An Infrastructure for the Dynamic Distribution of Web Applications and Services[1]

Enrique Duvos Azer Bestavros

Department of Computer Science
Boston University

{eduvos,best}@cs.bu.edu

December 2000

BUCS-TR-2000-027

Abstract

This paper presents the design and implementation of an infrastructure that enables any Web application, regardless of its current state, to be stopped and uninstalled from a particular server, transferred to a new server, then installed, loaded, and resumed, with all these events occurring "on the fly" and totally transparent to clients. Such functionalities allow entire applications to fluidly move from server to server, reducing the overhead required to administer the system, and increasing its performance in a number of ways: (1) Dynamic replication of new instances of applications to several servers to raise throughput for scalability purposes, (2) Moving applications to servers to achieve load balancing or other resource management goals, (3) Caching entire applications on servers located closer to clients.

[1] This work was supported by NSF (Grant #9986397) and was completed as part of the first author's Masters project at Boston University.

An Infrastructure for the Dynamic Distribution of Web Applications and Services

Enrique Duvos Azer Bestavros

Department of Computer Science
Boston University

Introduction

Motivation

During the past few years, the Internet has become an almost indispensable medium for many businesses and individuals. Thousands of users rely on the Internet daily to perform fast and reliable transactions, and this number is rapidly rising. But as the amount of users demanding high quality service from their Internet application grows, the systems supporting those services become more and more overwhelmed, increasing the chance of long delays, and sometimes, failing to meet client requests.

To keep up with this pace, the software architectures supporting those applications have evolved over the years, going from pure mainframe systems to client-server models, to the three-tier architectures widely used nowadays. These systems typically involve web, application, and database servers collaborating with each other to distribute the workload. But even though these architectures greatly improve the scalability of the services they support, the quest for faster and more reliable practices seems to be one of the hottest fields within the software development community. A good example of this work is the continuous development in areas such as content caching, content replication, and load balancing.

Although different in concept, these procedures share the same goal, to avoid long response times and augment the number of client requests served. Caching and replication techniques reduce response times by moving and distributing content closer to the end user, speeding content searches, and reducing client-server communication. On the other hand, load-balancing techniques boost the overall system by distributing requests among a set of servers, also referred to as a cluster. To achieve this level of distribution, all servers in the cluster must run a copy of any application that is to be balanced. This implies that any new application must be installed on every server in the cluster, and that any new server added to the cluster must be configured with all the applications currently running on other servers. The end result is a system where client requests can be directed and executed by any server, and where parameters such as server load or network distance to the server are used to determine a request's destination.

Problem definition

Unfortunately, the techniques mentioned above present certain drawbacks as the number of servers and applications increase:

- Upgrading applications among servers or adding new servers to a cluster can become a complex and time-consuming operation.
- Servers need to be manually pre-configured (new software must be installed) in order to accept new applications.
- Simple and fast applications do not perform as expected, since they compete for resources with high demand tasks running on the same server.
- Applications running on nearby servers produce shorter response times.

The solution proposed in this abstract addresses the idea that any application, regardless of its current state, could be stopped and uninstalled from a particular server, transferred to a new one, then installed, loaded, and resumed, with all these events occurring “on the fly” and totally transparent to clients. Thus, the migration software infrastructure will allow entire applications to dynamically move from server to server, reducing the overhead required to administer the system, and increasing its performance in these ways:

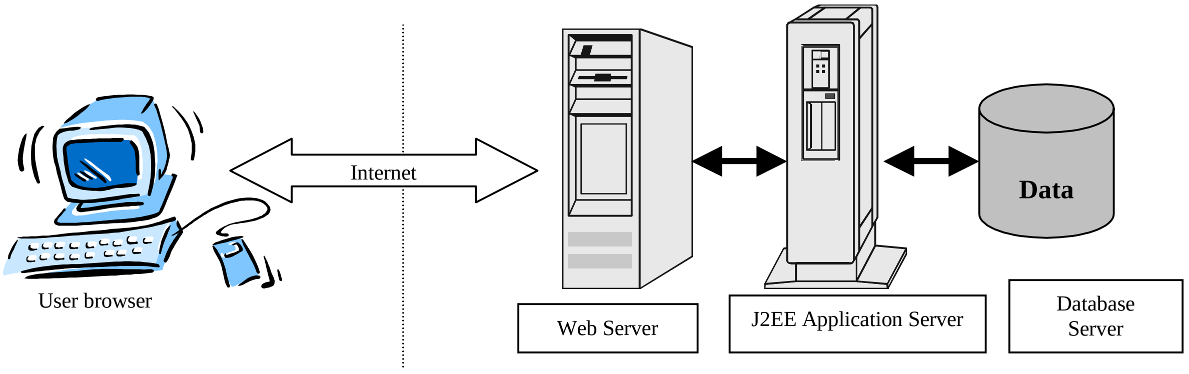
- Distributing new application instances among several servers to raise throughput.
- Moving frequently used applications to relatively inactive servers, where applications do not compete among each other for resources, and requests can be handled faster.
- Caching entire applications on servers located closer to clients, boosting response times by reducing network distances. (This concept is similar to “content caching”, where frequently accessed content is moved to servers located closer to final delivery destinations).

It is important to note that the term “application“ in this context refers to any independent software program, code, and data, that has been installed on a particular server, and that is currently running, idle, or stopped.

Environment

Although the following design could be generalized to other software components and architectures, my work concentrates on the application server module part of the three-tier architecture, and in particular, on Enterprise JavaBeans™ servers that follow the current Java™ 2 Platform, Enterprise Edition (J2EE™) specification from Sun Microsystems. Therefore, familiarity with this kind of technology is presumed throught the entire abstract.

J2EE Three-tier Architecture



The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture, are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

Enterprise JavaBeans™, or EJBs, are server-side, self-contained reusable software components that can be run on any EJB-compliant server. Beans define the business logic necessary to implement a particular application, while the EJB server provides automatic support for middleware services such as transactions, security, or database connectivity. Due to the inherited principles of application portability, rapid development, and easy deployment, Enterprise JavaBeans™ provide the perfect environment for designing a software infrastructure that allows reusable components to easily migrate between EJB servers. The proposed design takes advantage of those principles with the goal to support dynamic transfer, installation, and deployment of beans among a set of servers, with none or little pre-configuration required, and transparently to the end users.

The proposed migration infrastructure is comprised of the following components:

- A series of interconnected Enterprise JavaBeans™ servers running on different physical network servers.
- A set of different Enterprise JavaBeans™ deployed on one or more servers. These beans represent the set of independent applications to be distributed among the EJB servers.
- A “Monitor Bean,” the core of the migration infrastructure, which is responsible for the migration process, implemented as a session EJB, and deployed in each EJB server.
  We will to refer servers running monitor beans as “available” servers.
- A simple “performance” service that decides when applications need to be moved, migrated or removed from servers, and that communicates to one or more monitor beans.
- Access to a centralized directory service.
- A set of simple client applications used to test the business logic provided by the beans.

Software Requirements on each server:

- Java Runtime Environment version 1.3.
- jBoss version 2.0: an Open Source, standards-compliant, Enterprise JavaBeans™ application server.
- The Monitor EJB deployed on each jBoss server.

Other requirement and assumptions:

- At any point in time, all “available” servers run an instance of the monitor bean. This is the only bean that needs to be manually pre-installed in the server.
- Initially, at least one server must be deployed with all the beans that the system would be managing.
- Dynamic EJB deployment and loading, also referred to as “hot deployment,” is a service provided by the EJB server.
- EJB servers implement and support the Java™ Message Service, a reliable, flexible service for the asynchronous exchange of messages.

Note: Hot deployment and JMS support are services provided by jBoss 2.0 server.

System Architecture

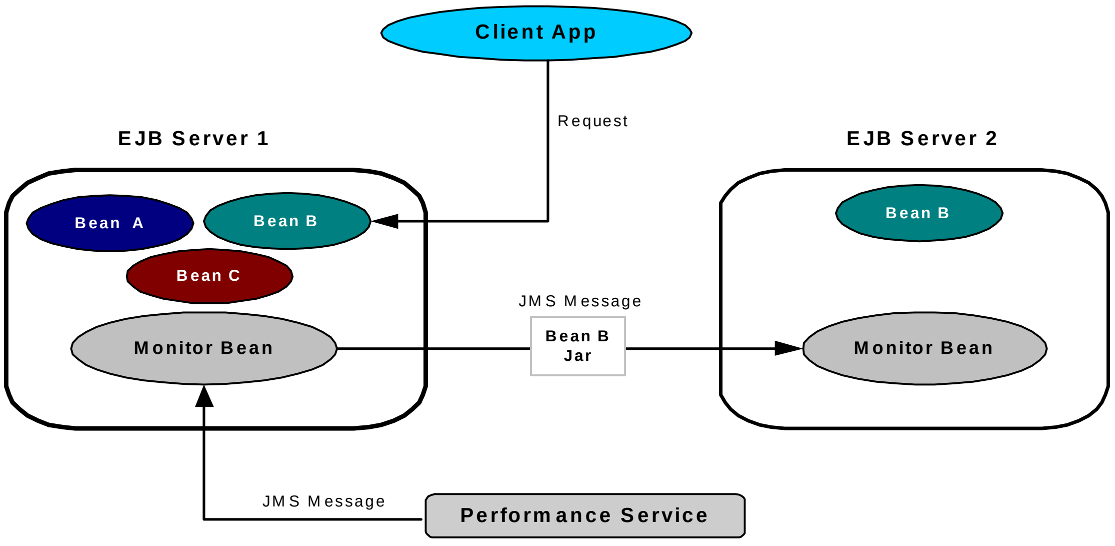
Overview of the Distribution Process

Let’s assume that initially only one server on our network contains all the beans, or applications, that need to be supported. Those beans have been deployed on the server and might be currently running and responding to client applications. While this is happening, a separate “performance” service, such as a load monitor, continually intercepts client requests, gathering data about the beans running on the server. As those beans start receiving and handling requests, the performance service periodically polls and analyzes the data that has been previously generated. The result of this analysis might indicate the need for additional instances of a particular bean is needed to increase its overall throughput, or that movement of one or more beans to a different server is required due to an overloaded server.

When one of these performance mismatches is detected, the performance service sends a JMS point-to-point message to the monitor bean running on that particular server, which reacts by replicating or migrating the affected beans to the appropriate available server. It does so following several steps. It first locates the bean’s deployable jar file that contains its source code and deployment information. The monitor then uses the EJB server utilities to pack the bean’s source code, deployment information, and state into a serializable byte array. When needed, the monitor is also responsible for undeploying, stopping, and saving the bean’s state.

When this process is complete, it simply sends one or more JMS messages to several monitor objects running on other servers. This message contains the associated byte array. The receiver bean then dynamically unpacks the byte array, and again, using the EJB server management utilities, recreates the bean's deployment jar file, deploys it, loads the necessary classes, and if needed, starts the bean, preserving its original state. EJB servers become application-distribution enabled by simply installing the monitor bean jar file.

Infrastructure Architecture



As mentioned before, any application can be either replicated or migrated depending on the specific need. Replication is achieved by creating copies of a particular bean on different servers. Thus, any replication operation produces at least n + 1 copies of a bean, where n is the initial number of copies currently in the system. Replication can be used to increase the number of requests served per application, or to easily distribute a new application among a set of servers.

On the other hand, migrating a bean entails removing it from the original server, and installing it on a different one, but maintaining the same amount of copies. This kind of behavior is desired if, for instance, an application is accessed by clients located far from the server, and moving it to a closer server would greatly reduce response times.

It is important to note than in order to avoid overloading the network with unnecessary data transfers and JMS messages, applications should only be distributed when critical performance thresholds are detected, so defining and implementing the policies used by the performance evaluator.

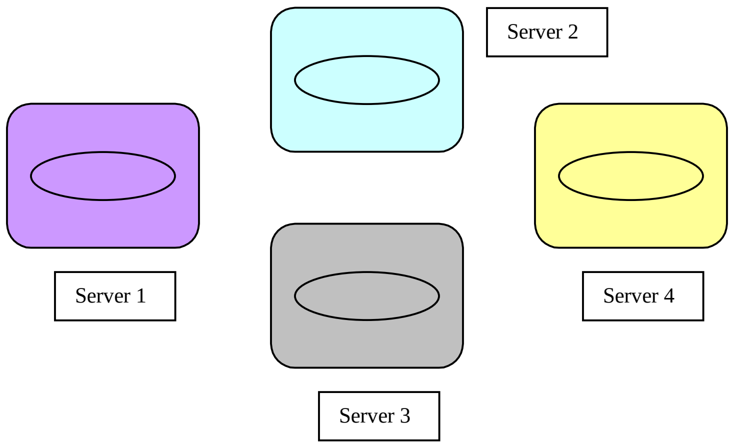
In the case of state-sensitive beans, meaning beans in which its state must be preserved, the monitor must somehow save the bean's current state before it is moved to a different server. At the other end, the receiving bean would use that state information to reactivate the bean at the same execution point. Fortunately, Enterprise JavaBeans follow an execution model in which the EJB server container is responsible for the state of each bean that is running. Thus, the tedious transaction management tasks involved in any bean life cycle, such as demarking transaction boundaries or rollback procedures, are completely handled by the server. At any point in this cycle, the container guarantees that if a bean is to be destroyed or simply returned to a pool of active beans, any request currently executing is completed and finished before the bean moves to a different state. Any bean that requires its execution state to be maintained across servers, will achieve this by implementing the activation/passivation application programming interface part of the Enterprise JavaBeans specification. Java Object Serialization, the encoding of objects, and the objects reachable from them, into a stream of bytes, is used for lightweight persistence of this state.

It is also worth mentioning that monitors, clients and performance service use a centralized directory service, where information such as number of servers on the system, their location, or the applications that each servers currently supports, can be shared among all the participants. This data is accessed through the Java Naming and Directory Interface, also referred as JNDI, which provides a unified interface for accessing any directory service, regardless of its location or implementation. For simplicity, we choose the network file system as the underlying JDNI service provider.

Monitor Bean

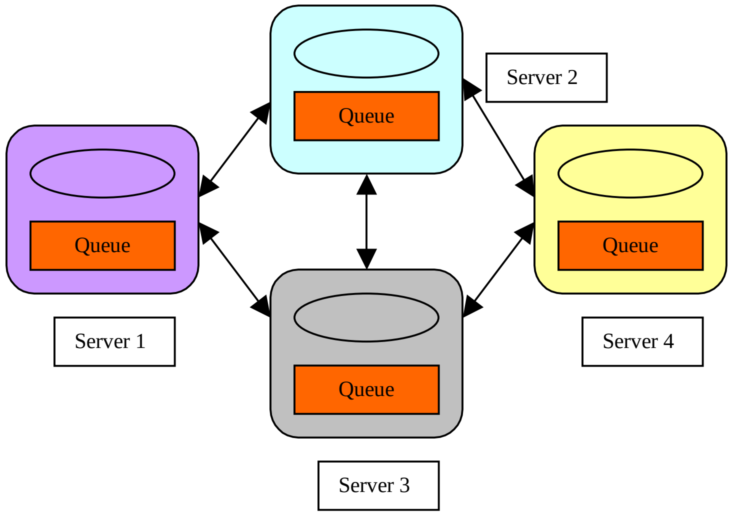
A Monitor bean represents the core of the distribution infrastructure. The jar file containing the remote and home interfaces, the bean class, and any other supporting classes used by the bean, is the only file required to be pre-installed. Once a server has been deployed with, it is then ready to start accepting and distributing beans.

The bean is designed as Stateful session bean. These objects are able to maintain state between method calls. They also have a unique identity that is assigned by the container at create time. To avoid unnecessary monitor instances, only one monitor per server is allowed at any point in time. The identifier is then used to locate remote instances running on others servers.



As mentioned previously, monitor beans communicate with each other and with the performance service through point-to-point Java messages that follow the JMS implementation. Point-to-point systems rely on queues of messages for their communication model. They are point-to-point in that clients send messages to a specific queue, while receivers pull or listen to the queue for new messages. Queues are managed and created by the EJB server, so little or no handling is needed by senders and receivers.

Each server creates a message queue at start time. Whenever a monitor wants to communicate to a remote instance, it sends the correct type of message to that bean's server queue. Every time a new message arrives at the queue, the server notifies any message listener registered for that queue. A listener is any object that wishes to receive messages from a particular queue. For this reason, the monitor also implements the JMS message listener interface, and they register themselves with the message queue at boot time. A monitor bean acts both as sender and receiver of messages. It is a sender when migrating or replicating a bean to a different server, while acting as a receiver at the other end pulling and handling those same messages. It also receives messages from the performance service that indicate which operation must be performed at that particular moment.



Class design

The bean class design includes the following interfaces, objects, and methods:

The Monitor remote interface includes:

- onMessage(javax.jms.Message message): Message listener method that gets called any time new messages arrive at the queue.
- register(): Registers the bean as a queue listener.

The Monitor Home interface includes:

- create(String identifier): Creates a new monitor bean.

Once any of these methods are called, the EJB server delegates its execution to a MonitorBean instance.

The MonitorBean class includes:

Stateful Session Bean methods:

- ejbCreate(String identifier): Creates and initializes the bean.
- ejbActivate(): Activates the bean, and initializes its state, when needed.
- ejbPassivate(): Passivates the bean, saving its state if needed.

JMS Methods:

- onMessage(javax.jms.Message message): Implementation of the same method defined on the remote interface. This procedure parses the message received, identifies its type, and then calls the appropriate message handler for it.

Monitor methods:

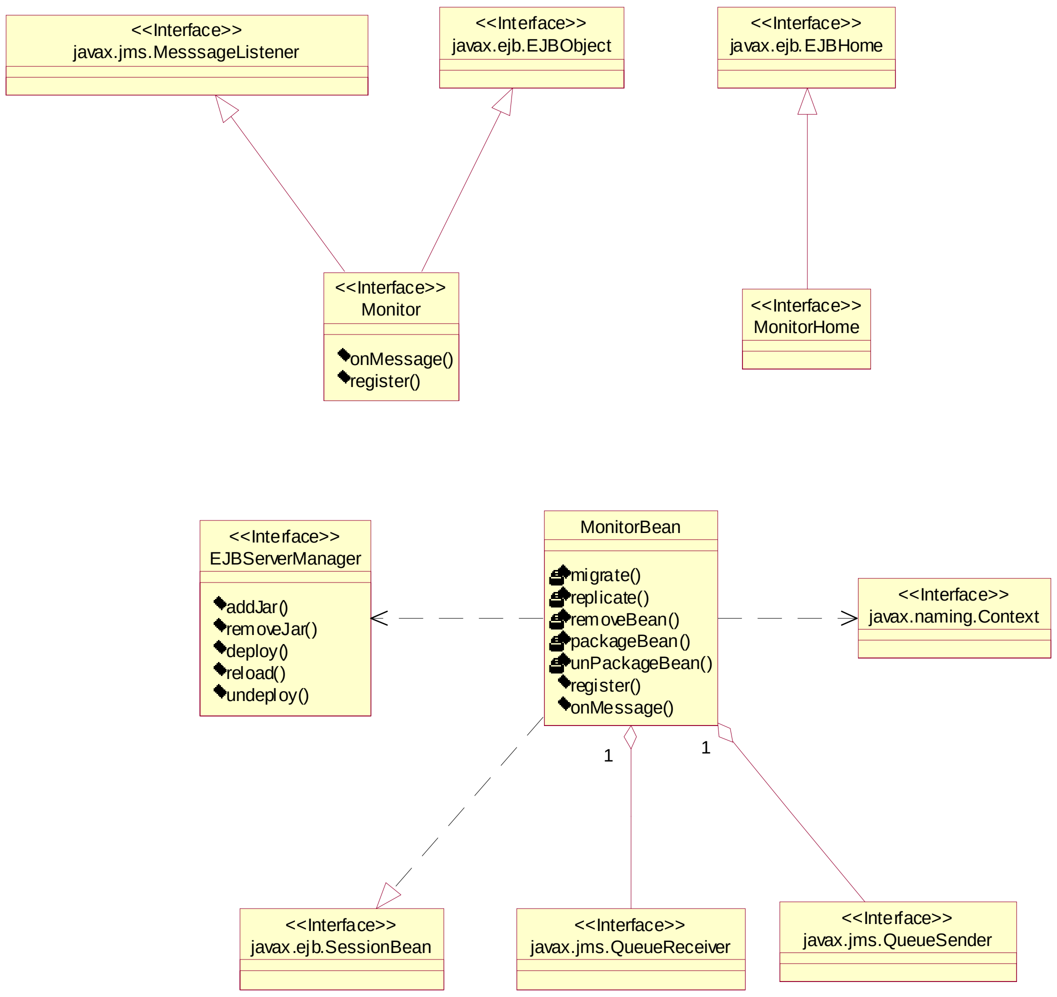
- migrate(ObjectMessage msg): Moves the specified bean to a different server. It does so by sending the correct message to the right queue on remote server. Removes the bean to me migrated from this server.
- replicate(ObjectMessage msg): Copies the specified bean to a different server. It does so by sending the correct message to the right queue on remote server.
- remove(ObjectMessage msg): Removes the specified bean from this server.
- installs(): Installs, deploys, and restarts the specified bean.
- removeBean(): Removes a bean from the server.
- packageBean(): Converts the specified bean jar file into a byte stream.
- unpackageBean(): Extracts the specified bean jar from the received byte stream.
- sendMessage(): Sends a JMS message to the specified monitor.

The remaining interfaces are support services that provide the following functionality:

- EJBServerManager: Provides server utilities that allow monitor beans to dynamically install and deploy jar files, as well as reloading bean classes.

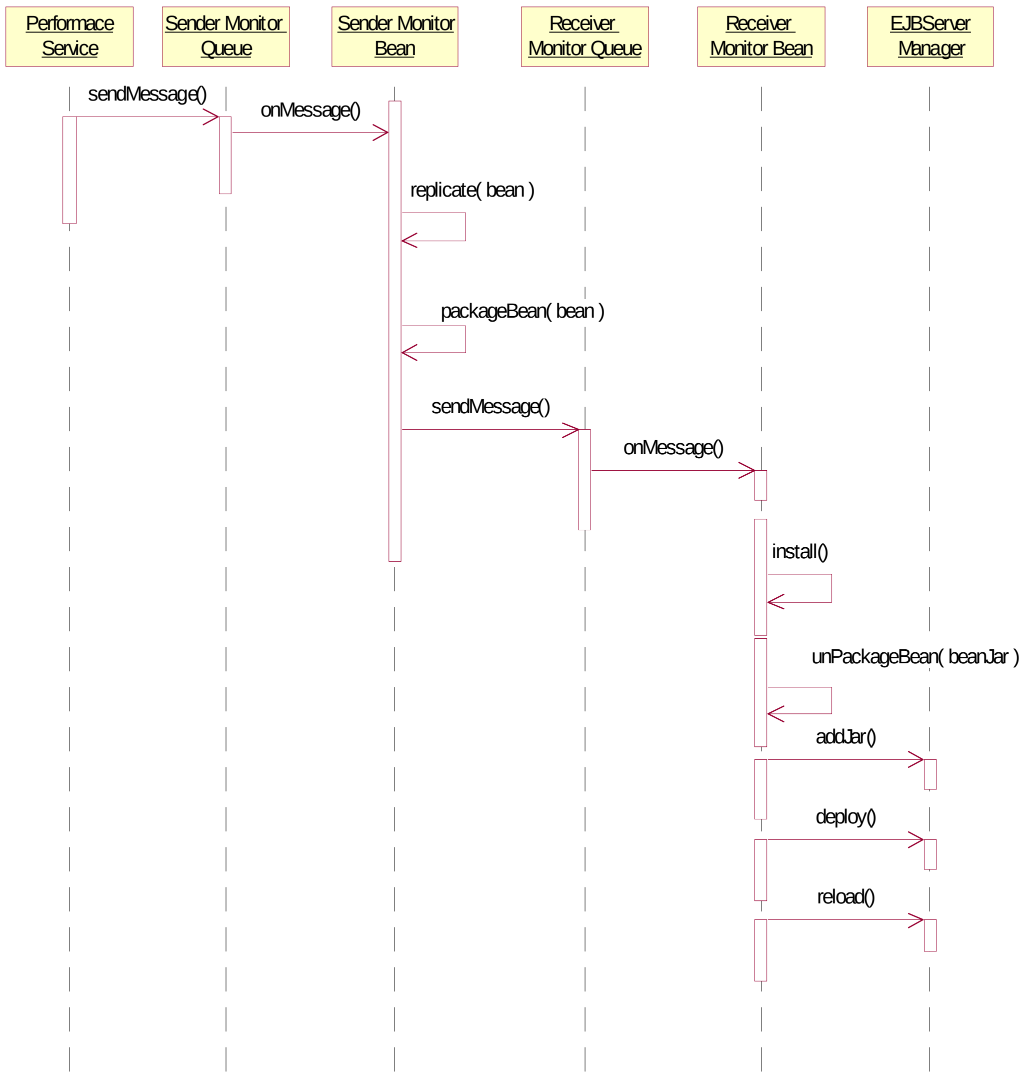
The class diagram bellows depicts the relationships among the most important objects and classes involved in the monitor bean.

Class Diagram (Booch Notation)

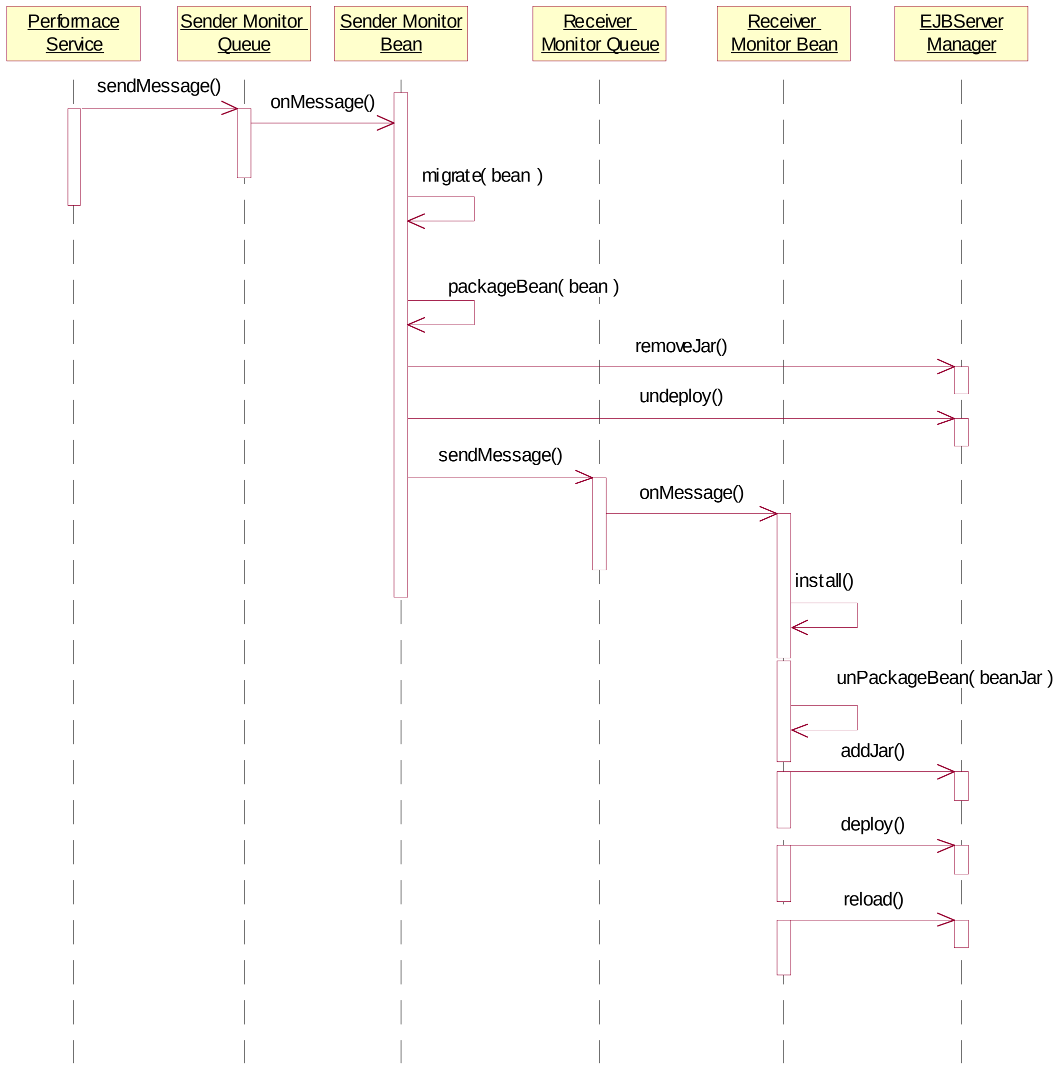


Use Case Scenarios

Replication Use Case



Migration Use Case



JMS Messages

At the core of the communication model sits the Java messaging service. Monitors use different type of messages to indicate other monitors whose operation needs to be performed at any given time. The performance service uses these messages to trigger monitor operations.

The following messages are supported in the system:

- *Install message:* Inter-bean object message that informs the receiving monitor bean that a particular bean needs to be locally installed. This message contains the name of the bean to be installed, and the byte array containing the bean's code, deployment information, and state information.

- *Replicate message:* Message used to indicate a monitor that a particular bean needs to be replicated to one or more servers. This message contains the name of the bean to be replicated, and the list of servers where it needs to be replicated. Replicate messages trigger one or more install messages.

- *Migrate message:* Message used to indicate to a monitor that a particular bean needs to be migrated to a different server. This message contains the name of the bean to be replicated and the name of the destination server. Migrate messages trigger one install message.

- *Remove message:* Message used to indicate to a monitor that a particular bean needs to be removed from the local server. This message contains the name of the bean to be removed.

Hot Deployment

Enterprise JavaBean servers use the deployment functionality to install and uninstall local beans.

The deployment process is typically divided in two stages:

- The Server's deployer tool first generates the additional classes and interfaces that enable the container to manage the enterprise beans at runtime. These classes are container-specific.
- The Server's deployer tool performs the actual installation of the enterprise beans and the additional classes and interfaces into the EJB Container.

Only when beans have been deployed and loaded, are ready to receive client requests.

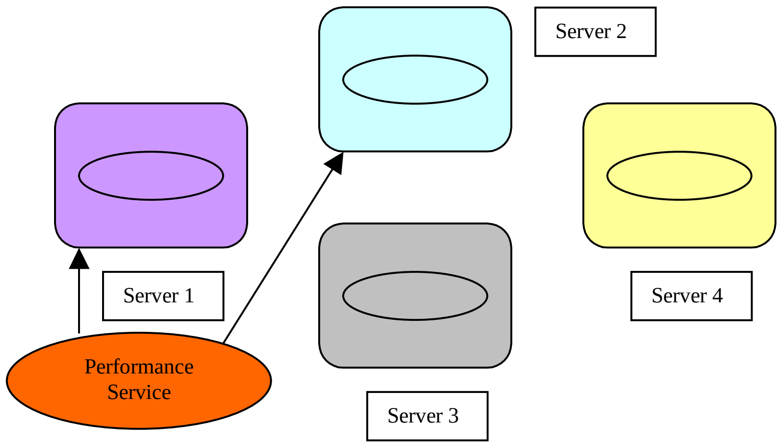
Currently, most of the EJB servers available on the market only support deployment of beans when the EJB server is stopped, since they need to be restarted in order to load the new classes. Hot deployment, or dynamic deployment and loading of beans on running EJB servers, is not part of the EJB 1.1 specification, and therefore, only a handful servers out there implement it.

Hot deployment is a key requirement, which is indispensable to support the dynamic distribution of beans proposed in this paper. The choice of jBoss as the underlying EJB server in this projects relies on its well-defined, stable, and easy-to-use hot deployment mechanism. Beans are hot deployed by simply dropping their jar files on a designated folder. jBoss constantly checks the folder for new files or updated versions, deploying and loading beans as needed.

Performance Service

Although monitors are responsible for the ultimate distribution of beans, they do not define when and where beans are moved. A secondary and totally independent process, named as Performance Service, analyzes and dictates the movement of beans among servers. By moving this decision process functionality out of the monitor logic, multiple distribution schemes can be used concurrently, optimizing the flow of beans, and reducing unnecessary data transfers.

In this particular implementation, this service is a simple command line application that sends replicate, migrate, and remove messages to any monitor in the system.



Since this type of service relies on the correct sequence of messages to control the system, replacing it with a more sophisticated version only requires knowledge of the message types and the actions they trigger. Thus, any user of the architecture might substitute it with their own version, tailored to support certain needs.

In fact, extending this service is one of the most interesting points for future development of the architecture.

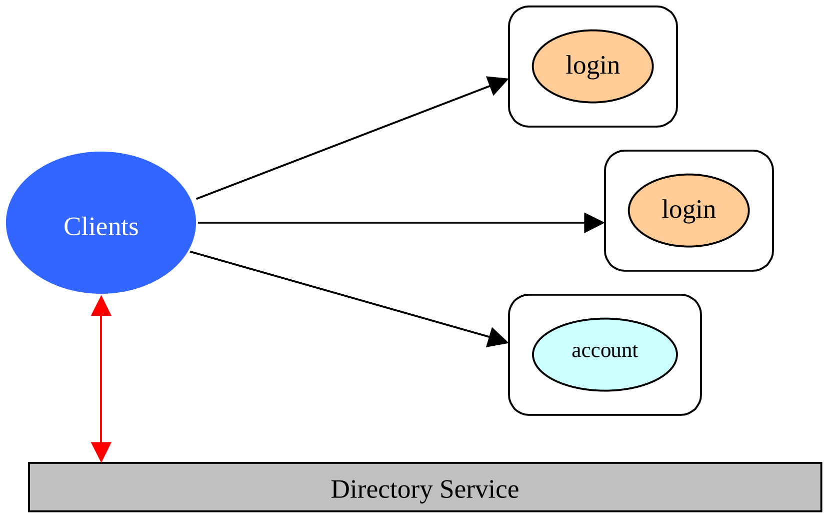
Example Applications

Financial Website

To demonstrate one of the possible applications of the infrastructure here exposed, WE created simple set of beans that mimic the behavior of common tasks used on any of the many financial websites available today. Thus, let’s assume that this web-based application is divided into a serious of tasks, such as user login and registration, account management, stock quotes, etc, and that every one of them is implemented as an independent session or entity bean. Part of the example includes a client application that simulates a user accessing some of the services provided by our financial.com business. A centralized directory is used by clients to obtain the list of available severs and beans applications deployed on them.

After initializing one or more servers with the monitoring software, and only one with the financial application, the client application starts making requests to the login bean, the account bean, and so on. While those requests are being processed, the performance service is started and we begin to send replicate, migrate, and remove messages to the monitor bean.

For instance, we might discover that the login bean on this server is handling too many requests, and we decide that second copy is needed on a server that was just added to the system. A replicate (login, new server) message is sent to the monitor, and the bean is copied to the new server, installed, and loaded. As the login bean is replicated from one server to the other, we are able to see that the client application now sends login requests to both servers. The same scenario can be described for a migration operation, as we discover that moving the account bean to a server closer to the client would reduce account balance checking, and lower the amount of applications running on the server. Again, the performance service sends a migrate (account, server) message to the monitor, which will immediately stop the account bean, save its state, remove it from the server, and send it to the monitor on the new server, which deploys and loads the bean. The entire process takes place transparently for the client. These clients use the centralized directory service to iterate through the list of servers in a round-robin fashion, discovering if the application they want to utilize has become available in that particular server.



Centralized Software Management System

The next example shows how the same infrastructure, with absolutely no modification, could be used by any Information Technology department to efficiently manage and administer software across an entire local area network. By installing the monitor bean in all the computers of the network, any system administrator, from a single location, could add, move and remove software from server to server.

If for instance, a new version of the client mail application needs to be installed on every computer, the administrator would simply install it on one server, and then would broadcast replicate messages to all the computers with the new mail version. The monitors running on those computers would then update the software by redeploying the new mail bean jar file.

Related Work

TUI Heterogeneous Migration System

Developed by Peter Smith and Norman C. Hutchinson, from Department of Computer Science at University of British Columbia, Tui is a general purpose migration package that allows processes to be migrated between machines of different architectures, also referred as Heterogeneous Process Migration.
As mentioned in the report, Process Migration has the ability to move a currently executing process between different processors, which are connected only by a network (that is, not using locally shared memory). The operating system of the originating machine must package the entire state of the process so that the destination machine may continue its execution. The process should not normally be concerned by any changes in its environment, other than by obtaining better performance.

Reference: http://www.cs.ubc.ca/spider/psmith/tui.html

The Process Introspection Project

Developed by Adam J Ferrari at the Computer Science Department of Virginia University, the Process Introspection Project provides a cross-platform solution for reliably state preserve running processes. As defined in the abstract, this project is a design and implementation effort, the main goal of which is to construct a general purpose, flexible, efficient checkpoint/restart mechanism appropriate for use in high performance heterogeneous distributed systems. This checkpoint/restart mechanism has the primary constraint that it must be platform independent; that is, checkpoints produced on one architecture or operating system platform must be restartable on a different architecture or operating system platform. Process Introspection is the ability of a process to examine and describe its own internal state in a logical, platform independent format. In some senses, all processes that employ a custom programmed checkpoint/restart implementation utilize the concept of Process Introspection.

Reference: http://www.cs.virginia.edu/~ajf2j/introspect/

BEA Clustering

Application distribution extends some of the concepts developed over the years in the areas of load balancing and server clustering. The term “clustering” can be defined as the cooperation of two or more replicated servers to ensure fast, continuous service to users. Clustering for Web applications means delivering scalability (via automated load balancing across replicas) and high availability (via automated failover across replicas).

One of the most sophisticated clustering solutions available today is provided by BEA Systems. Their flag J2EE application server, BEA WebLogic Server, can be arranged into a BEA Cluster is a group of WebLogic servers that coordinate their actions in order to provide scalable, highly-available services in a transparent manner. Both Web clustering, which ensures that the Web pages that your customer is requesting are available and delivered in near real time, and bean component clustering, which ensures that Enterprise JavaBean (EJB) business services and objects are both efficient and available, are supported. All Business logic is clustered by replicating the component (EJB) that provides the service across several different servers.

Reference: **http://www.bea.com/products/weblogic/server/clustering.pdf**

Future Work

Synchronization

Depending on the level of complexity, there might be some cases in which pure Java Object serialization does not provide all the requirements needed to successfully maintain an application synchronized across several distributing operations, and therefore, failing to preserve the application’s internal state intact.
Further development and integration with more advanced techniques and algorithms, such as the checkpoint/restart mechanism part of the process introspection introduced before, would resolve some of these issues, converting the dynamic bean distribution model into a reliable application synchronization practice.

Tombstones

When replicating, migrating, and removing applications -- in our case, beans-- we need to be careful not to remove applications when there are no more copes of its type running on other servers in the system. Unless we provide a mechanism to prevent this kind of scenario, a certain application might be mistakenly dropped from all available servers, leaving no more instances to return client requests.

Further work might be developed on this area, making sure that there is always a “tombstone” copy for each type of bean distributed. These copies could never be deleted through the monitors, and only by manually removing it from the server.

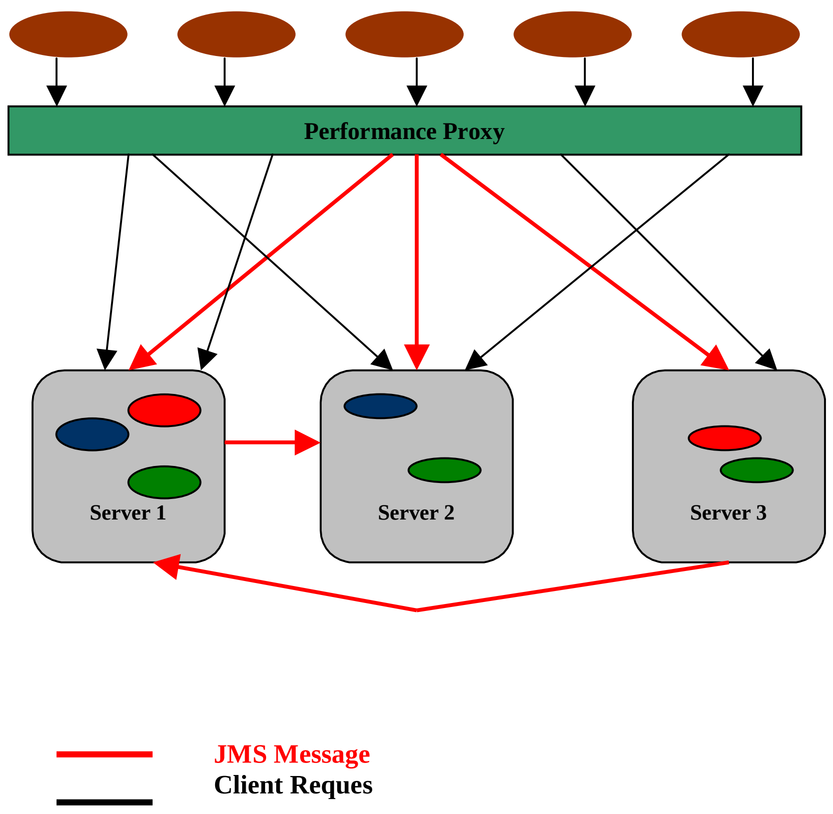
Directory Services

A directory service provides a centralized repository for commonly used data and resources used by all the parties involved in the distribution infrastructure. Although a simple file system was used oin this particular implementation, a more robust, scalable and distributed systems might be utilized. This might be accomplished by replacing the underlying provider to a CORBA COS naming service, an RMI registry, or a Lightweight Directory Access Protocol server, all of which support multiple concurrent accesses. Since all directory related calls are accessed through the Java Naming Directory Interface, no code changes are required if the provider is replaced.

Performance Proxy

While describing the performance service on the system architecture section, we briefly mentioned that more complex and advanced implementations of this service could easily be developed as part of one of the future enhancements. Indeed, it provides a perfect environment to apply well-known concepts such as load balancing, or failover recovery systems, into a distributed architecture. The performance service described then was a simple command based JMS sender that instructed monitors when and how to react.

One of the most interesting and obvious variants of the performance service is a client performance proxy. This proxy, or interceptor, would act as a service that continuously runs on the background. It acts as a smart middleware between clients and the application beans running on the EJB servers. The proxy would permanently monitor all client requests, and based on some policies previously defined, it would detect which applications and servers are exposed to heavy traffic. Using the JMS messages described in this document, it would redistribute the applications in the system in order to obtain a well-balanced architecture, maximizing throughput and reducing server overhead. Client request can then be redirected using a variety of load balancing and failover algorithms (random, round robin, server-load based).



Conclusion

The ideas and software design presented on this report are only intended to be the base groundwork from which more advanced complex systems might be built upon.

The combination of some of the advantages of the dynamic application distribution, such as application caching, unified software management, or dynamic software replication, with some of the latest development in the areas of web and content caching, cache management algorithms, or load balancing and replication strategies, presents a strong infrastructure capable of enabling and supporting high intense network based distributed applications, such as internet based e-commerce sites, business to business exchanges, or corporate intranets.

Appendix

Copies of this document and the software infrastructure can be obtained upon request by e-mail to: eduvos@bu.edu

Related links of interest:

- TUI Heterogeneous Migration System:
  http://www.cs.ubc.ca/spider/psmith/tui.html

- The Process Introspection project:
  http://www.cs.virginia.edu/~ajf2j/introspect/

- BEA Clustering:
  http://www.bea.com/products/weblogic/server/clustering.pdf

- Sixth International Workshop on Web Caching and Content Distribution:
  http://cs-pub.bu.edu/pub/wcw01/