TUFTS-CS Technical Report 2004-13

January 2005

Exploring the Potential of Out-of-Band Communication For Enterprise Java Beans

by

Joshua A. Rosen
Dept. of Computer Science
Tufts University
Medford, Massachusetts 02155

Exploring the Potential of Out-of-Band Communication For Enterprise Java Beans

Joshua A. Rosen

Department of Computer Science
Tufts University
Medford, MA 02155 USA
jrosen@cs.tufts.edu

Master's Project Paper
Advisor: Judith A. Stafford

Abstract. Operating systems today provide the ability to monitor interactions between processes and libraries, but component frameworks, such as EJB (Enterprise Java Beans) and CORBA (Common Object Request Broker Architecture), cannot monitor interactions between components. As more applications are being developed using component frameworks, there is a need to enhance the set of tools available to developers. This paper describes a communication mechanism, called an *out-of-band channel*, which allows trace messages to be sent from EJB containers to a centralized monitoring entity. This channel enables new tools to be created which can monitor the the run-time invocation relationships between EJB components and also trace the execution flow across components within an enterprise. A prototype architecture is presented along with a discussion of the unique challenges associated with monitoring components that are distributed across an enterprise.

1. Introduction

Before there were component models, component frameworks, containers, and request brokers, there were operating systems and libraries. The operating system had visibility into all of the interactions and dependencies between different processes and libraries. Multiple processes could share a single library, and libraries could use other libraries. Developers could request information about such dependencies and interactions using utilities that were usually bundled with the operating system itself.

Developers designed and wrote software for a specific platform and operating system. While this made porting of code difficult and required platform-specific knowledge, developers could rely on numerous tools and utilities to demystify the complex interactions between application building blocks.

With the advent of component technologies, such as Common Object Request Broker Architecture (CORBA) and Enterprise Java Beans (EJB), developers benefitted from a separation of concerns where they needed to know little, if anything, about the specifics of the operating system. Components could be architected according to a component model, rather than directly to an operating system. This clearly provided numerous benefits to software developers and commercial enterprises alike. For developers, component frameworks increased the potential for code to be reused as well as made portability possible to any platform and operating system that supported the framework. For commercial users, this

created a marketplace where components could be purchased and deployed to any number of different system configurations.

However, the ability for component frameworks to insulate the developer from the complexity of the operating system came at the expense of being able to use many of the rich tools that the operating system provided for developers. In many respects, component developers are now more limited than their old-school counterparts. The component frameworks, which are realized by CORBA Object Request Brokers (ORBs) and EJB containers, do not provide an analogy to these powerful operating system tools at the component interaction level. As a result, the EJB developer is required to use intuition and experimentation instead.

One of the key necessities for efficiently building software out of a set of components is being able to predictably assemble them. Trial-and-error assembly is both difficult and time consuming. It is of utmost importance for the developer to know how the blocks interact and to understand the dependencies between them.

Operating systems today provide the ability to monitor interactions between processes and libraries, but component frameworks cannot monitor interactions between components. In order to provide such a mechanism for component frameworks, it must be possible to extract information from the components themselves in real-time and to make this information accessible to a set of new utilities.

An *out-of-band channel*, which will allow the components and the new utilities to communicate, is the focus of this research. The out-of-band channel will provide the ability to capture information about a given component, such as stack trace or dependancy information, without disrupting its normal operation. The EJB component model, using the JBoss application server as an example, has been selected to demonstrate the advantages of such a mechanism, as well as some of the challenges that accompany any distributed component environment.

2. Background

The out-of-band channel, which is a communication mechanism allowing beans to send messages to a centralized monitoring entity, will allow new utilities to be created that can trace an execution flow and monitor dependencies. These features are based on functionality that exists today in many operating systems. Below is a description of these operating system utilities. In addition, this section outlines some core technologies that serve as the foundation for the out-of-band channel.

2.1 Operating System Utilities

There are several examples of Unix utilities that help the developer understand dependencies and trace interactions. These utilities help uncover “uses” relationships between processes and libraries and also run-time invocation relationships. For example, the `ldd` command has the ability to show the static dependencies that a given binary has on a set of libraries. The set of dependent libraries were determined when the binary file was compiled. This command not only lists which libraries are required for the executable to run, but it also shows where these libraries have been found in the file system. This is an invaluable tool as it

helps the user determine in advance which libraries are necessary and more importantly which are missing.

In reality, most applications do not define at compile-time each and every library which they might use. Instead, the operating system provides the ability for an application or library to dynamically load other libraries as they are needed. Once again, the operating system provides tools for learning about these dynamic dependencies. The `pmap` command in Linux, which uses the proc file system, displays the current set of libraries that a given process is using. The set of libraries that a process uses is constantly changing. A user could watch this process over a period of time and begin to understand what dependencies it has on other libraries.

An even more powerful tool that many operating systems provide is the ability to watch the execution flow of a process while it is running. Some examples of this are the `truss` command on Solaris and the `strace` command on Linux. These commands utilize instrumentation in the kernel that captures each system call and makes it accessible to these user-space commands. The `truss` command on Solaris also has the ability to trace execution into user libraries. The `truss` and `strace` commands not only provide a window into what a process is doing, but also can help debug common problems such as segmentation violations and hanging processes.

2.2 Dynamic Proxies

The EJB architecture is based around a Java technology known as dynamic proxies. Dynamic proxies are dynamically generated classes that implement a given interface. In the case of EJBs, the interface is the `EJBObject` stubs. The code, which is generated at runtime, can be downloaded to the client and contains all the logic for the client to invoke the stubs and communicate to the server-side components. Remotely, there exists an invocation handler that processes each invocation.

The invocation, which includes the method name, arguments, and some additional context information, is marshaled by the proxy in order to be sent over the wire. The EJB architecture relies on this capability to provide location transparency of components. In order to implement dynamic proxies, application servers perform introspection on the class in order to generate the dynamic code. This demonstrates the high level of visibility that EJB containers have into each method invocation.

Each application server implementation can implement dynamic proxies differently as long as they adhere to the proper interfaces. Figure 1 shows the sequence of interactions between a client, which performs a download of the dynamic proxy, and the server. In this diagram, the client uses the `InitialContext` class to perform a JNDI lookup. This will cause the server to generate a dynamic proxy which can then be downloaded by the client. This proxy implements the bean's `EJBObject` interface. The client uses this interface as if the object were local to the client. Invoking one of the methods in the stubs will in turn cause the invocation to be marshaled in an application-server-specific way. The application server will handle this invocation and execute the remote method.

Since proxies are generated and downloaded dynamically, this ensures that the client code and bean implementation do not need to change even when porting to a different application server. Although proxies can implement marshaling differently, the client is not affected since the proxy will always implement the same interface.

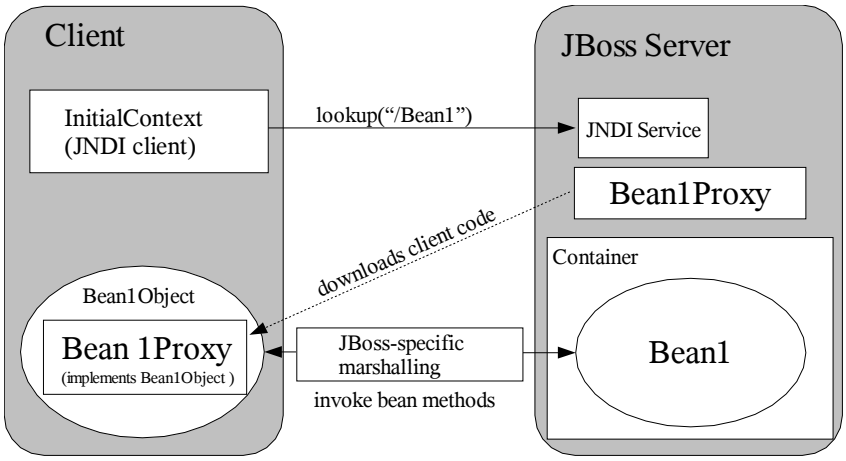

Figure 1: Dynamic Proxies

2.3 JBoss Interceptor Stack

A key part of JBoss' implementation of the EJB interfaces is something called the Interceptor Stack. This inventive solution is a mechanism for handling method invocations in a modular way. The Interceptor Stack acts as a buffer between the caller and the bean, enabling the semantics of an invocation to be verified before it is actually invoked within the bean. Handling method invocations provides additional security and robustness by allowing policies to be enforced and method arguments to be validated. Each interceptor in the stack is given full control of a method invocation in order to perform its specialized function. Each interceptor in the stack can read or manipulate the invocation one at a time. The last Interceptor in the stack is the container itself which actually invokes the method of the bean.

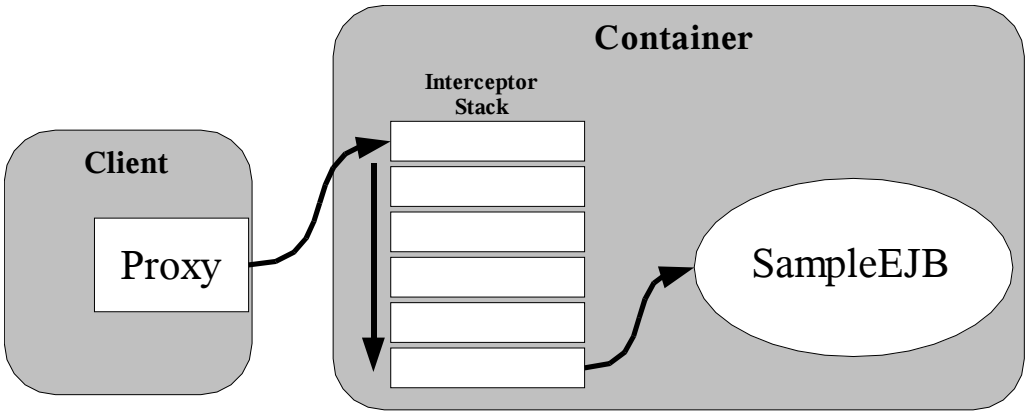

Figure 2: JBoss Interceptor Stack

The interception takes place between the proxy invocation and the bean itself. For example, there is an Interceptor that applies transactional semantics to an invocation, one that checks for the appropriate security, and another that logs the invocation.

Interceptors can be added to the Interceptor Stack by editing an XML-based configuration file. It is not necessary to recompile JBoss in order to add functionality. Figure 2 is a logical

illustration of the Interceptor Stack.

3. Requirements

This project investigates the potential of enhancing component frameworks to provide some similar tools that exist for developing and deploying operating-system-specific applications. In particular, the Enterprise Java Beans (EJB) technologies is used as a demonstration of such enhancements. There are three main requirements for this project: to be able to trace the execution flow between EJBs in an enterprise, to find out what dependencies a given EJB has on other EJBs by watching it, and to accomplish the first two requirements without requiring modification to the beans themselves.

For the first requirement, tracing the execution flow between beans in an enterprise, a command similar to `truss` should be created. However, unlike within an operating system where all libraries tend to be locally accessible, EJBs can live across many application servers and across the network. This presents a number of unique challenges that do not exist when tracing an application contained within a single system.

The second requirement would require the creation of a similar command to `ldd` or `pmap` for EJBs. Similar to OS-specific applications, which can dynamically open libraries, EJBs can dynamically make use of other components. Despite the many descriptors that get deployed along with a bean, none of these define which other components might be used. This can only be learned by watching the communications of a bean over a period of time.

Finally, the third requirement states that tracing an execution flow and monitoring dependencies should not require modifying the beans themselves. The goal for this project is to add instrumentation in such a way that existing beans can be traced or monitored. In other words, a key goal of this project is to not require debug statements to be placed in the bean implementations.

4. Out-of-Band Communication Channel

Within an operating system, trace and dependancy information is made available by the kernel which brokers all system call invocations. The kernel, in a sense, provides a framework within which processes can run. In doing so, the operating system has insight into all operations that take place on the system.

The EJB architecture, on the other hand, allows for components to be located anywhere within the enterprise. An application can be made up of components that are deployed in more than one EJB container, located on multiple servers. Thus, unlike an operating system, a single container does not have visibility into all invocations and dependencies. A container only knows about those beans that are contained within it. Containers can participate in tracing method invocations and tracking dependencies but must rely on an external mechanism that takes an enterprise-wide perspective.

One approach is for information about method invocations to be captured locally by each container in a distributed environment. Such information is then communicated to a centralized entity that all containers know about. This communication path is termed the out-of-band channel.

To understand what is meant by an out-of-band channel, it is necessary to consider how

EJB components communicate in-band. The EJB specification as it exists today requires that all interfaces to the bean be defined in either the EJBLocalObject or EJBObject interfaces. These interfaces define the set of business methods that are accessible by local or remote clients. These interfaces are considered *in-band* interfaces. This term is used because it indicates that these interfaces are publicly accessible to other beans and can be invoked using standard EJB semantics. However, these interfaces are specified by the developer and therefore are not suitable for this project. Since it is a requirement for this project that the beans themselves need not be modified, it is not possible for the utilities created for this project to rely on specific business methods or method arguments.

Instead, an out-of-band communication path must be used to extract trace and dependancy information from the beans. An out-of-band communication channel provides endless possibilities for gathering information from EJB components. It could be used to accumulate performance or usage statistics for beans. Further, it could be used to send custom application data that is not suitable to be sent over the in-band channel to other beans. For this research, the out-of-band channel will be used to transport trace messages as bean methods are invoked in order to trace execution flow and identify dependencies.

5. Design Overview

The precise architecture of an out-of-band channel is one of the main concentrations of this research. Such a communication channel does not currently exist. However, this research will describe a prototype implementation in the context of the EJB architecture. It will also present a set of challenges that are posed by this mechanism.

To understand how such an out-of-band channel could be possible, it is important to examine the EJB architecture itself. Clients wishing to invoke a bean's business methods are able to locate the bean using a JNDI lookup. This lookup returns a pointer to a proxy. This proxy is generated dynamically by the application server and is download by the client. The proxy includes all code necessary for the client to communicate with the bean. Since this client code is generate dynamically, it is hypothesized that the application server must already be using introspection to examine the contents of beans. Therefore, it should be possible to extract information about method invocations at some point between when the proxy code is invoked and when the actual bean method is invoked.

This requires modifying the implementation of `javax.ejb.*`. Sun Microsystems defines these interfaces, which make up a part of the J2EE architecture. However, what Sun provides is simply a set of interfaces and not an implementation. Many vendors, such as JBoss, Sun, IBM, and BEA, have an implementation of these interfaces called an application server.

Thus, modifying `javax.ejb.*` means modifying the application server code that implements these interfaces. For this research, JBoss is used since it is an open source code base. JBoss is fully J2EE compliant and written completely in Java. It includes a set of EJB containers, one for each type of bean, and also a JNDI lookup service implementation.

The design of the out-of-band channel is built around a specialized entity bean that is reachable from anywhere in the JNDI namespace. This entity bean, which is called MonitorEJB, registers a special reserved JNDI name that all JBoss application servers will know about. JBoss, in turn, captures method invocations out-of-band and sends messages to the specialized entity bean.

MonitorEJB will acts as a service that is always listening for messages. There are many

reasons for this service being an entity bean. First, entity beans are persistent and can be accessed by multiple client simultaneously. This will allow multiple JBoss servers to send trace messages at the same time. Second, the service could have been implemented as a JMX MBean. The JBoss architecture uses JMX as it core and implements many of its features using MBeans. However, other EJB containers are not implemented using JMX. The decision to make the monitoring service an entity bean makes it possible for it to be deployed to any EJB container.

Figure 3 illustrates the design of the centralized monitoring service in the context of multiple JBoss containers.

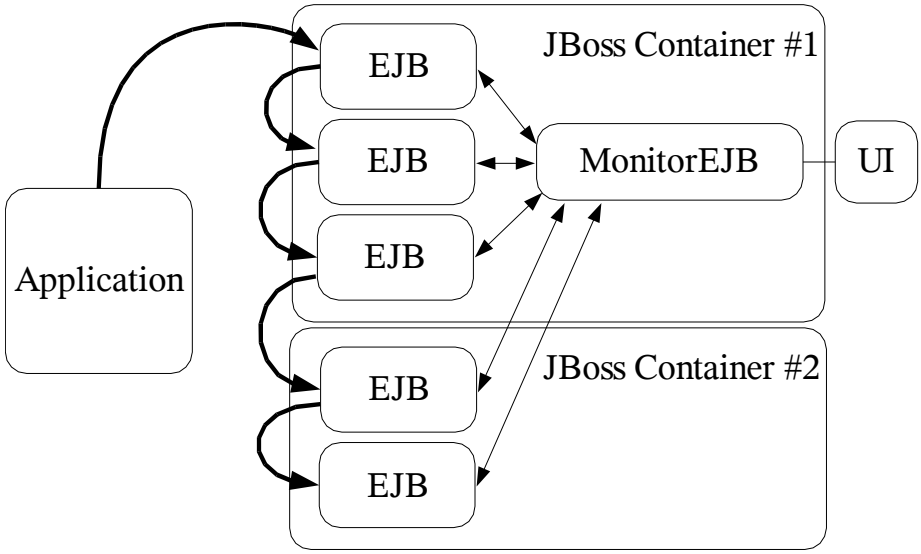

Figure 3: Centralized Monitoring Entity

In this diagram, there is an application that invokes one or more business methods of a given EJB. This EJB, in turn, invokes one or more business methods of another EJB, and so on. These EJBs are distributed over multiple JBoss containers, yet they each can communicate with the MonitorEJB. Although this diagram shows the MonitorEJB as co-located in the same container as some of the EJBs, it could be located in any container. Finally, the diagram shows a UI block that would display to the user information gathered by the MonitorEJB.

5.1 TraceInterceptor

JBoss' Interceptor Stack is used as the mechanism for capturing method invocations and sending these trace messages to the monitoring entity (MonitorEJB). The out-of-band channel architecture relies on a new Interceptor, called TraceInterceptor. Figure 4 illustrates the TraceInterceptor in the context of the JBoss Interceptor Stack.

The TraceInterceptor, like all Interceptors, contains a method called `invoke()` that takes an Invocation as an argument. The invocation itself is a mutable object that was created by the proxy, marshaled, and transported over the wire from the client. It contains the method name, arguments, and some context. The `invoke()` method can examine the invocation, modify it if necessary, and then pass the invocation to the next Interceptor.

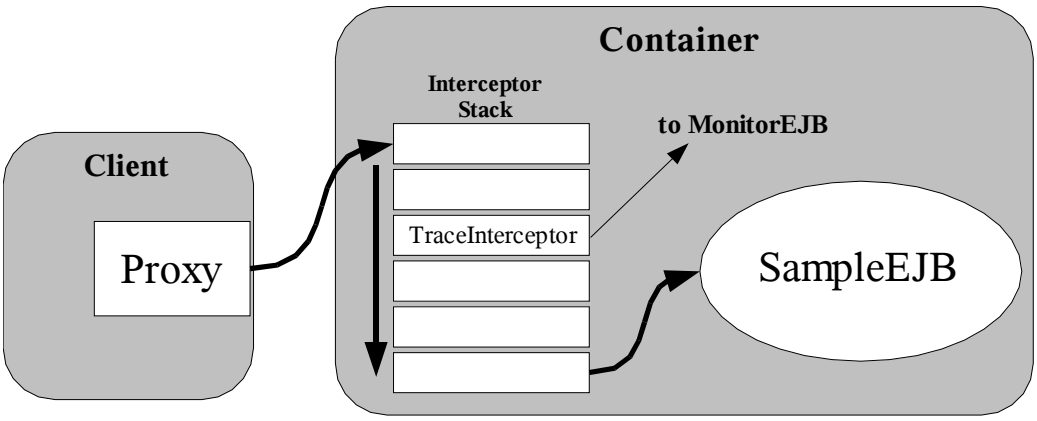

Figure 4: TraceInterceptor

TraceInterceptor's `invoke()` method parses the method arguments and packages them up in a form that can be sent to MonitorEJB. This is accomplished by concatenating the arguments together in a comma-separated String. The `invoke()` method in TraceInterceptor also retrieves the name of the bean whose method is being invoked and passes this information, along with the arguments and method name to MonitorEJB.

TraceInterceptor relies on a library called OOBUtils that contains helper methods for communicating with MonitorEJB. In particular, the `methodCalled()` method is called from OOBUtils which locates MonitorEJB and invokes its `methodCalled()` method. Finally, TraceInterceptor relinquishes control on the invocation by passing the method invocation to the next Interceptor.

5.2 MonitorEJB

The MonitorEJB entity bean is always running and is stateful. It can be located by performing a JNDI lookup using the reserved name "MonitorHome." There are two utilities, `ejbtruss` and `ejbdepends`, that retrieve trace and dependancy information from MonitorEJB and output it to the user. While most entity beans are backed by a database, MonitorEJB keeps everything in memory. Data persists until it is read by the `ejbtruss` or `ejbdepends` command.

However, the semantics of an entity bean hold true. MonitorEJB will keep track of primary keys such that each primary key corresponds to a separate thread of execution. The utilities must identify the appropriate primary key in order to fetch the correct execution trace. Although the design allows for MonitorEJB to handle multiple threads of execution, the current implementation only supports a single primary key.

MonitorEJB contains a method named `methodCalled()`. This method is a key part of the out-of-band channel. This method is invoked twice by the container from the TraceInterceptor each time a method is invoked. It is first invoked when a bean's business method is called. At this point, the business method name and arguments are passed to MonitorEJB. Second, it is invoked as the business method returns. This allows the return value of the business method to also be passed to the MonitorEJB.

MonitorEJB will keep track of the order in which methods are being invoked based on the order in which it receives trace messages. From this, it is not only able to produce a complete,

nested stack trace from beginning to end, but it is also able to understand the dependencies between beans. If a business method in one bean invokes the business method in another bean, the first bean is considered to have a dependancy on the second bean.

Each time a method is called, a new instance of the TraceEntry class is created which can store information about a single method invocation. TraceEntry instances are stored in an ordered list so that they can be retrieved in the order that methods were invoked.

MonitorEJB will keep a cache of the current component stack. Component names are pushed and popped from this stack. This allows a given call to `methodCalled()` to determine the previous component in the stack. From this information, a DependsEntry is created which stores a single relationship from one component to the other.

As a consequence for implementing the MonitorEJB as an EJB, rather than using some other technology such as JMX, there is the potential for infinite recursion to occur. For example, as the business methods of MonitorEJB are invoked by the TraceInterceptor, this invocation will itself be intercepted and traced by the MonitorEJB. Thus, the MonitorEJB avoids this by breaking the recursion if the bean name matches the reserved name for MonitorEJB.

5.3 The ejbtruss Command

In order for the user to retrieve a stack trace that was captured by MonitorEJB, a new utility called `ejbtruss` was created. This utility is actually a shell script that invokes the main method in the TraceClient class. The TraceClient class contains an infinite loop that continuously attempts to retrieve data from MonitorEJB using a specific primary key. MonitorEJB provides a business method called `getTrace()`. This method returns an ArrayList of TraceEntry instances. Upon copying this list to TraceClient, MonitorEJB will destroy it from its memory.

TraceClient will loop through these trace entries and produce a human-readable form a each entry. The output to `ejbtruss` will include nested method calls with arrows indicating whether the method is being called or is returning. The method arguments and return values will also be displayed.

5.4 The ejbdepends Command

In order for the user to retrieve information that was captured by MonitorEJB about the run-time invocation relationships between EJBs, a new utility called `ejbdepends` was created. Like `ejbtruss`, this utility is a shell script wrapper. It relies on a class called DependsClient. The shell script and main method take one string argument which is the name of a component. Using this name, DependsClient will invoke the `getDepends()` method of MonitorEJB. The name of the component is also passed via this method invocation. Depends client will return an ArrayList of Strings, where each String is the name of a component that is a dependancy.

If a given EJB component does not have any EJB components that it is dependent on, the ArrayList that is returned by `getDepends()` will be empty. The `ejbdepends` utility, in turn, will not produce any output. Lack of output is a standard Unix user-interface convention that means there is no data.

The list of EJBs that a given components is dependent on is not recursive. In other words,

it does include indirect dependencies. This is by design. Not only does this make it more consistent with the Unix commands, such as `ldd` and `pmap`, but it also more accurately reflects the dynamic nature of dependencies. For example, if component A depends on component B which in turn depends on component C, component A only depends on component C as long as component B chooses to use it. If at some point, component B chooses to use component D as a replacement for component C, this would not affect component A. Therefore, component A really only has a current run-time dependancy on component B.

6. Evaluation of the Out-of-Band Channel

In order to demonstrate the prototype implementation of the out-of-band channel, several very simple stateless session EJBs that perform basic mathematical tasks were created.
The design and setup of the prototype environment as well as the results of the evaluation are described below.

6.1 Demonstration Beans

There were four simple stateless session EJBs provided for evaluation purposes, which are interdependent. The first of these beans is AdderEJB. This stateless session contains one method, `add()` which takes two integer arguments and adds them together. A shell-script wrapper called `runadder` was created to run AdderEJB's business method from the command line.

Next, the SubtractTwoEJB session bean contains one method, `subtractTwo()`, takes one integer argument and returns an integer that is two less. The implementation of the SubtractTwoEJB in turn invokes the `add()` method from AdderEJB twice using -1 as one of the arguments. Although unnecessarily complicated, SubtractTwoEJB's reliance on AdderEJB is intended to demonstrate a dependancy. A shell-script wrapper called `runsubtracttwo` was created to run SubtractTwoEJB's business method from the command line.

A third bean created for demonstration purposes is RNumberEJB. This stateless session bean contains four methods: `getRandomPositiveEven()`, `getRandomPositiveOdd()`, `getRandomNegativeEven()`, `getRandomNegativeOdd()`. These methods return a random number that is either positive or negative and even or odd. A shell-script wrapper called `runrnumber` was created to run RNumberEJB's business methods from the command line.

Finally, a fourth stateless session bean called EnigmaEJB was created which contains one method, `run()`. This method, which takes one integer as an argument, is intentionally non-descriptive. The name of the bean and method are mysterious by design. This will demonstrate the power of the out-of-band channel in providing insight into the bean. A shell-script wrapper called `runenigma` was created to run EnigmaEJB's business method from the command line.

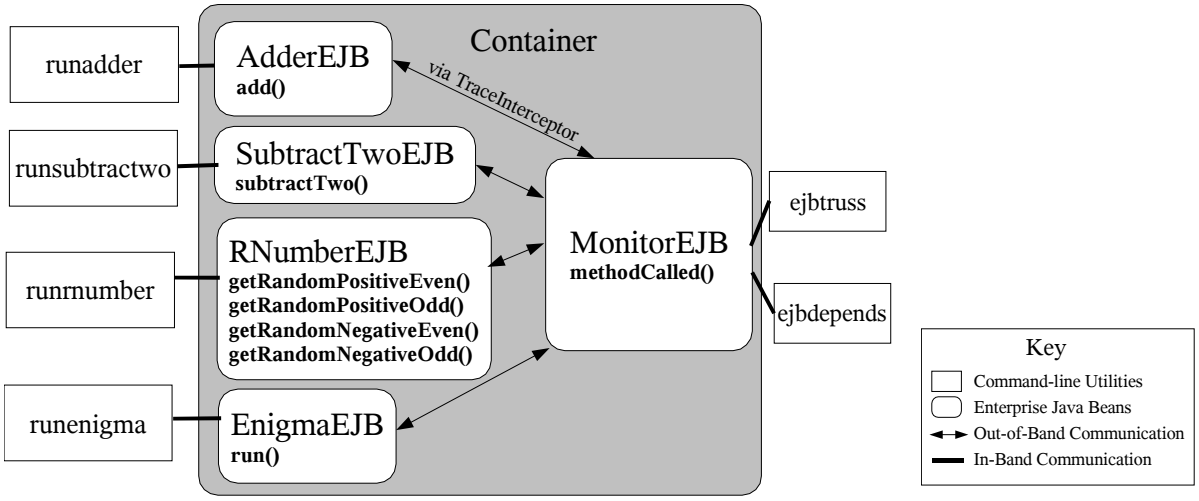

Figure 5: Demonstration Design

6.2 Setup and Deployment

Before starting JBoss, the TraceInterceptor must be activated for the stateless session bean container. This is accomplished by editing the Stateless Session Bean section of the `standardjboss.xml` file. This XML file contains all of the configuration state for the JBoss application server. The following line must be added from within the container-interceptors tag of the `standardjboss.xml` configuration file:

```
<interceptor>org.jboss.ejb.plugins.TraceInterceptor</interceptor>
```

For the TraceInterceptor, this line can be placed anywhere in the Interceptor Stack. This is because the TraceInterceptor does not modify the invocation or rely on anything but the method name and arguments. Other Interceptors require previous Interceptors to be invoked first.

Upon starting JBoss, each of the four stateless session beans as well as the MonitorEJB must be deployed to the container. This can be done by moving the jar files containing the beans to the deploy directory of the JBoss server. The beans could all be deployed to the same container or to different containers as long as a single JNDI server knows about all of the bean. The precise configuration of JNDI for this purpose is outside the scope of this document.

6.3 Experimentation and Results

Once setup and deployment were completed, the out-of-band channel was evaluated by using the `ejbtruss` and `ejbdepends` commands in combination with the `runadder`, `runsubtracttwo`, `runrnumber`, and `runenigma` test clients.

The following is example output of the `ejbtruss` command while executing the `runenigma` command with '1234' as the command-line argument:

```
-> EnigmaEJB:run(1234)
   -> RNumberEJB:getRandomPositiveOdd()
   <- RNumberEJB:getRandomPositiveOdd() = 1475228185
   -> AdderEJB:add(1234,1475228185)
   <- AdderEJB:add() = 1475229419
   -> SubtractTwoEJB:subtractTwo(1475229419)
      -> AdderEJB:add(1475229419,-1)
      <- AdderEJB:add() = 1475229418
      -> AdderEJB:add(1475229418,-1)
      <- AdderEJB:add() = 1475229417
   <- SubtractTwoEJB:subtractTwo() = 1475229417
<- EnigmaEJB:run() = 1475229417
```

Although the bean name and method name do not provide much insight into EnigmaEJB's functionilty, the output to `ejbtruss` illustrates what is happening. Each line of the `ejbtruss` output corresponds to a separate message that was passed over the out-of-band channel.

The first line of the truss output is the call to the `run()` method of EnigmaEJB with the argument that was passed via the command line utility, `runenigma`. The `run()` method then calls the `getRandomPositiveOdd()` method of RNumber which returns '1475228185'. The arrows indicate whether the method is being invoked or is returning. Because EnigmaEJB uses the RNumberEJB bean, we can conclude that EnigmaeEJB has a dependancy of RNumberEJB.

The argument to `run()` is then added to the random number using the `add()` method of AdderEJB. This means that EnigmaEJB also has a dependancy on AdderEJB. The result of the addition is then passed to the `subtractTwo()` method of SubtractTwoEJB. As the `ejbtruss` output shows, the call to `subtractTwo()` uses AdderEJB as part of its implmentation. The result of `subtractTwo()` is then returned by the `run()` method of Enigma at the end of the stack trace.

It should be noted that the output of `ejbtruss` only shows the business method of beans and not internal library calls that the bean implementations may be using. For example, the RNumberEJB uses class Random to generate a random number. However, ejbtruss is a component-level utility that understands EJB semantics and does not trace at the JVM-level.

As the `run()` method of EnigmaEJB was being executed, MonitorEJB was receiving trace messages and recording dependencies. Now that the execution is complete, MonitorEJB understands any dependencies that it observed. Using the `ejbdepends` utility, the user can retrieve this information.

For example, running `ejbdepends` with 'AdderEJB' as an argument will fetch the names of components that AdderEJB is dependant upon:

```
$ ./ejbdepends AdderEJB
```

The ejbdepends command returns no output which means that AdderEJB does not depend on any other beans. This is the correct behavior which can be validated by looking at the `ejbtruss` output as well as the source code.

When running ejbdepends with 'SubtractTwoEJB' as an argument produces the following output:

```
$ ./ejbdepends SubtractTwoEJB
   AdderEJB
```

This shows that SubtractTwoEJB depends on AdderEJB. From looking at the ejbtruss output, it is clear that the `subtractTwo()` method using the add method within AdderEJB to add -1 twice.

Finally, we can observe the dependencies of EnigmaEJB by running the `ejbdepends` command with 'EnigmaEJB' as the argument:

```
$ ./ejbdepends EnigmaEJB
   RNumberEJB
   AdderEJB
   SubtractTwoEJB
```

This output shows that EnigmaEJB depends on RNumberEJB, AdderEJB, and SubtractTwoEJB. Further, it can be assumed that EnigmaEJB would not run properly if it could not locate the other three EJB components.

The output of the `ejbtruss` and `ejbdepends` commands confirm that the out-of-band channel is working properly. It is clear that method invocations are being intercepted and that messages are being propagated from the TraceInterceptor to the MonitorEJB. The trace output and dependancy information was determined to be correct by examining source code.

The set of demonstration beans fully exercises the out-of-band channel. From the perspective of a single method invocation, which is the level at which the TraceInterceptor operates, there is little variation from one method invocation to the next beyond the number and type of arguments. Since any invocation can be introspected using standard Java library calls, it is likely that the out-of-band channel could be used to send trace messages relating to any method invocation, regardless of the number or type of arguments. Thus, the results from this demonstration suggest that the out-of-band channel could work with any EJB component.

6.4 Discussion

The design outlined in this document has two serious limitations that will reduce its general applicability. First, this approach assumes a single path of execution within the enterprise. In reality, application servers handle many clients simultaneously and use a multi-threaded model for increasing performance. While the MonitorEJB has the ability to separate execution flows based on unique primary keys, the TraceInterceptor, which sends messages to MonitorEJB, does not have sufficient contextual information to demultiplex the execution flows. In other words, the out-of-band channel currently operates as one large channel, rather than providing small, separate channels for each path of execution.

Secondly, the current design relies on trace messages that are sent from the container to the MonitorEJB to be in order. If these messages become out of order, this would disrupt the logic within this specialized entity bean. Network latency and race conditions make this situation possible, however unlikely. A mechanism, such as a sequence number would be needed to order these messages. In short, messages that travel across the out-of-band channel are not currently guaranteed to remain in order. However, since these messages can travel across an enterprise, there would need to be a mechanism for propagating sequence numbers throughout the path of execution. Architectural enhancements to the out-of-band channel are necessary to address these two major limitations. This will not only require augmentation of the new components introduced by this research, but also require wholesale changes to the application server design and architecture.

In order to track separate execution flows simultaneously and to ensure that messages are ordered, the notion of an execution flow "cookie" or context must exist. This context could be passed from invocation to invocation, container to container and would persist for the life of execution flow. Within the context, a unique identifier would differentiate one execution flow from another. The context could also include a sequence number that is incremented by some container each time a method invocation is intercepted as part of that execution flow.

Figure 6 illustrates an execution flow context which is created when the execution flow begins at the client and propagates from bean to bean, container to container.

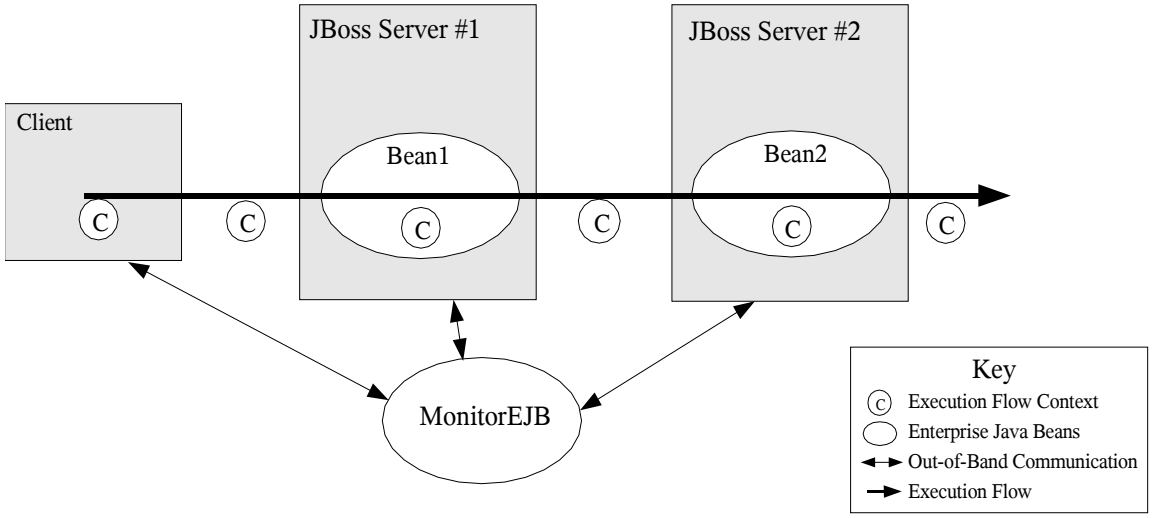

Figure 6: Execution Flow Context Propagation

From the perspective of MonitorEJB, it would be simple to use the execution flow context to demultiplex messages as they are received and to guarantee order. However, the tracking of execution flow contexts across the enterprise is certain to be difficult. Each container must understand the concept of a *network thread* and preserve the semantics of such a concept whenever the flow of execution travels through that container. In order for an application server to support execution flow contexts, there must be a way to transport additional information from the caller to the container, possibly over the wire. Currently, JBoss marshals the method name and arguments along with the invocation context information and sends this information over the wire. It is the dynamic proxy that is responsible for setting up the context information and for marshaling.

The dynamic proxies could also be used to marshal the execution flow context along with the invocation context, method name, and method arguments. A given proxy will also have to look for an existing context for the given execution flow or create a new execution flow context if it does not exist. Likewise, the containers must be extended to unmarshal the execution flow contexts. This would be involve extending the existing invocation handlers.

Since dynamic proxies are generated by the server but downloaded to the client, they execute in the caller's environment. This is a powerful tool for the application server to arrange for the client to execute specific code. Existing clients and beans would not need to change in order to begin propagating execution flow contexts. Instead, an enhanced dynamic proxy would be provided by the server.

The addition of execution flow contexts also presents further unique challenges. Conceptually, a bean handles incoming invocations and executes outgoing invocations on

other beans. From the perspective of a single bean, it is difficult to correlate incoming invocations with out going invocations. For example, consider the example of SubtractTwoEJB. It relies on AdderEJB to implement its logic. It handles incoming invocations when its `subtractTwo()` method is invoked. In addition, it makes outgoing invocations by calling the `add()` method in AdderEJB.

When `subtractTwo()` is called by multiple clients, the `add()` method will then be called multiple times. Since `subtractTwo()` is called more than once, there will be multiple execution flow contexts that need to be tracked for the same path of execution. The `subtractTwo()` method will download a dynamic proxy for communicating with AdderEJB. The proxy could look for an existing context in its environment that was left by the incoming invocation in the form a static variable. However, it will be difficult for the proxy to select which existing execution flow context to associate with which outgoing call to the `add()` method if multiple execution flow contexts exist statically in the proxy's environment. It is likely that some key Java technologies such as dynamic proxies or the EJB interfaces themselves would need to be extended to solve this problem.

Assuming that a solution could be found such that the correct execution flow context be propagated from proxy to container, the container must be able to track these execution flow contexts internally. This would likely involve keeping a cache of method invocations as well as preserving state for all bean instances. Unfortunately, such an approach is counter to some of the core architectural goals for most application servers.

More specifically, when a client performs a lookup on a component and downloads proxy code, the client can interact with the bean as if there is a corresponding server-side instance of the bean for its sole use. In reality, the EJB container goes to great lengths to reduce the number of actual bean instances necessary on the server. Unlike in pure RMI, where each client object is communicating directly with an actual server-side object, the JBoss container handles all method invocations. For most types of beans, such as stateless session beans and entity beans, only one cached instance is necessary to handle multiple clients. One exception is for stateful session beans where there must be an instance for each client.

Tracking execution flow contexts would require some representation of each bean instance within the container in order to associate an execution flow context with it. This, in turn, would produce additional overhead and reduce performance. At the very least, the ability to track execution flow contexts would be an optional feature that users could disable to improve performance.

The dynamic proxy technology as well as the EJB specification is insufficient to adequately address these design roadblocks today. While it is possible to approximate a solution to this problem, each approach contains limitations that will prevent it from having universal appeal.

7. Related Work

This research relates to work previously done in the area of using the JBoss Interceptor Stack to extend EJB containers. The concept that commercial EJB containers need to be extended to provide adequate predictable assembly of components is not new. It has been previously demonstrated by Gary Vecellio et al. at MITRE that services can be added to the container that make assembly more predictable [3]. Services that enforced policies at run-time were created. This was accomplished by allowing assertions to be made about the assembly of

components. The Interceptor Stack was used to monitor interactions between components. However, this research seems to assume that all components are located within a single container.

Work by K. Simons and J.A. Stafford demonstrate the use of the JBoss Interceptor Stack to handle exceptions [2]. This innovative research allows the container to manage exceptions in order to increase the predictability and reliability of component behavior. The Interceptor Stack not only used to recover from exceptions but also used to prevent exceptional behavior in the the first place.

Very little work has previously been done relating to the stack tracing of execution flows across EJBs in an enterprise. However, J. Lambert and A. Podgurski have developed a way to capture and analyze stack traces in a distributed Java environment [1]. They created a tool call xdProf which receives stack trace transmissions at fixed intervals from a set of Java Virtual Machines (JVMs). They provide graphic tools for analyzing the stack traces. The xdProf tools do not operate at the EJB component or container level. Instead, it traces all Java library calls.

In the commercial space, Quest Software produces an application called PerformaSure which can monitor the performance of a multi-tiers J2EE system even in a clustered, multi-system environment. PerformaSure can trace and reconstruct the execution path across the entire J2EE system in order to evaluate its performance. They rely on network snooping and on agents that gather information from each server and transport this information to a central hub. A Quest white paper suggests that stack trace information is retrieved by automatic instrumentation of the Java class loader [4].

8. Future Work

Additional work is necessary to determine whether the limitations outlined can be solved without modifications to core Java technologies and specifications. It remains unclear whether JBoss could support execution flow contexts and still remain J2EE compliant.

Further, more research is needed to understand how the out-of-band channel could be used to transport other types of information, such as performance statistics, and what applications this might have.

Lastly, it is not fully understood what side effects may exist to utilizing an out-of-band channel. Although a decrease in performance is likely, it is not clear to what extent performance is reduced and whether this could prevent some applications from functioning properly.

8. Conclusions

It is clear from this research that there is significant potential for enhancing component frameworks to provide additional development and diagnostic tools. The ability to trace an execution flow at a component level and to study the dependencies between components is a compelling feature that undoubtedly would reduce complexity and increase the proliferation of commercial components. However, significant work is necessary to achieve parity between the tools provided by operating systems and those provide by component frameworks, such as EJB containers.

Monitoring applications that are distributed across an enterprise is complex and challenging. This researches provides a basic approach to centralizing the monitoring of EJBs for the purpose of learning more about them. However, it does not fully address the issues it raises. Cooperation between authors of the EJB specification and the companies and groups that provide implementations is necessary in order to provide a complete solution.

9. References

[1] J. Lambert and A. Podgurski. xdProf: A Tool for the Capture and Analysis of Stack Traces in a Distributed Java System, *Proceedings of the SPIE – The International Society for Optical Engineering*, 2001.

[2] K. Simons and J.A. Stafford. CMEH: Container-Managed Exception Handling for Increased Assembly Robustness. *Component-Based Software Engineering 7th International Symposium*, CBSE 2004. May 24-25, 2004.

[3] G. Vecellio, W. Thomas, and R. Sanders. Containers for the Predictable Behavior of Component-based Software. *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, 2002.

[4] G. Vona. Measuring J2EE Application Performance in Production. Quest Software. http://www.quest.com.

Appendix A: TraceInterceptor Source Code

A.1 TraceInterceptor.java

```
package org.jboss.ejb.plugins;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.ServerError;
import java.rmi.ServerException;
import java.util.Map;

import javax.ejb.EJBException;
import javax.ejb.NoSuchEntityException;
import javax.ejb.NoSuchObjectLocalException;

import org.jboss.ejb.Container;
import org.jboss.invocation.Invocation;
import org.jboss.invocation.InvocationType;
import org.jboss.metadata.BeanMetaData;

public class TraceInterceptor extends AbstractInterceptor
{
   protected String ejbName;
   protected boolean callLogging;

   public void create()
      throws Exception
   {
      super.start();

      BeanMetaData md = getContainer().getBeanMetaData();
      ejbName = md.getEjbName();

   }

   public Object invokeHome(Invocation invocation)
      throws Exception
   {
      try
      {
       return getNext().invokeHome(invocation);
      }
      catch(Exception e)
      {
         throw e;
      }
   }

   public Object invoke(Invocation invocation)
      throws Exception
   {
```

```
      String methodName;
      if (invocation.getMethod() != null)
      {
         methodName = invocation.getMethod().getName();
      }
      else
      {
         methodName = "<no method>";
      }


      // Organize the method arguments into a buffer
      StringBuffer arglist = new StringBuffer();

      Object[] args = invocation.getArguments();
      if (args != null)
      {
         for (int i = 0; i < args.length; i++)
         {
            if (i > 0)
            {
               arglist.append(",");
            }
            arglist.append(args[i]);
          }
      }
      String arglistString = arglist.toString();

      // Send a trace message to the MonitorEJB
      OOBUtils.methodCalled(ejbName, methodName, arglistString, null);

      try
      {
       Object ret = getNext().invokeHome(invocation);
         Integer retval = (Integer)ret;

         // Send the return value to the MonitorEJB
         OOBUtils.methodCalled(ejbName, methodName, null,
                   retval.toString());
         return ret;
      }
      catch(Exception e)
      {
       throw e;
      }
   }

}
```

A.2 OOBUtils.java (Helper methods for TraceInterceptor)

```
package org.jboss.ejb.plugins;

import javax.rmi.*;
```

```
import javax.naming.*;
import java.util.*;
import org.jrosen.Monitor.*;

public class OOBUtils {

 // Called remotely each time a bean's business method is invoked.
 public static void methodCalled (String componentname, String methodname,
            String args, String retval) {

        try{

          Properties p = new Properties();

          p.put(Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
          p.put(Context.URL_PKG_PREFIXES,
                  "jboss.naming:org.jnp.interfaces");
          p.put(Context.PROVIDER_URL, "localhost:1099");

          //Now use those properties to create
          //a JNDI InitialContext with the server.
          InitialContext ctx = new InitialContext(p);

          //Lookup the MonitorEJB in order to send it a message
          Object obj = ctx.lookup("/MonitorHome");

          MonitorHome mHome = (MonitorHome)
            PortableRemoteObject.narrow(obj,MonitorHome.class);

          MonitorObject ejbObject;

          try {
             // Check to see if a stack trace is already in progress
             // (The key should be a session identifier.)
             ejbObject = mHome.findByPrimaryKey("oobtrace");
          } catch (javax.ejb.FinderException fe) {

             // Else start a new trace
             ejbObject = mHome.create("oobtrace");
          } finally {
          }

          // Invoke the remote method, sending the method name,
          // arguments, and return value (if present) to MonitorEJB
          ejbObject.methodCalled(componentname, methodname, args, retval);

        } catch (Exception e){
          e.printStackTrace();
        }
   }
```

Appendix B: MonitorEJB Source Code

B.1 MonitorEJB.java

```
package org.jrosen.Monitor;

import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.ArrayList;
import java.util.Hashtable;

public class MonitorEJB implements EntityBean {
  private EntityContext entityContext;
  private ArrayList traceList;
  private ArrayList dependsList;
  private ArrayList componentstack;
  String name;

  public String ejbCreate(String n) throws javax.ejb.CreateException {
     // Initialize all in-memory data structures.
     traceList = new ArrayList();
     dependsList = new ArrayList();
     componentstack = new ArrayList();
     this.name = n;
     return n;
  }

  public void ejbPostCreate(String foo) {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void ejbLoad() {
     traceList = new ArrayList();
     dependsList = new ArrayList();
     componentstack = new ArrayList();
  }
  public void ejbStore() {
  }
  public String ejbFindByPrimaryKey(String primarykey) {
     // Limitation: This MonitorEJB currently only handles one thread.
     return primarykey;
  }
  public void setEntityContext(EntityContext entityContext) {
     this.entityContext = entityContext;
  }
  public void unsetEntityContext() {
     this.entityContext = null;
  }
```

```
// Receives a trace message that a method has been invoked.
public void methodCalled(String componentname, String methodname,
          String args, String retval)
                throws java.rmi.RemoteException {
   TraceEntry te = new TraceEntry (componentname, methodname, args,
                           retval);

   // Register the message
   traceList.add(te);

   // Method is being called, determine dependancy relationship.
   if (retval == null) {
         // Keep track of the previous component using a stack
         // representation.
         if (componentstack.size() >  0) {
               String callercomponent = (String)componentstack.get
                     (componentstack.size()-1);
               componentDepends(callercomponent, componentname);
         }
         componentstack.add(componentname);
   }

   // Method is returning
   else {
         // Returning...take last component off stack
         componentstack.remove(componentstack.size()-1);
   }

}

// Register a dependancy between two componenets.
private void componentDepends(String componentA, String componentB)
                     throws java.rmi.RemoteException {
   // Check to see if we already know about the dependency
   ArrayList currentdepends = getDepends(componentA);
   for (int i=0; i< currentdepends.size(); i++) {
    if (componentB.equals(currentdepends.get(i))) return;
   }

   // Create a new dependancy record and add it to the list.
   DependsEntry de = new DependsEntry (componentA, componentB);
   dependsList.add(de);
}

// Get a stack trace for a given execution flow
public ArrayList getTrace() throws java.rmi.RemoteException {

    // Save the old list to return and then make a new one
    ArrayList ret = traceList;

    traceList = new ArrayList();
    return ret;
}
```

```
  // Get a list of dependencies for a given component.
  public ArrayList getDepends(String componentname) {
      // Generate a new list
      ArrayList returnlist = new ArrayList();

      // Search through all known dependencies and return only those
      // that apply.
      for (int i=0; i< dependsList.size(); i++) {
            DependsEntry dtemp = (DependsEntry)dependsList.get(i);
            if (dtemp != null &&
                  dtemp.getA().compareTo(componentname) == 0) {
                  returnlist.add(dtemp.getB());
            }
      }
      return returnlist;
  }

}
```

B.2 DependsEntry.java (Stores a single dependancy)

```
package org.jrosen.Monitor;

import java.util.ArrayList;
import java.io.*;

public class DependsEntry implements Serializable {
   private String componentA;
   private String componentB;

   public DependsEntry (String ca, String cb) {
      componentA = ca;
      componentB = cb;
   }

   public String getA() {
      return componentA;
   }

   public String getB() {
      return componentB;
   }
}
```

B.2 TraceEntry.java (Stores a single trace message)

```
package org.jrosen.Monitor;

import java.util.ArrayList;
import java.io.*;

public class TraceEntry implements Serializable {
```

```
   private String componentname;
   private String methodname;
   private String retval;
   private String args;

    public TraceEntry (String cname, String mname, String args1, String r1)
{
      componentname = cname;
      methodname = mname;
      retval = r1;
      args = args1;
   }

   public String getComponentName() {
      return componentname;
   }

   public String getMethodName() {
      return methodname;
   }

   public String getArgs() {
      return args;
   }
   public String getRetVal() {
      return retval;
   }

}
```

B.3 MonitorHome.java

```
package org.jrosen.Monitor;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface MonitorHome extends EJBHome {
  public MonitorObject create(String foo) throws RemoteException,
            CreateException;
  public MonitorObject findByPrimaryKey(String foo) throws
            RemoteException, FinderException;

}
```

B.4 MonitorObject.java

```
package org.jrosen.Monitor;

import java.rmi.*;
```

```
import javax.ejb.*;
import java.util.*;

public interface MonitorObject extends EJBObject {
  public void methodCalled (String componentname, String methodname,
            String args, String retval) throws RemoteException;
  public ArrayList getTrace() throws RemoteException;
  public ArrayList getDepends(String componentname) throws
            RemoteException;
  }
```

Appendix C: MonitorEJB Clients Source Code

C.1 TraceClient.java (Outputs a stack trace to the user)

```
package org.jrosen.Monitor;

import javax.rmi.*;
import javax.naming.*;
import java.util.*;

public class TraceClient {

 public static void main( String args[]) {

  int depth = 0;

  // Start off with a blank line
  System.out.println("");

  while (true) {
        try{

          Properties p = new Properties();

          p.put(Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
          p.put(Context.URL_PKG_PREFIXES,
                  "jboss.naming:org.jnp.interfaces");
          p.put(Context.PROVIDER_URL, "localhost:1099");

          //Now use those properties to create
          //a JNDI InitialContext with the server.
          InitialContext ctx = new InitialContext(p);

          //Lookup the MonitorEJB
          Object obj = ctx.lookup("/MonitorHome");

          MonitorHome mHome = (MonitorHome)
            PortableRemoteObject.narrow(obj,MonitorHome.class);

          MonitorObject ejbObject;

          try {
             // Lookup an existing stack trace in progress
             ejbObject = mHome.findByPrimaryKey("oobtrace");
          } catch (javax.ejb.FinderException fe) {

             // If it doesn't exist create one (should not be the case)
             ejbObject = mHome.create("oobtrace");
          } finally {
          }

          // Download the stack trace.
```

```
  ArrayList trace = ejbObject.getTrace();


  // Loop through each trace entry in the stack trace and
  // output it to the user one at a time.

  for (int i=0; i<trace.size(); i++) {
    TraceEntry te = (TraceEntry)trace.get(i);

    // Extract the fields from the trace entry.
    String cname = te.getComponentName();
    String mname = te.getMethodName();
    String ret = te.getRetVal();
    String arglist = te.getArgs();
    String depthstring = "";

    // Nesting logic...
    // Add a few spaces for each depth
    for (int j=0; j<depth; j++) {
          depthstring = depthstring + "   ";
    }

    // If there is not return value, it must be a method call
    if (ret == null) {
          System.out.println(depthstring + "-> "
                + cname + ":" + mname + "(" + arglist + ")");

          // Increase the depth for each call
          depth++;

    // There is a return value so it is a method return
    } else {
          // Decrease the depth because the method is returning.
          depth--;
          depthstring = "";
          for (int j=0; j<depth; j++) {
                depthstring = depthstring + "   ";
          }
          System.out.println(depthstring + "<- "
                + cname + ":" + mname + "() = " + ret);
    }

     if (i+1 == trace.size()) {
      // If the top level method returns print two empty lines
      // as a separator.
      System.out.println("\n\n");
     }

  }

// Wait for more data from MonitorEJB
Thread.sleep(300);

} catch (Exception e){
  e.printStackTrace();
}
```

```
  }
 }
}
```

C.2 DependsClient.java (Outputs a dependancy list to the user)

```
package org.jrosen.Monitor;

import javax.rmi.*;
import javax.naming.*;
import java.util.*;

public class DependsClient {

 public static void main( String args[]) {

  int depth = 0;

  try{

    Properties p = new Properties();

    p.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
    p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
    p.put(Context.PROVIDER_URL, "localhost:1099");

    //Now use those properties to create
    //a JNDI InitialContext with the server.
    InitialContext ctx = new InitialContext(p);

    //Lookup the MonitorEJB
    Object obj = ctx.lookup("/MonitorHome");

    MonitorHome mHome = (MonitorHome)
      PortableRemoteObject.narrow(obj,MonitorHome.class);

    MonitorObject ejbObject;

    try {
       // Look up an existing execution flow within the bean.
       ejbObject = mHome.findByPrimaryKey("oobtrace");
    } catch (javax.ejb.FinderException fe) {
       ejbObject = mHome.create("oobtrace");
    } finally {
    }

    // Pass the componnt name to MonitorEJB and retrieve a listening
    // of dependent components.
    ArrayList depends = ejbObject.getDepends(args[0]);

    for (int i=0; i<depends.size(); i++) {
            String dep = (String)depends.get(i);
```

```
            System.out.println("   " + dep);
    }

  } catch (Exception e){
    e.printStackTrace();
  }

 }
}
```

Appendix D: AdderEJB Demo Bean Source Code

D.1 AdderEJB.java

```
package com.rosen.Adder;

import java.rmi.RemoteException;
import javax.ejb.*;

public class AdderEJB implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
  }
  public int add(int number, int amount) throws java.rmi.RemoteException {
    int newnumber = number + amount;
    return newnumber;
  }

}
```

D.2 AdderHome.java

```
package com.rosen.Adder;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface AdderHome extends EJBHome {
  public AdderObject create() throws RemoteException, CreateException;
}
```

D.3 AdderObject.java

```
package com.rosen.Adder;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface AdderObject extends EJBObject {
  public int add(int number, int amount) throws RemoteException;
```

```
}
```

Appendix E: SubtractTwoEJB Demo Bean Source Code

E.1 SubTractTwoEJB.java

```
package com.rosen.SubtractTwo;

import javax.ejb.*;
import javax.rmi.*;
import javax.naming.*;
import java.util.*;
import com.rosen.Adder.*;

public class SubtractTwoEJB implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
  }
  public int subtractTwo(int number) throws java.rmi.RemoteException {

    int newnumber = 0;
    try {
          Properties p = new Properties();

          p.put(Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
          p.put(Context.URL_PKG_PREFIXES,
                  "jboss.naming:org.jnp.interfaces");
          p.put(Context.PROVIDER_URL, "localhost:1099");

          InitialContext ctx = new InitialContext(p);

          Object obj = ctx.lookup("/AdderHome");
          AdderHome ejbHome = (AdderHome)
          PortableRemoteObject.narrow(obj,AdderHome.class);

          AdderObject ejbObject = ejbHome.create();

          newnumber = ejbObject.add(number, -1);
          newnumber = ejbObject.add(newnumber, -1);

    } catch (Exception e){
       e.printStackTrace();
    }

    return newnumber;
```

```
  }
}
```

E.2 SubtractTwoHome.java

```
package com.rosen.SubtractTwo;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface SubtractTwoHome extends EJBHome {
        public    SubtractTwoObject    create()    throws    RemoteException,
CreateException;
}
```

E.3 SubtractTwoHome.java

```
package com.rosen.SubtractTwo;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface SubtractTwoObject extends EJBObject {
  public int subtractTwo(int number) throws RemoteException;
}
```

Appendix F: RNumberEJB Demo Bean Source Code

F.1 RNumberEJB.java

```
package com.rosen.RNumber;

import javax.ejb.*;
import javax.rmi.*;
import javax.naming.*;
import java.util.*;

public class RNumberEJB implements SessionBean {
  private SessionContext sessionContext;
  private Random r;
  public void ejbCreate() {
      r = new Random();
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
  }
  public int getRandomPositiveEven() throws java.rmi.RemoteException {

    int number = r.nextInt();
    if (number % 2 == 1) { number++; }
    if (number < 0 ) { number = number * -1; }

    return number;
  }

  public int getRandomPositiveOdd() throws java.rmi.RemoteException {

    int number = r.nextInt();
    if (number % 2 == 0) { number++; }
    if (number < 0 ) { number = number * -1; }

    return number;
  }

  public int getRandomNegativeEven() throws java.rmi.RemoteException {

    int number = r.nextInt();
    if (number % 2 == 1) { number++; }
    if (number > 0 ) { number = number * -1; }

    return number;
  }

  public int getRandomNegativeOdd() throws java.rmi.RemoteException {
```

```
    int number = r.nextInt();
    if (number % 2 == 0) { number++; }
    if (number > 0 ) { number = number * -1; }

    return number;
  }
}
```

F.2 RNumberHome.java

```
package com.rosen.RNumber;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface RNumberHome extends EJBHome {
  public RNumberObject create() throws RemoteException, CreateException;
}
```

F.3 RNumberObject.java

```
package com.rosen.RNumber;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface RNumberObject extends EJBObject {
  public int getRandomPositiveOdd() throws java.rmi.RemoteException;
  public int getRandomPositiveEven() throws java.rmi.RemoteException;
  public int getRandomNegativeOdd() throws java.rmi.RemoteException;
  public int getRandomNegativeEven() throws java.rmi.RemoteException;
}
```

Appendix G: EnigmaEJB Demo Bean Source Code

G.1 EnigmaEJB.java

```
package com.rosen.Enigma;

import javax.ejb.*;
import javax.rmi.*;
import javax.naming.*;
import java.util.*;

import com.rosen.Adder.*;
import com.rosen.SubtractTwo.*;
import com.rosen.RNumber.*;

public class EnigmaEJB implements SessionBean {
  private SessionContext sessionContext;
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
  }
  public int run(int number) throws java.rmi.RemoteException {

    int newnumber = 0;
    try {
          Properties p = new Properties();

          p.put(Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
          p.put(Context.URL_PKG_PREFIXES,
                  "jboss.naming:org.jnp.interfaces");
          p.put(Context.PROVIDER_URL, "localhost:1099");

          InitialContext ctx = new InitialContext(p);

          Object obj3 = ctx.lookup("/RNumberHome");

          RNumberHome ejbHome3 = (RNumberHome)
          PortableRemoteObject.narrow(obj3,RNumberHome.class);

          RNumberObject ejbObject3 = ejbHome3.create();

          int randomnumber = ejbObject3.getRandomPositiveOdd();

          Object obj = ctx.lookup("/AdderHome");

          AdderHome ejbHome = (AdderHome)
```

```
            PortableRemoteObject.narrow(obj,AdderHome.class);

            AdderObject ejbObject = ejbHome.create();

            newnumber = ejbObject.add(number, randomnumber);

            Object obj2 = ctx.lookup("/SubtractTwoHome");

            SubtractTwoHome ejbHome2 = (SubtractTwoHome)
            PortableRemoteObject.narrow(obj2,SubtractTwoHome.class);

            SubtractTwoObject ejbObject2 = ejbHome2.create();

            newnumber = ejbObject2.subtractTwo(newnumber);


        } catch (Exception e){
           e.printStackTrace();
        }

        return newnumber;

    }
}
```

G.2 EnigmaHome.java

```
package com.rosen.Enigma;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface EnigmaHome extends EJBHome {
  public EnigmaObject create() throws RemoteException, CreateException;
}
```

G.3 EnigmaObject.java

```
package com.rosen.Enigma;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface EnigmaObject extends EJBObject {
  public int run(int number) throws RemoteException;
}
```