JBoss architecture evaluation

January 14, 2009

Contents

1 Introduction

This document is the result of the pattern-based recovery and evaluation assignment of group B. For this assignment we have chosen to recover and evaluate the architecture of the JBoss application server version 5.0.

The goal of this document is to give an overview of the most important patterns that form together the core structure of the JBoss application server. Further more we want to evaluate the found patterns using the Pattern Based Architectural Review method [1] [2].

The patterns are found using various methods. First of all we looked into the documentation that can be found on various parts of the JBoss website. We tried to substract patterns from written text as well as from the images. We also have looked into the source code to see if we can find the described patterns. We tried also to reverse engineer by generating uml diagrams from the code. The structure of the code directories however, was that complex and large that the tools we used where not able to produce useful results. We also have some expierence with building applications on top of JBoss and other application servers.

The document is structered in such a way to reflect the steps of PBAR. Chapter 2 and 3 describes the context, the stakeholders of the system and the concerns of the stakeholders which together lead to the most important Quality attributes that should drive the development of the system. The chapters 4 and 5 give an overview of the patterns we have discovered and how those patterns are related to eachother. For each of the patterns we describe the problem they solve, what the impact on the architecture is and what variant (if applicable) is used. In chapter 6 we give an evaluation of how well the quality attributes identified are addressed. Finally in chapter 7 we give some recomendation of where additional assesment of the architecture might be needed.

One last remark to make here is that for the quality attributes we use the defenitions as given in the quality model described in the ISO-9126 standard.

[1]We will refer to this method as PBAR in the rest of the document.

2 System context

Java Bean Open Source Software (JBoss) is an open source component based framework for deploying webapplications and services in a service oriented architecture. In april 2006 it was bought by Red Hat who is still the current owner. Enteprises that want to deploy their distributed application using JBoss can either become costumer of Red Hat to get support or use their own JBoss experts.

JBoss provides middleware services for data and code integrity, centralized configuration, security, performance, total cost of ownership and transactions. It is an implementation of the Java 2 Enterprise Edition (J2EE) standards using Java SE and therefore platform indepedent. JBoss connects the custom application build on top of it with one or more databases. Figure 2 [2] gives an overview of how an J2EE platform connects with other componenents.

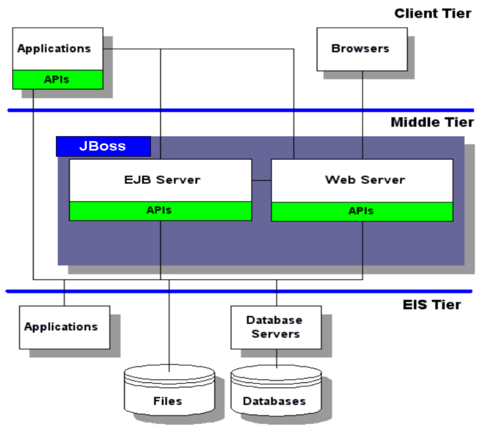

Figure 1: System context diagram

Using an application server enables developers to create distributed applications without having to implement basic features for these kind of application (e.g. logging, transaction support, caching, etc.) over and over again.

[2]http://www.service-architecture.com/application-servers/articles/application_server_definition.html

3 Stakeholders

From the system context we derive the most prominent stakeholders of the system. Table 1 lists these stake holders with their most significant concerns. The stakeholders are listed in order of importance.

stakeholder	**concerns**	**Priority**
Red Hat Management	- The product should integrate well with Red Hat Linux	high
	- The product should contribute to sales increase of Red Hat products	medium
	- The product should be very competitive on the application server market	high
	- It should be relatively simple to add new features to the product	high
Red Hat Customers	- The product should provide a stable and reliable environment for the business applications.	high
	- The product should scale to the needs of the costumer	high
	- The product should provide means for high performing distributed applications	high
	- The product should perform well.	high
	- The product should integrate well with existing business applications	medium
Application Developers Using JBoss Platform	- The product should be well documented.	medium
	- The product should provide means to easily integrate clustering, security, transaction support, caching, monitoring and persistence into a custom application.	high
Red Hat JBoss Developers	- It should be easy to test if contribution do not break functionality or performance of the system	medium
Community JBoss Developers	- It should be easy to get known with the internals of the JBoss product.	low

Table 1: Stakeholders and their concerns

From the stake holders we derive the following Quality Attributes, most important first:

1. Reliability - The application server should be an high perfoming framework which works as expected under specified circumstances.
2. Adaptability - The application server should easily integrate in various environments (e.g. It should easily adapt to various communication protocols and hardware platforms).
3. Availability - The application server should be able to avoid failures.

4. Changeability - The application server should be easy changable to adept to the specific needs for the application which is run on top of it.

4 Architecture - Logical and process view

This section describes the overall architecture of the JBoss server including the most prominent patterns.

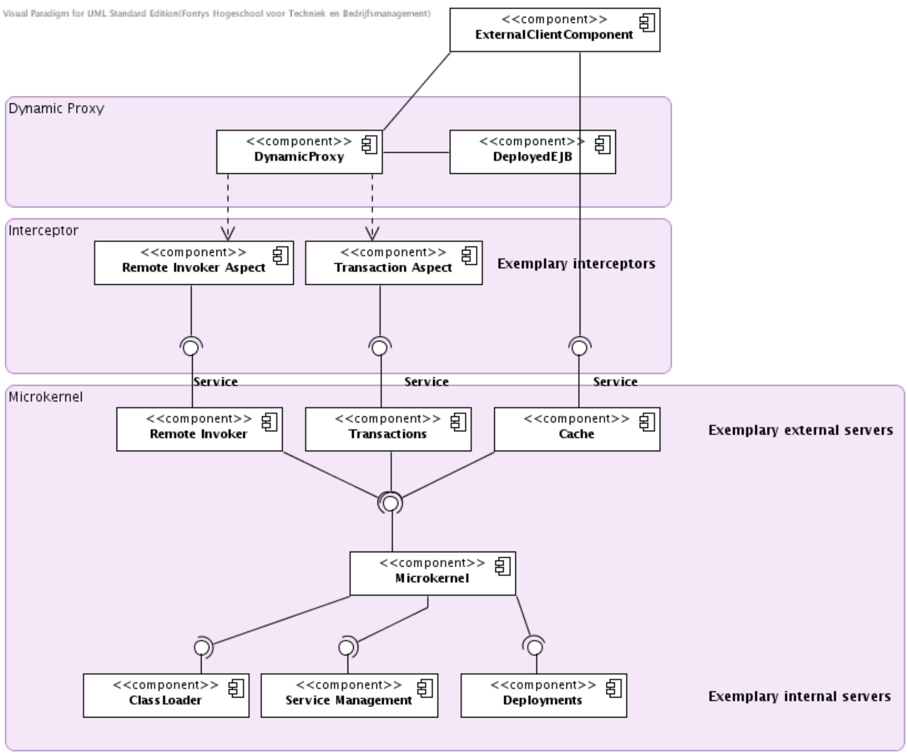

Figure 2: Core architecture

Although figure 2 optically reminds of a Relaxed Layered system, the Microkernel pattern describes the architecture better. The JBoss server may be used as part of business applications that use the layers pattern, though.

Two of the major concerns of the JBoss stakeholders are related to changeability and adaptability. The Microkernel pattern was used to satisfy these concerns. The functional core of the architecture is the so called Microcontainer that can be seen as a microkernel representing the central component of the system. It implements central services like classloading, deployment and service management. On top of the microcontainer, services are deployed as external servers. These services include remote invoking, transaction- and cache management. Besides the provided services, any other service can be deployed on top of the microkernel by providing a predefined service interface. The microkernel implementation used here is JMX (Java Management Extension), services are implemented using so called Managed Beans (MBeans).

Some of the services that provide functionality to satisfy cross cutting concerns of component base enterprise applications are wrapped in so called interceptors, providing means for Aspect Oriented Programming (AOP). The pattern, that can be seen here is the Message Interceptor pattern. Interceptors can be combined in a chain to provide business component behavior around plain components. The variant chosen in the JBoss server is the Chain-of-Responsibility variant. Each interceptor calls the next following interceptor. An intercepting-context is passed among the invocations. Please see the Interceptor pattern in the pattern section for further details and diagrams. Interceptors can be configured to be used by dynamic proxies, invokers and containers, that provide a runtime environment for business components. The combination of the Dynamic Proxy pattern and the interceptors is prominent in the JBoss remoting package. Clients always use business components through so called dynamic proxies. It is *dynamic*, because the behavior of the proxy can be changed at runtime. Method calls on the proxy object are delegated to a proxy handler. The proxy handler itself is configured to invoke a chain of interceptors, before finally the remote-invocation-interceptor is used as a Broker to forward requests to the Application Server itself, that also invokes a chain of interceptors before actually invoking the target method on a business component. Further explenations and detailled diagrams can be found in the dynamic proxy pattern explenation in the pattern section.

Subsuming the combination of patterns that is explained in figure 2 is *Microkernel*, *Message Interceptor*, *Dynamic Proxy* and implicitly *Broker*. Other patterns, that were identified, but that are less prominent in the architecture are explained in the following sections.

5 Pattern Documentation

This section describes how the possible patterns (and their variants) used to design the architecture of the JBoss system. Each pattern is represented by a table whose structure is inspired by [3]. The tables provide details on the reasons for choosing the corresponding patterns with respect to the system requirements.

Since the amount of JBoss documentation is enormous, we started with the overviews presented in figures 3 [3] and 4 [4].

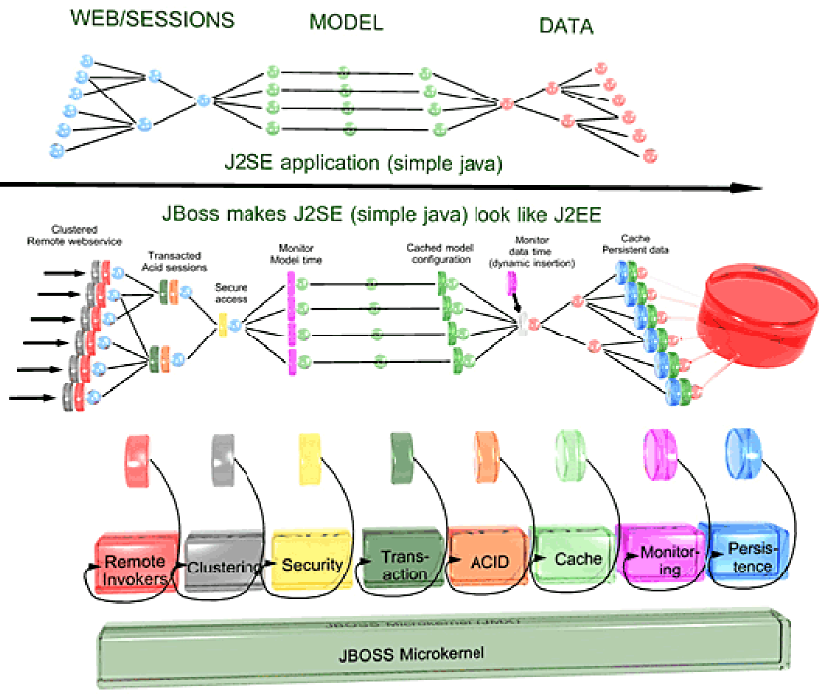

Figure 3: JBOSS Deployment Architecture

[3] http://www.jboss.com/products/jbossas/architecture
[4] http://www.jboss.org/projects/

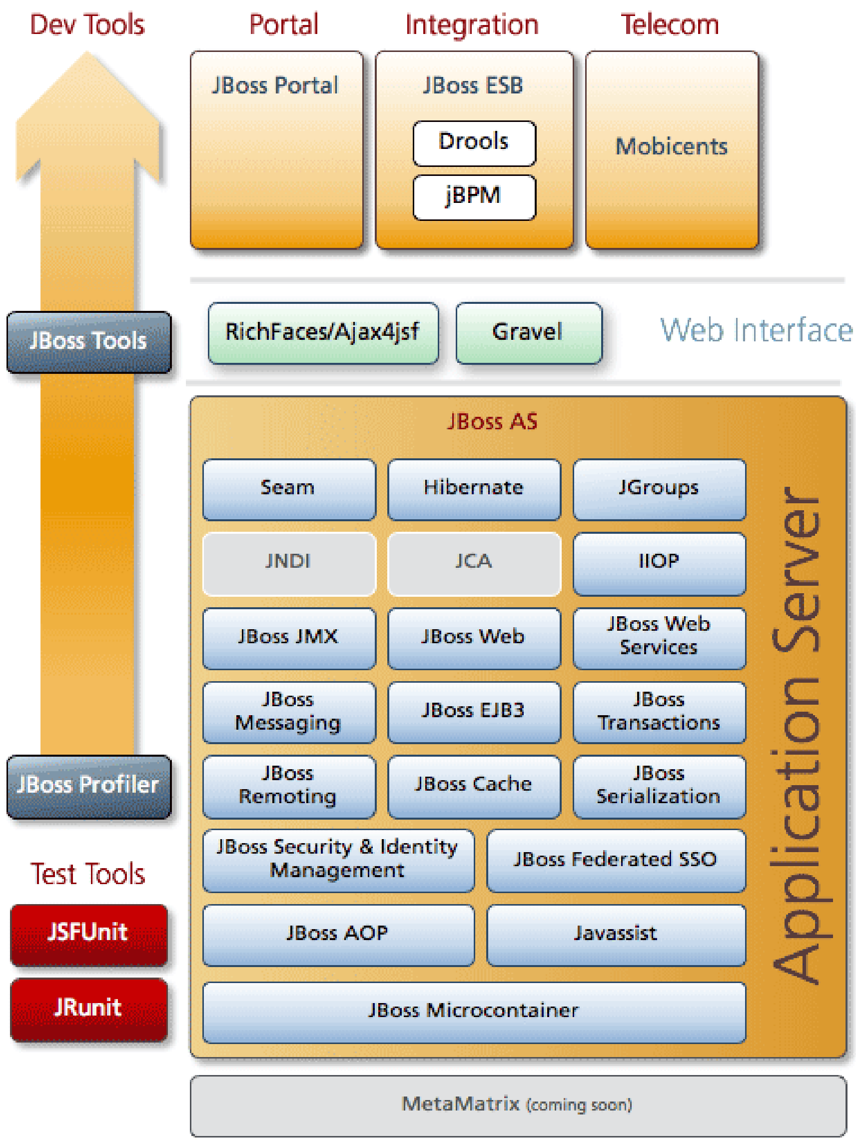

Figure 4: JBOSS Projects Overview

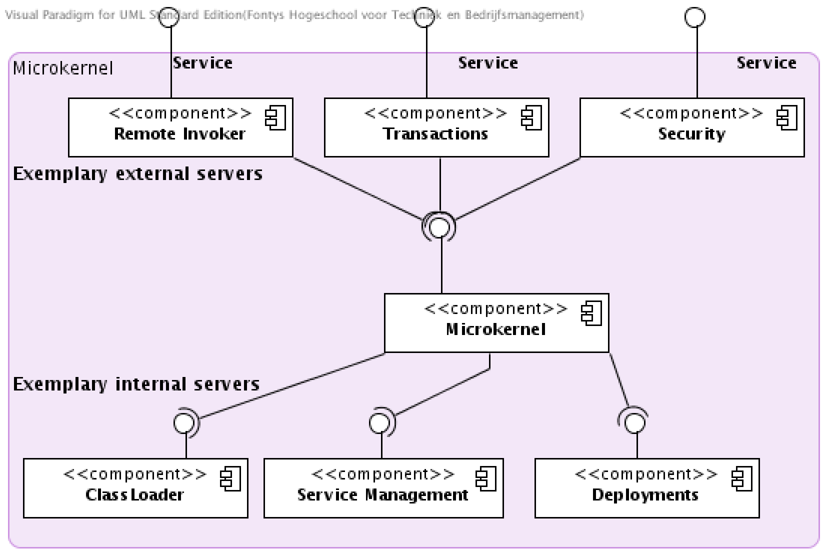

Figure 5: The JBoss microkernel

5.1 JBoss Core

5.1.1 Microkernel

Pattern Section	Comments
Name	Microkernel
Problem	Two basic forces were taken into consideration when designing the Jboss server. 1. The Application Server platform must cope with continuous software evolution. The services provided by the Jboss are among others specified in the Java Community Process and new versions emerge rapidly (2 years). 2. The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies. The Jboss server should be configurable for extremely lightweight application type setups like junit testcase or Mobile Applications, as well as for full feature application servers.
Category	Adaptable Systems
Context	Development of several applications that user similar programming interfaces that build on the same core functionality.
Variants	The variant chosen to satisfy the forces is documented in POSA1. There it is the base variant of the Microkernel pattern. In this variant Client and Server (Service Provider) communicate directly. Messages are not passed through the microkernel on every request.

Solution	The Jboss developers decided to encapsulate the fundamental services needed by a full featured JEE Application Server as described in the JEE specification in a so called Microcontainer. The internal servers (services) provided by the microcontainer are the following: 1. Class loading 2. Deployments 3. State management 4. Lifecycle and Dependency management 5. Configuration 6. Service management Other services provided by the application server are implemented as external servers. For every external server, a Service Proxy is created. The instance of the service proxy is bound to the client. Some well known external servers in the Jboss download versions are: 1. AOP 2. Security and Identity Management 3. Remoting 4. EJB3 5. Transactions 6. Web-Services This is just a small excerpt from existing external servers.
Rationale	The JBoss Application Server uses the microcontainer to integrate enterprise services in order to provide a standard Java EE environment. If additional services are neeed, then they can simply be deployed on top of the container to provide the needed functionality. Likewise any services that are not needed can be removed simply by changing the configuration. Since JBoss Microcontainer is very lightweight and deals with POJOs (Plain Old Java Object) it can also be used to deploy services into a Java ME runtime environment. This opens up new possibilities for mobile applications that can now take advantage of enterprise services without requiring a full JEE application server.

<table>
<tr><td>Consequences</td><td>External servers do not need to be ported to a new software environment. Only the microcontainer has to be ported, which improves adaptability.
The microcontainer based architecture is very flexible and extensible. Extending the system with additional capabilities only requires the addition or extension of servers.
The microcontainer based architecture is complex in design and implementation. It is a non-trivial task to analyze and predict the basic mechanisms that must be provided by the microcontainer. As a result it is likely, that services were forgotten and need to be implemented bypassing the microcontainer. The effort for refactoring might be higher than in a layer based system for example.</td></tr>
<tr><td>Related patterns</td><td>Layers, Interceptor</td></tr>
</table>

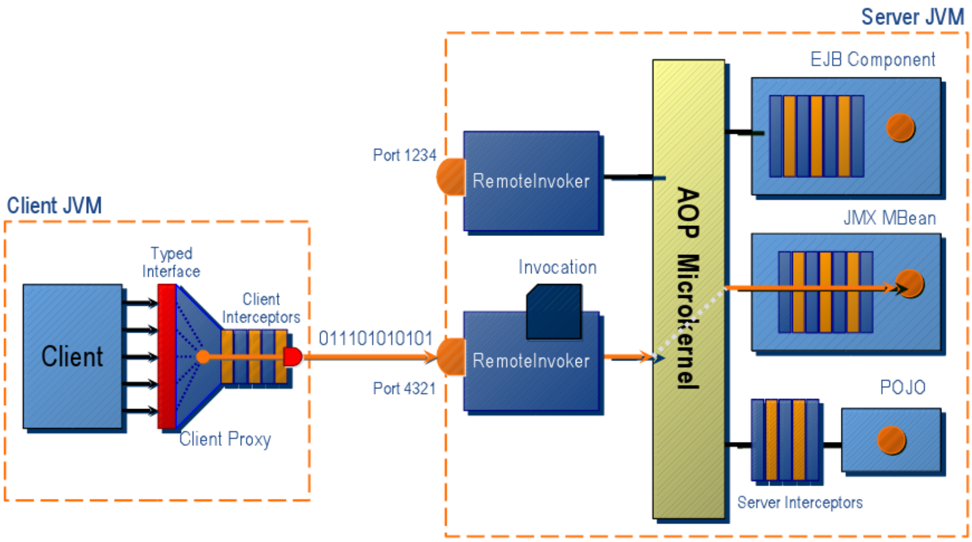

Figure 6: Interceptors and Invokers

5.1.2 Interceptor

Pattern Section	**Comments**
Name	Interceptor
Problem	The JBoss started as a server, providing a runtime environment for Enterprise Java Beans (EJB). The main challenge here was to wrap the specified EJB services like transaction, security, logging, profiling, caching and others, around a client's call of an EJB method. The developers needed to create a flexible and, for the user, easy-to-change implementation, as the EJB services are optional services the users must be able to choose among.
Category	Aspect Oriented Programming
Context	Separation of cross-cutting concerns in enterprise java applications.
Variants	The variant of the interceptor pattern used in the JBoss server, see figure 6, is the *Message Interceptor* pattern as documented in [4]. A chain of interceptors is applied in an indirection layer. Some interceptors can be configured to be invoked before the invocation of the target component, others may be configured to be invoked afterwards. The pattern is used to implement the concept of *container based deployment* as specified in the EJB core specification. A container can be seen as runtime environment for EJB components. In the JBoss server, containers are specified as a chain of interceptors that are applied before a method on a bean component is invoked.

Solution	The interceptors are invoked for the indirection layer invocation events *before* and *after* a method invocation. It is done using the dynamic proxy pattern. The handler of the dynamic proxy gets to know the chain of interceptors that need to be processed before and after a method on the target object, hidden by the proxy is invoked. The interceptors themselves are typically used to invoke services, registered as external servers in the microkernel architecture.
Rationale	The JBoss Application Server uses interceptors to wrap enterprise services around method invocations on deployed components. If additional services have to be invoked, then they can simply be plugged in as interceptors in the interceptor chain. Likewise any services that are not needed can be removed simply by changing the configuration.
Consequences	1. Services needed for deployed components can be configured very flexible and extensible. 2. Interceptors may introduce single points of failures. If one interceptor crashes the whole chain of interceptors is interrupted. 3. An architecture that makes excessive use of interceptors is complex in behaviour. The behaviour of the system is hard to predict, as interceptors are dynamic components invoked at runtime. 4. The concept of AOP, that is followed using interceptors is also currently not accurately supported by IDEs.
Related patterns	Microkernel, Dynamic Proxy

5.2 JBoss Enterprise

5.2.1 Dynamic Proxy

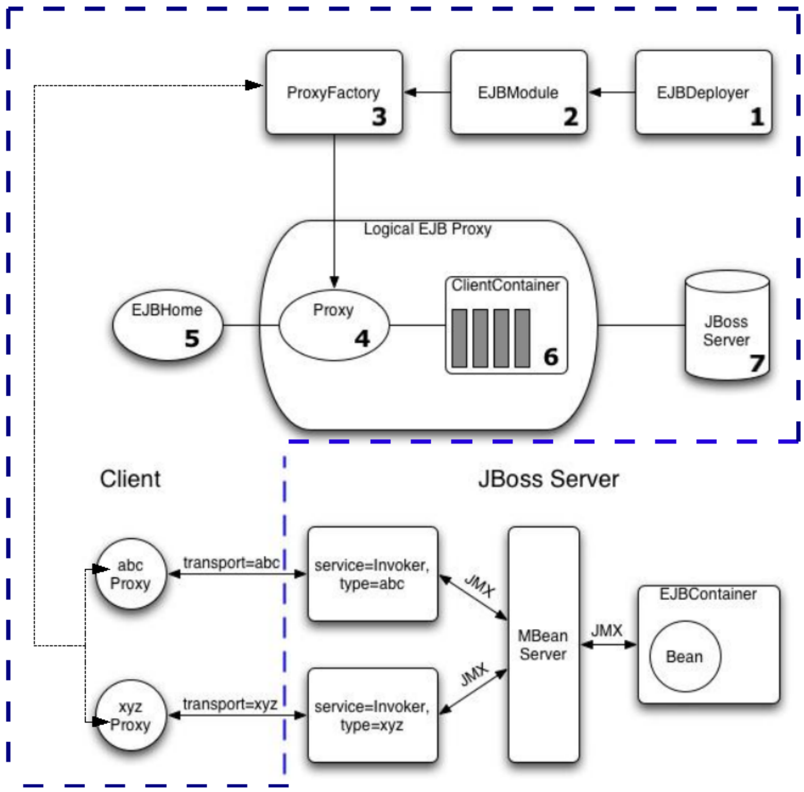

Figure 7: EJB Invocation using Dynamic Proxy

Pattern Section	**Comments**
Name	DYNAMIC PROXY

Problem	For any developer who builds an application on JBoss, the mechanism for invocation of a remote service should be similar to a local function call. This capabilty would allow developers to construct distributed systems with ease. The abilty of the JBoss environment to provide remote service transparency is an essential need.
Category	Distributed Communication
Context	Client applications should be able to make remote invocations in a transparent manner.
Variants	The stated problem can be solved using a variant of the DYNAMIC PROXY pattern. The presence of JAVA invocation handlers provides the functionality for the given variant (instead of the standard PROXY pattern).
Solution	The proxy element of the DYNAMIC PROXY pattern serves as a substitute for a target POJO on the remote server. The proxy is an object that can implement a list of specified interfaces at run time when it is created using java reflection. The first step is to construct a POJO which implements one or more interfaces that are to be exposed for remote method invocation. A transporter server is then wrapped around the POJO to expose it remotely. On the client side, in order to be able to call on the target POJO remotely, a client transporter is used. The client transporter takes in the locator to find the target pojo (same as one used when creating the transporter server) and the interface for the target POJO, on which the remote method invocations are to be made. The return from this create call is a dynamic proxy which can be cast to the same interface type supplied. This is a result of the target POJO being serialized and sent to the remote client across network. At that point, any method can be invoked on the returned object, which will then make the remote invocations using JBoss Remoting. This mechanism is also used by JBoss to provide EJB functionality. In the EJB specification, EJBHome implements the bean home interface and EJBObject implements bean remote interface. EJBObject interface presents a client's view of EJB and it is the responsibility of container provider to generate the EJBHome and EJBObject. In the design of JBoss EJB container, there is NO EJBHome and EJBObject object implementation at all. It is the proxy element that takes on the role of EJBHome and EJBObject.

Rationale	By simply providing the locator url of the remote service and the given interface a client application can obtain a proxy object, which can be called on directly. This provides location transparency implying ease in development.
Consequences	1. It is possible to create multiple target POJOs using the transporter server in clustered mode, implying scalability. 2. Clustering also allows for automatic, seamless failover of remote method invocations improving availability. 3. Since the application developer can call on proxy objects directly, usability improves. 4. Changes in the target POJO can done without any impact implying changeability. 5. Since JAVA reflection is used extensively in generating the proxy, reliability can decrease.
Example uses	An example of the proxy pattern is a reference counting pointer object. In situations where multiple copies of a complex object must exist, the proxy pattern can be adapted to incorporate the Flyweight Pattern in order to reduce the application's memory footprint. Typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.

¡¡¡¡¡¡¡ .mine ======= ¿¿¿¿¿¿¿ .r265

5.3 JBoss Remoting

The purpose of JBoss Remoting[5] is to provide a single API for most network based invocations and related service that uses pluggable transports and data marshallers. The JBossRemoting API provides the ability for making synchronous and asynchronous remote calls, push and pull callbacks, and automatic discovery of remoting servers. The intention is to allow for the use of different transports to fit different needs, yet still maintain the same API for making the remote invocations and only requiring configuration changes, not code changes.

5.3.1 Client - Server

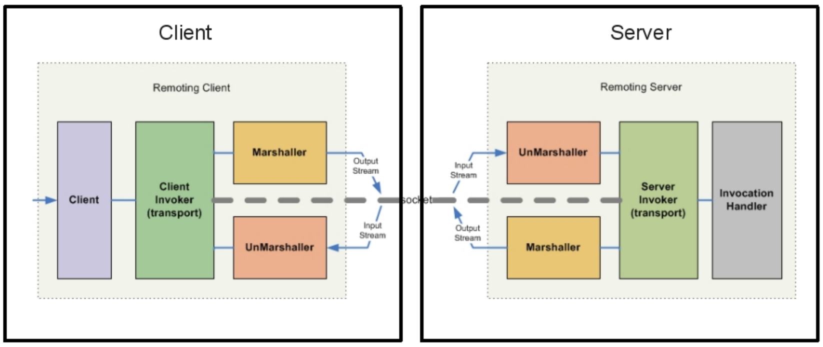

Figure 8: Client - Server Decoupling

Pattern Section	Comments
Name	CLIENT-SERVER
Problem	Since JBoss is in itself an application server, a system based on clients connecting to the server is an inherent feature of JBoss and is an essential need for all concerned stakeholders.
Category	Remote Invocation
Context	Clients can connect to application servers and perform remote invocations on services provided by the server.
Variants	The variant used here is the basic variant of Client-Server pattern as described in [1]

[5] http://www.jboss.org/jbossremoting/docs/guide/

Solution	The J2EE platform acts as a server capable of handling web, ejb and basic remoting requests. The Client Tier can be one or more applications or browsers. There can be additional sub-tiers on the server side including the Enterprise Information System (EIS) tier which links to existing applications, files, and databases. As seen in fig.8, the client application calls the remoting client api to make an invocation request to a service on a remote server. The request is directed to the appropriate invocation handler on the remote server, which handles the request and generates a response, which is sent back to the client.
Rationale	An application server framework needs a distinction between client applications and server processes, to benefit from the advantages of a distributed system. This allows distinction between application written on the client side and server invocations, implying better management of functionalities.
Consequences	1. Decoupling of the functionalities into client and server elements allows for better changeability of either component without affecting the other. 2. The single point of interaction also improves integrability of the client - server components. 3. Since the remote server can provide services for multiple clients, reusability improves. 4. Due to heavy dependence on the remote server, reliability and availability can be adversly affected.
Example uses	JBoss Remoting can be used to implement content management systems where multiple clients perform remote invocation calls to obtain specific content from a data mangement application hosted on a central server.

5.3.2 Invoking and Marshalling

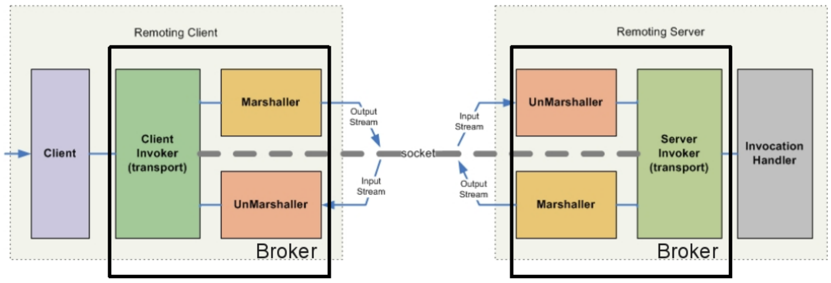

Figure 9: Invoking/Marshalling using the Broker Pattern

Pattern Section	**Comments**
Name	BROKER
Problem	One important feature of any application server is its ability to handle different methods of client-server communication across networks. For the benefit of all users of JBoss it is necessary to hide the communication details, so that they can focus on developing applications that efficiently tackle their domain specific problems. This aspect has been given considerable importance in the development of JBoss.
Category	Remote Invocation
Context	Server identification should be performed in a manner which allows for remoting servers to be easily identified and called upon. Communication details regarding transport protocols, data (un)marshalling and serialization should be hidden from the user.
Variants	The stated problem can be solved using a variant of the BROKER [1] pattern. In this variant the requestor (Client) hides the communication details by calling on the the appropriate invoker which in sequence calls on the marshaller / unmarshaller to handle the request / response.

Solution	Server identification can be done (via an Invoker-Locator object) using a simple string with a URL based format (e.g., socket://myhost:5400). This is all that is required to either create a remoting server or to make a call on a remoting server. As seen in fig.9, the broker element, on both client and server, consists of an invoker which handles transport details and a component that performs marshalling/unmarshalling of data. When a user calls on the Client to make an invocation, it will pass this invocation request to the appropriate client invoker, based on the transport specified by the locator url. The client invoker will then use the marshaller to convert the invocation request object to the proper data format to send over the network. On the server side, an unmarshaller will receive this data from the network and convert it back into a standard invocation request object and send it on to the server invoker. The server invoker will then pass this invocation request on to the users implementation of the invocation handler. The response from the invocation handler will pass back through the server invoker and on to the marshaller, which will then convert the invocation response object to the proper data format and send back to the client. The unmarshaller on the client will convert the invocation response from wire data format into standard invocation response object, which will be passed back up through the client invoker and Client to the original caller
Rationale	The client can easily identify the server as well as specify the transport protocol by providing a simple locator url. The same locator url is also used to create and initialize a remote server. The broker element takes the information embedded within the locator url and constructs the underlying remoting components needed to build the full stack required for either making or receiving remote invocations. By restricting the transport layer specifications to the locator url, the broker element ensures that user applications on the client and invocation handling applications on the server are not impacted by the underlying details relating to communication.

Consequences	1. Solution to hide communication details improves integrability between the clients and servers as the broker element provides a single point of contact between the two. 2. Changeability of communication functionality improves as the broker element can be adapted without affecting the client / server application layers. 3. Since the broker element provides a simple interface to the client applicaton usability improves. 4. Adaptability, reliability and availability of the system can be affected, since the broker element presents itself as a single point of failure.
Example uses	If a user decides to change the transport protocol from sockets to http, the only change required will be the locator url (e.g. from 'socket://myhost:5400' to 'http://myhost:80').

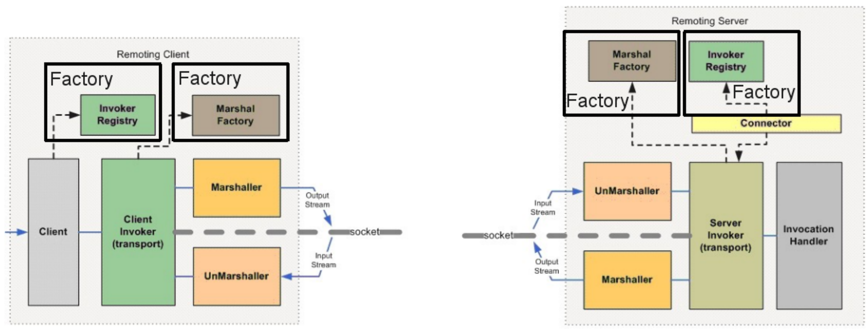

Figure 10: Invoking/Marshalling using the Factory Pattern

Pattern Section	Comments
Name	FACTORY
Problem	The feature that make an application server interesting to use, is its ability to handle different flavours of transport protocols, data formats for wire transfer and serialization implementations. Consequently, it is required that JBoss consider them as essential.
Category	Remote Invocation
Context	Different protocols that transport the same remoting API should be pluggable. Provided protocols should include Socket, RMI, HTTP(S), Multiplex, Servlet and BiSocket. Different implementations of (un)marshalling and serialization for data streams should be easily integrated.
Variants	The stated problem is solved using a variant of the FACTORY pattern. The transportation level is split into client and server factories, each of which provide separate functionalities, whereas the (un)marshalling level consists of a single factory.
Solution	The transport implementations within JBoss remoting, called invokers, are responsible for handling the wire protocol to be used by remoting clients and servers. As seen in fig.10 the JBoss remoting loads client and server invoker implementations (within the InvokerRegistry) using factories (TransportClientFactory / TransportServerFactory). The invokers are generated based on the locator url provided by the client API which in turn call the MarshallFactory, where the marshaller/unmarshaller is generated based on the data type information in the locator url.

Rationale	The factory design allows the implementation of different invoking and marshalling methods.
Consequences	1. The FACTORY pattern improves changeability with respect to the invokers and (un)marshallers, as the transport details can be changed / adapted without affecting the other components. 2. As the factory provides a standard interface, integrability becomes easier as all implementation details are hidden behind the interface.
Example uses	In the case of remote invocation on the server 'myhost' on port 5400 using sockets and marshalling data type as serializable, the locator url would be 'socket://myhost:5400/?datatype=serializable'. To perform the remote call, the client begins by passing the locator url to the *InvokerRegistry* which returns a *ServerInvoker* instance, to handle the transport of data. The *ServerInvoker*, in turn, obtains a *SerializableMarshaller* instance from the *MarshalFactory* to convert the data into the required wire format for the remote invocation call.

5.3.3 Discovery

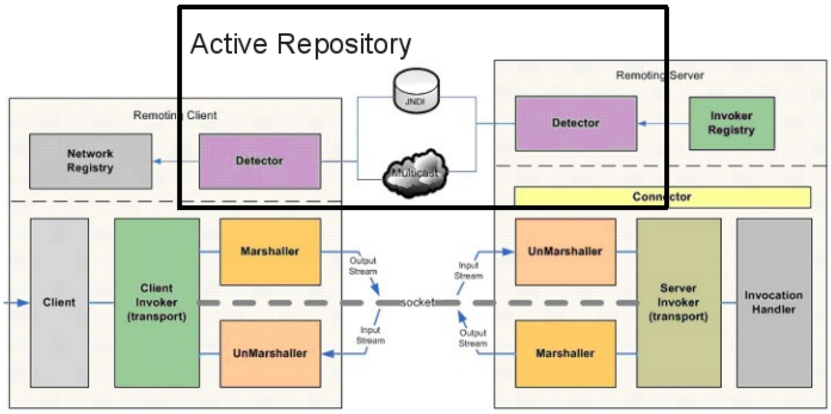

Figure 11: Discovery using the Active Repository Pattern

Pattern Section	Comments
Name	ACTIVE REPOSITORY (EXPLICIT / IMPLICIT INVOCATION)
Problem	To ensure availability of services deployed on remote servers, the JBoss architecture should ideally include a mechanism which allows clients to discover when particular services are not available and seamlessly switch over to the ones that are.
Category	Remote Invocation
Context	Client applications should be able to automatically detect remoting servers as they come on and off line.
Variants	The stated problem can be solved using a variant of the ACTIVE REPOSITORY [1] pattern. JBoss remoting provides notification functionalities via the EXPLICIT INVOCATION [1] (multicast broadcast) pattern or IMPLICIT INVOCATION [1] (JNDI server binding), in which case the PUBLISH-SUBSCRIBE [1] pattern is used.

Solution	To add automatic detection, a remoting Detector will need to be added on both the client and the server side as well as a NetworkRegistry to the client side. With respect to fig.11, when a Detector on the server side is created and started, it will periodically pull from the InvokerRegistry all the server invokers that it has created. The detector will then use the information to publish a detection message containing the locator and subsystems supported by each server invoker. The publishing of this detection message will be either via a multicast broadcast or a binding into a JNDI server. On the client side, the Detector will either receive the multicast broadcast message or poll the JNDI server for detection messages. If the Detector determines a detection message is for a remoting server that just came online it will register it in the NetworkRegistry. The NetworkRegistry houses the detection information for all the discovered remoting servers. The NetworkRegistry will also emit a notification upon any change to this registry of remoting servers. The change to the NetworkRegistry can also be for when a Detector has discovered that a remoting server is no longer available and removes it from the registry.
Rationale	In this variant the Detector object on the server side acts as the central repository to all the client Detector objects that subscribe to it. Both the multicast broadcast and/or the JNDI server binding done by the server detector provides, to all connected clients, timely information regarding available services. The client application is then made aware of the changes and an appropriate reaction can be implemented.
Consequences	1. Since the information concerning services is consumed by all subscribed clients, reusability is improved. 2. Loose coupling of server and client detectors implies better changeability. 3. Clients running on different platforms can subscribe to the server detector, implying enhanced intergrability. 4. Since it is possible for a single client to connect to many services and vice versa, adaptability, reliability and availability of the system is improved.

Example uses	

6 Quality Attribute Evaluation

The earlier section provided details on the patterns extracted from the JBoss architecture and their related consequences with respect to the quality attributes. This section gives an overview of how well the quality attributes are addressed by the patterns identified, along with corresponding recommendations. This corresponds to step five of the PBAR method

	Microkernel	Interceptor	Client-Server	Broker	Factory	Active Repository	Dynamic Proxy	Plugin
Reliability			-	-		-	-	
Adaptability	+			+	+	+		
Availability		-	-	-		+	+	
Changeability	+	+	+	+	+	+	+	
Integrability			+	+	+	+		
Reusability			+			+		
Usability	-	-		+			+	

Figure 12: JBoss Patterns Vs Quality Attributes Matrix

The matrix in fig.12 gives a brief summary of the impact (positive/negative) that the patterns have on the quality attributes. The quality attributes given in the matrix correspond to the stakeholder concerns and are presented in the matrix in order of ascending importance. These attributes include,

6.1 Reliability

Most of the patterns extracted from the architecture affecting reliability introduce singular entities on the communication path. The entities include (but are not restricted to) the remote server in the CLIENT-SERVER pattern, the broker element in the BROKER PATTERN, the repository component in the ACTIVE REPOSITORY pattern and the proxy object in the DYNAMIC PROXY pattern. These entities become bottlenecks in the case of overload and can therefore affect the performance ofthe system. Data throughput and timing issues may occur in that case. Another major reason for decreased reliability is the number of potential indirections that are present in the current system.
The reliability of the system seems to be negatively impacted by the chosen combination of patterns. One of the main reasons that reliability fails is due to the fact that client invocations target a single service on the remote server. If that target service is unavailable, the remote invocation fails. However, clustering of remote services is already provided by the JBoss architecture. So, clustering combined with discovery of services (provided by the ACTIVE REPOSITORY pattern) and network transparency (provided by the BROKER pattern), makes the system more fault tolerant and hence much more reliable.

6.2 Adaptability

The current JBoss archictecture is highly adaptable in a number of system components. These include communication protocols and service discovery mechanisms. This is mainly achieved by the BROKER, FACTORY and ACTIVE REPOSITORY patterns.
It appears that the combination of chosen patterns gives a highly adaptable system. Since the system is highly configurable, it can have a major impact on other quality attributes. For example, smart configuration of interceptors, communication and discovery can lead to a more reliable system, since it becomes more easy to do failure recovery.

6.3 Availability

Interceptors, remote servers and broker elements are essentially single points of failure. This implies that the patterns that bring these elements into the architecture have an adverse effect on the availability of the system. On the other hand, discovery of services by the ACTIVE REPOSITORY and clustering of services made possible by the DYNAMIC PROXY helps in reducing downtime. Availability of the system is partially addressed by the chosen patterns. One of the areas of improvement could be the implementation of interceptors.

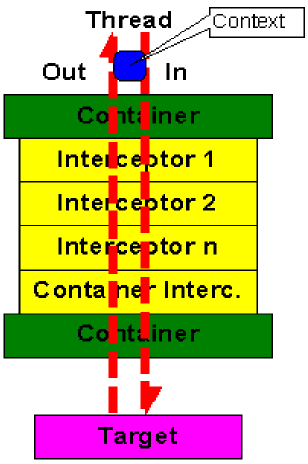

Figure 13: Interceptor Stack

As seen in fig.13, in the JBoss archictecture, interceptors are stateless components arranged in a stack, wherein every call proceeds through the stack from first to last. The stack is embedded in a specific container (e.g. MicroContainer or EJB Container). Each interceptor is responsible for invoking the next interceptor in the stack, as seen in the following code snippet,

```
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;
```

```
public class HelloAOPInterceptor implements Interceptor {

   public Object invoke(Invocation invocation) throws Throwable {
         // PERFORM the INBOUND Tasks and when done go to the next
         // If the call has to be ended throw the appropriate exception

         System.out.print("Hello, ");

         // Call next Interceptor in the stack
         // PERFORM the OUTBOUND Tasks and when done return method
         // If the call shouldn't be complete throw and exception

         return invocation.invokeNext();
    }
}
```

This method of implementation allows for the entire stack to crash when the *invokeNext* method is not reached. This failure could occur in either of the inbound or the outbound tasks. This problem can be solved by implementing an iterator in the container itself. The iterator is responsible for invoking the interceptors in the right order and managing any failures if they occur. In this way, it is assured that the entire stack is executed or the failures (if any) are handled properly.

6.4 Changeability

One of the major advantages of the JBoss architecture is that it can be easily modified. A large number of components can be changed both at compile time as well as at run time. In fact all the patterns identified seem to have a positive impact on changeability.
It seems that the quality attribute of changeability is addressed very well by the chosen patterns. The decision to make the JBoss architecture highly changeable does negatively affect reliability (to some extent). However it remains a good design decision which should not be changed without good reason, because it fufills an essential need of an application server, which is to be highly configurable.

6.5 Others

Most of the patterns support the integration of individual components and their reusability. However, due to the complexity of the JBoss architecture, usability from the developers point of view has a negative impact with respect to the core components. However the architecture also provides elements that hide unnecessary details implying ease in application development.

7 Recommendations

This section corresponds to the sixth step of the PBAR method. We provide recommendations of where additional, more detailed assessment should be made.

7.1 Negative Impact

Out of all the quality attributes addressed, reliability seems to be the worst affected. As already mentioned this problem can be partially solved by specific configurations of the system. However, more research needs to be done to mine applied patterns which have not been included our findings. The results of this research could eventually bring out patterns which have a positive impact on reliability and help in the analysis to resolve the problem of reliability.
Availability is also negatively impacted by some of the patterns, but we have managed to find patterns which partially solve the problem. Furthur research into the variants of the patterns could yield better results.

7.2 Conflicting Impacts

The CLIENT-SERVER and BROKER patterns both have a negative impact on reliability and availability, but they are retained in the architecture because of the benefits they bring to other quality attributes. Variants of these patterns (e.g RELIABLE BROKER) could be used to reduce the negative impacts.

7.3 QAs not addressed

Efficiency is an important requirement in any application server, but this requirement has not been addresed by our research. The reason for this being that more time and resources are required to test specific parts of the system with respect to time behaviour and resource utilisation in a real time setup. Compliance to efficiency standards is another area which has not been investigated by our research.

7.4 Subsystems with no patterns

As JBoss is a huge architecture framework, it is difficult to document every pattern implemented by it. This document presents the components and related patterns that represent the core architecture. Other subsystems not addressed in this document include JBoss RichFaces/Ajax4jsf, JBoss Cache, Hibernate etc. [6]

[6] http://www.jboss.org/docs/

References

[1] P. Avgeriou and U. Zdun. Architectural patterns revisited - a pattern language. In *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, July 2005.

[2] Neil B. Harrison and Paris Avgeriou. Assessing quality requirements through pattern-based architecture reviews. *?*, ?(?):?, ?

[3] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *IEEE Softw.*, 24(4):38–45, 2007.

[4] Uwe Zdun. Pattern language for the design of aspect languages and aspect composition frameworks. In *IEE Proceedings Software*, 2004.