edward, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., New York, 1996.

[21] L.L. Peterson and B.S. Davie, *Computer Networks: A Systems Approach*, Morgan Kaufmann, 1996.

[22] Ernest N. Prabhakar, "Implementing Distributed Objects: Doing It the Easy Way with NeXT's Portable Distributed Objects", *Dr. Dobb's Journal*, August 1995.

[23] Ravi Ramamoorthi, Adam Rifkin, Boris Dimitrov, and K. Mani Chandy, “A General Resource Reservation Framework for Scientific Computing”, *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, Marina del Rey, December 1997.

[24] Ron Resnick, “Bringing Distributed Objects to the World Wide Web”, `http://www.interlog.com/~resnick/javacorb.html`, 1996. Excerpted in *Dr. Dobbs Sourcebook special issue on distributed objects*, January 1997.

[25] Douglas C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications”, *Proceedings of the Sixth USENIX C++ Technical Conference*, Cambridge, April 1994.

[26] Douglas C. Schmidt and Andy Gokhale, “Evaluating CORBA Latency and Scaleability Over High-Speed ATM Networks”, *IEEE 17th International Conference on Distributed Systems (ICDCS 97)*, Baltimore, May 1997.

[27] Douglas C. Schmidt and Andy Gokhale, “Measuring the Performance of Communication Middleware on High-Speed Networks”, *SIGCOMM Conference*, Stanford University, August 1996.

[28] Douglas C. Schmidt and Andy Gokhale, “Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance”, *Thirty-first Hawaii International Conference on System Sciences (HICSS)*, January 1998.

[29] R. Srinivasan, *RFC 1831 - Open Network Computing RPC: Remote Procedure Call Protocol Specification*, Version 2, August 1995.

[30] Steve Vinoski, “CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments”, *IEEE Communications Magazine*, Volume 14, Number 2, February 1997.

[31] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, “A Note on Distributed Computing”, Sun Technical Report TR-94-29, November 1994.