TUHH
Technische Universität Hamburg-Harburg

Monitoring and Managing Business Processes and Resources in J2EE Platform

Master Thesis

Submitted by:
Aravind Kumar Alagia Nambi
Master of Science in Information and Media Technologies
aravind.alagia@tu-harburg.de
Matriculation Number:
23307

Supervised by:

Prof. Dr. J. W. Schmidt
STS - TUHH

Prof. Dr. Karl-Heinz Zimmermann
Technische Informatik VI - TUHH

M.Sc. Miguel GARCIA
STS - TUHH

Hamburg, Germany
2004-10-30

Abstract

As J2EE adoption grows, the business use and importance of J2EE application deployments are on the rise. Increasingly, mission-critical applications are being built and deployed on J2EE infrastructures. This trend is driving the demand for better administration, monitoring, and management of J2EE applications to foresee any breakdowns that may occur. The software industry has widely recognized the importance of J2EE application management to enterprise IT, and has been seeking ways to address this growing need. However, awareness of enterprise J2EE deployment issues and how to best address this need are only now emerging. My thesis concentrates on finding a better solution to monitor and manage the business process.

Acknowledgements

My special thanks go to Prof. Dr. Joachim W. Schmidt, research assistant MSc. Miguel Garcia at Software Systems Department of the Technical University of Hamburg-Harburg and Dipl-Inf. Krisztiàn Szitàs for providing the opportunity to work on this research topic, for their support, encouragement and guidance throughout the thesis work.

Content

1. INTRODUCTION 6

2 BUSINESS PROCESS MANAGEMENT 7

2.1 Managing J2EE Applications 8
2.2 Management through Event Processing 8
2.3 Business-Oriented Management 9
2.4 External Instrumentation 10
2.4.1 Analysis of JMX Studio 11
2.5 Internal Instrumentation 12
2.5.1 Features of Aspectwerkz 13
2.5.2 Analysis of Aspectwerkz 14
2.6 Best practices for Business Process management 15

3 BUSINESS PROCESS MANAGEMENT WITH JAVA MANAGEMENT EXTENSION (JMX) 16

3.1 Java Management Extension (JMX) 16
3.2 The Instrumentation Level 17
3.2.1 Standard MBean 18
3.2.2 Dynamic MBean 18
3.2.3 Model MBean 21
3.2.4 Open MBean 23
3.3 Agent level 23
3.3.1 MBean Server 23
3.4 Distributed services level 24
3.5 Notification Model 25
3.5.1 Notification Broadcaster 25
3.5.2 Notification 26
3.5.3 Notification Listener 27
3.5.4 Notification Filter 28
3.6 Architecture of the Monitor 28
3.6.1 Logging Service 29
3.6.2 StandardMBean interface 31
3.6.3 Notification Broadcaster 32
3.6.4 Notification Listener 32

4. BUSINESS PROCESS MANAGEMENT WITH JAVA MESSAGE SERVICE (JMS) 33

4.1 Java Message Service (JMS) 33
4.1.1 JMS API Architecture 34
4.2 Messaging Domains 35
4.2.1 Point-to-Point Messaging Domain 35
4.2.2 Publish/Subscribe Messaging Domain 36
4.3 Message consumption 36
4.4 JMS API model 37
4.4.1 Message selectors 40
4.5 Architecture of the Monitor 40
4.5.1 Logging service 41
4.5.2 JMS Topic 43
4.5.3 Filter 43
4.5.4 Monitor 44
4.6 LoggingService incorporated in Business Process 46

5 PERFORMANCE ANALYSIS 47

5.1 Filter options 47
5.1.1 Database 47
5.1.2 File system 47

5.2 JMX IMPLEMENTATION OF FILTER OPTION 48
5.2.1 Filter repository *48*
5.2.2 Persistence of the filter repository *49*
5.2.3 Manipulation of filter parameters through MBeans *50*
5.3 COMPARISON OF JMX AND JMS ARCHITECTURES 52
5.3.1 Performance *52*
5.3.2 Server Cluster *52*

6 SYSTEM-LEVEL MANAGEMENT 55

6.1 ARCHITECTURE OF JBOSS APPLICATION SERVER 55
6.2 J2EE SERVER MANAGEMENT WITH JSR 77 56
6.2.1 Differences between JMX and JSR 77 .... *56*
6.3 J2EE MANAGEMENT MODEL 57
6.3.1 Managed objects *57*
6.3.2 Events *58*
6.3.3 State Management *59*
6.3.4 Performance Monitoring *59*
6.4 J2EE MANAGEMENT EJB COMPONENT 59

7 MANAGEMENT OF TOMCAT 63

7.1 JMX AND TOMCAT 63
7.2 MODELER COMPONENT 63
7.3 CUSTOM SERVICE IN TOMCAT 64
7.4 MANAGING AND MONITORING TOMCAT 65
7.4.1 Java Specification Request 160 .... *66*
7.4.2 Connectors *66*
7.4.3 Connection Establishment *66*

8 CONCLUSION 69

REFERENCES 70

1. Introduction

The term "enterprise computing" is definitely one of today's busiest buzzwords. It is computing that uses technology as a strategic tool for modeling a business process. The computing could be mission-critical business computing or mundane applications. In today's world, most of the enterprise computing is based on Enterprise Java Bean (EJB) technology.

Any enterprise computing solution must focus on modeling business concepts without imposing restrictions on business processes. Even if one sticks to this principle and develop a business process, nothing is impregnable to the flaws that may arise at some point in time during the production mode. The flaw could be due to performance bottlenecks at the EJB level, flaw at the business logic level at overload state, etc. The business process can not afford to have a flaw at any time. This requires a monitoring tool that can check the health and performance of the business processes and can predict a potential breakdown. This can in turn alert the Corporate to make the necessary changes to keep their business process running.

An effective strategy, that encompasses the technologies Java Messaging Service (JMS) and Java Management Extension (JMX), is to be devised that monitors and manages the business process without affecting the performance of the business process. Considering the vast pool of Java 2 Enterprise Edition (J2EE) based application servers in the market, it becomes necessary to develop a generic set of libraries for the tool that are independent of the specifics of the J2EE based application servers.

2 Business Process Management

The primary driver for business process management is business productivity and efficiency goals. Enterprises deploying J2EE applications are often using these applications to drive their business processes. Since the goal of these J2EE applications is often to enable efficient business operations, managing these applications provides visibility into how the applications are impacting operations and shows ways to improve productivity and efficiency.

The first step in managing applications is monitoring availability and performance. In order to minimize downtime, and avoid disruption to business operations, there is a clear need to monitor and measure the availability of applications. In addition, the performance of the application as measured by key business methods or user transactions needs to be tracked. This helps ensure the application performs to the service levels needed, and enables pro-active and rapid response if the application is under-performing.

The business process could be Enterprise Java Beans (EJB), Servlet, etc. EJB poses big challenge because the overall health and performance of such applications depends on a variety of parameters associated with the various application components (or "Beans") and how they interact with each other. It becomes necessary to provide the granularity of reporting necessary to pinpoint potential or active performance bottlenecks at the Bean level. The performance of these applications is critical for the companies that deploy them, because they support critical eBusiness operations: including interactions and transactions with customers, suppliers and other business partners. Hence the management tool must not just give performance metrics, like the time taken for the execution of a set of statements, but also be able to test the veracity of the business data and also the behaviour of the logic at varying loads.

The business process management has effects both at the production phase and at the development phase. At the development phase, management tool helps in checking/debugging the logic. At the production phase, management tool exposes the business data and the logic flow at varying load. This helps in predicting the failure of the business process and also helps the Corporate come up with a solution before the business process breaks down.

The management of the business process is to be achieved in two levels

- At the business logic level (Business-level management)
- At the system resource level, which the business process accesses (System-level management)

Both of them complement each other. Any effect in one will affect the other. Poor business logic may use the system resource inefficiently and crash the whole system. At the business level management, the business specific data should be checked for its veracity. And at the system level management, the resources can be checked. The system resources are the system infrastructure and the resources of the J2EE platform.

2.1 Managing J2EE Applications

Enterprise adoption of J2EE has led to increasing deployment of applications built on J2EE in mission-critical business applications. It is useful to understand the different dimensions of management and the drivers for application management in these deployments. This will help provide a better picture of enterprise requirements.

In the early stages of any technology, when deployments are few and small-scale, there is not a lot of focus on ensuring that these applications are well managed. As the technology matures, however, an increasing emphasis is placed on going beyond application features and functionality. Enterprises then focus attention on ongoing operations, administration, and maintenance. As business use matures and scales, managing availability, downtime, upgrades, performance, sizing, security, integration, and other management aspects loom large in enterprise software deployments. As J2EE application deployments are increasingly serving mission-critical business functions, the importance of application management is rapidly growing.

The systems infrastructure upon which J2EE applications are deployed needs to be monitored and managed. This is to ensure that the CPU, memory, disk, network, and other resources needed by the applications are available and reliable. As the scale of the infrastructure grows, as well as the criticality of the applications to the business, it becomes more important to proactively manage these elements to ensure availability and sound performance. Not all of these aspects will need tight management in a given environment

In addition to the system infrastructure, the J2EE infrastructure — including thread and connection pools, database, Web server and other software elements — needs to be monitored and managed. Here, the deployment choices will dictate what needs to be managed closely. It is quite useful for managers to be able to track the usage of various resources in the J2EE infrastructure, and be confident it is operating in good condition. Administrators may need to be notified by the management system when specific resources (e.g., connection pools) are running at full capacity to enable prompt action.

Beyond that, the focus typically shifts to end-to-end application performance and identifying performance issues in any part of the application transaction chain. Visibility into different application transactions and the break-up of different elements of the transaction is useful in isolating performance bottlenecks. This requires close monitoring of the business process, in essence the business logic and the business-specific data.

2.2 Management through Event Processing

Complex Event Processing (CEP) is an emerging technology, invented by David Luckham Professor Emeritus of Electrical Engineering, Stanford University, for building and managing information systems. The goal of CEP is to enable the information contained in the events flowing through all of the layers of the enterprise IT infrastructure to be discovered, understood in terms of its impact on high level management goals and business processes, and acted upon in real time.

The enterprise is operating in a complex environment of events happening on a global scale. These are high-level business, logistics, and application-to-application events. They form the global event cloud in which the open enterprise is operating. The scale of the global event cloud that each enterprise must interact with is continually increasing. The need for instant insight into the operations of the electronic enterprise has become crucial. Insight into the Global Event Cloud is also essential simply to control the enterprise's business processes. Electronic business processes are no longer sequential flows of activities related to document processing. The new generation processes incorporate complex, parallel, asynchronous decision-making. These processes work at web speed with far less human involvement than before.

CEP is an event processing technology utilizing the concepts of

- Causal , timing and membership relations between events,
- Patterns of multiple events together with their relationships,
- Event-pattern triggered reactive rules,
- Event-pattern constraints,
- Event abstraction hierarchies.

CEP can be used to design business processes, simulate and test them, and monitor their production operations.

Processes are event-driven. They receive and react to events, and create new events which are sent to other processes. Each process is an independent entity (often called an agent) executing in parallel with other processes. Coordination and synchronization is by means of events communicated between the processes. A unique aspect of the CEP methodology is that event pattern constraints expressing business requirements can be used to monitor simulator output or the events from actual operations. The events could be transferred between application to application in a local environment or distributed environment. Though CEP is an effective method to manage and monitor the business process, it mandates the enterprise to adhere to the CEP methodology. A methodology to be devised to suite all range of J2EE applications to enable them monitored and managed effectively with minimum effort from the business process developer.

2.3 Business-Oriented Management

Providing the ability to application developers and others to add manageability to an application at any stage, a number of benefits arise. This ability to quickly and easily overlay access to application management information and control is most valuable to business managers. Once applications become mission critical and widely used, many ideas are generated in the business to improve visibility or use the information in the application to trigger proactive management. The ability to quickly add manageability will make it possible for short-term projects to mine this information and provide proactive monitoring.

In today's information economy, concepts such as business activity monitoring (BAM) and real-time enterprise all capture a common need of enterprises to get rapid access to information and respond quickly based on that information. BAM is a term defined by Gartner for the concept of providing real-time access to critical business performance indicators to improve the speed and effectiveness of business operations. And the real-time enterprise is a similar idea, where the end-to-end business processes in the enterprise are integrated in real-time, reducing lead-times and improving efficiency and responsiveness to customer demands.

In an enterprise J2EE deployment, application management could thus be used at multiple levels. Once application availability and performance is being monitored, administered, and managed, the IT operations problem can be considered under control. However, the applications and infrastructure are typically being used for a business purpose and the business manager's desire visibility as well. As the business use evolves, the business managers would like to monitor and act on specific business data being generated by these applications. This helps drive business decisions and achieve efficiencies and other business benefits.

Though various methodologies and technologies can be adopted to suit the business needs of a Corporate, but there are two basic ways of implementing those technologies in the management of business process:

- Instrumentation code can be made external to the business process.
- Business process and the instrumentation code can be interlaced

Before deciding on the management architecture for a J2EE platform, both the instrumentation techniques have to be analysed.

2.4 External Instrumentation

External instrumentation of the business process makes the business logic developer come clean with his work without worrying about the complex implementation of the instrumentation. External instrumentation can be implemented with technology like Java Management Extension (JMX). This requires a management tool that should be able to automate the instrumentation code as dictated by the business process developer itself or the administrator. The instrumentation code should be able to expose the data specific to the business process and can check the consistency of the business data for varying loads.

The advantages of external instrumentation can be summarized as follows:

- Ability to add and change the management information and control being exposed without touching or disrupting the application code itself.
- Existing business process can be made manageable without changing the code.
- Business logic stays clean.

AdventNet ManageEngine JMX Studio is one such tool where the business process is externally instrumented. But the strategy differs for different components in the business process. For EJBs, the instrumentation code is external and for servlet it is intrusive when it comes to performance measurement. But in most cases, the instrumentation code is external to the business process.

2.4.1 Analysis of JMX Studio

- **Performance measurement**

  ManageEngine JMX Studio can generate auto-instrumented MBeans for application response measurement which exposes attributes such as 'Method Execution Count', 'Method Start Time', 'Method End Time', 'Method Execution Time', etc., for methods in loaded EJB, servlet, or Java class. The EJB performance monitoring is non-intrusive, that is the business logic methods will not be intruded. Instead the stubs generated by the J2EE server vendor will be regenerated. To enable performance measurement, the stub regenerated EJB jar in the original J2EE application has to be redeployed. Servlets and Java class performance monitoring implementation is intrusive. The loaded servlets and Java classes, which need performance statistics such as method execution time and count for its methods, will be regenerated.

- **Business process Management:**

  Similarly, any EJB, servlet, or Java class can be loaded into ManageEngine JMX Studio such that it displays all the attributes and operations pertaining to the loaded Java class. For a given business process, JMX Studio generates JMX agent with auto-instrumented MBeans.

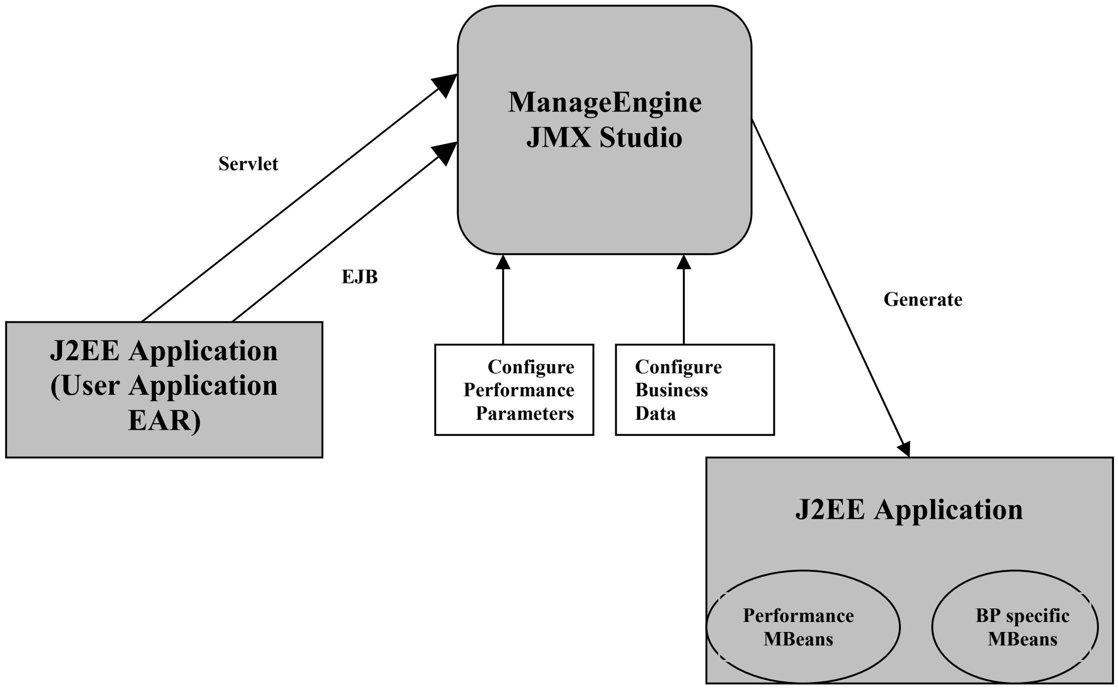


Figure 1*: AdventNet ManageEngine JMX Studio*

The figure 1 demonstrates the working of the JMX Studio. The MBeans are generated for the configuration details set by the developer or the administrator. In both the cases, the performance measurement and business management, a wrapper class is generated and the wrapper class is instrumented keeping the original business process intact. This strategy even allows to access business process related data.

JMX Studio can not be used if the actual working of the business logic is to be monitored. As management is a key part in many cases, the information that comes from the management tool should be very precise in locating the point of vulnerability. In JMX Studio, the inner details are not furnished. In some situations, where a complex business process accesses two or more data sources and involves nested transactions, the information that could be provided will not be sufficient.

2.5 Internal Instrumentation

The business process can also be instrumented by implementing instrumentation code within the business process. The tradition example is a simple *System.out.println* statement injected at the critical points of the application to get the data at the run-time to trace the point of breakdown. This is a simple and effective means of managing an application in a single Java Virtual Machine (JVM) environment. In a distributed environment, there should be a replacement of *System.out.println* statement with a light-weight instrumentation code. The instrumentation code should be able to manage and expose the manageable parameters.

Advantages of internal instrumentation are:

- To get deeper into the business process to pinpoint the exact point of vulnerability.
- The business process developer has full control of the manageability of the business logic.
- The business process related data can be exposed with ease.

Though the above mentioned advantages look promising, improper instrumentation can have adverse effect over the stability of the business process itself. Hence a managing strategy has to be developed that relieves the developer of the instrumentation technique. The other alternative could be automating the instrumentation code injection.

Aspectwerkz, an open source framework, is a management tool that injects the instrumentation code at the run time and makes the business process manageable without changing a single line of the business process. The architecture is based on Aspect-Oriented Programming (AOP) methodology. Aspect-oriented programming allows the programmer to inject pieces of functionality into existing code. The architecture is analyzed based on the article by Ramachandran Krishnamurthy in O'Reilly OnJava.com.

2.5.1 Features of Aspectwerkz

- **Agent-Server Architecture**

  To reduce the overhead on the application that is being profiled, agent-server architecture is used. The aspects* incorporate lightweight code that captures the timing information and then transmits this information to a server which is expected to run on a different machine on the network. The server parses this information and stores this information in a MySQL database. Since all of the profiling information is in a database, SQL can be queried to view the profiling data from different angles.

- **Capturing CPU Time**

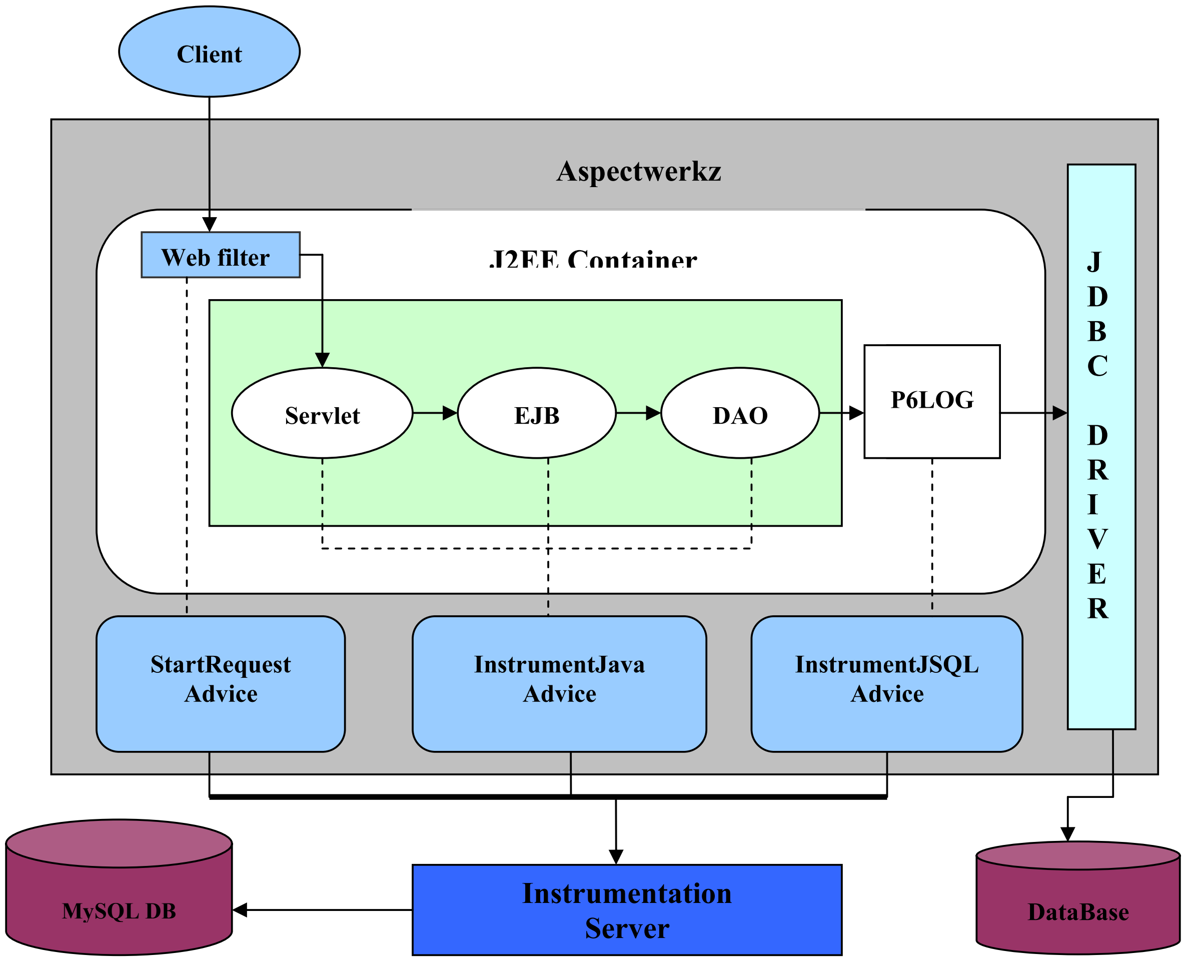
  The CPU time is a more accurate measure of the execution time of a method. The CPU time is captured using the JVM Profiler Interface* (JVMPI).

- **Capturing SQL Execution Time**

  Most J2EE applications are data-centric and typically persist data in relational databases. A critical aspect of performance analysis of a J2EE application would therefore rely on the timing information of the SQLs fired from the J2EE application. The SQL execution time is captured by utilizing the P6Log* driver. This piece of software acts as a layer between the J2EE connection pool and the actual JDBC driver and captures the timing information of the SQLs fired. Aspects are applied to this software to retrieve that information.

- **Capturing the Sequence of Method Execution**

  The flow of information is captured using ThreadLocal variables. The ThreadLocal variable holds a unique ID for each request, along with a sequence number that runs in the order of the method execution. The limitation in this implementation is that the sequence can be captured meaningfully only when all of the components that are to be profiled are executed in the same JVM; i.e., there are no remote calls.



Fgure 2: ***Architecture of Aspectwerkz***

The figure 2 demonstrates the architecture of the Aspectwerkz. The demonstration includes a business process that has servlet and EJB which follows the Data Access Object* (DAO) pattern. The *InstrumentJava* advice measures the performance metrics of servlet, EJB and DAO. The *InstrumentJSQL* measures the performance metrics of the SQL queries.

2.5.2 Analysis of Aspectwerkz

The Aspectwerkz helps in getting the performance metrics of the business process without the necessity of change in code. This is a huge advantage when the need is just to measure the performance of the business process. The aspects are the pieces of instrumentation code that are injected into the compiled java code of the business process. Though this can be termed as internal instrumentation, but the actual implementation of the management part stays out of the business process. The point of injection of the instrumentation code is decided by the administrator.

The Aspectwerkz concentrates more on the performance metrics and does not provide a solution to monitor/manage business process related data. This data is very much needed to study the stability of the business process. The business process related data normally are a result of some complex transaction or a complex logic that involves more than one data source. In this scenario, it is hard to pinpoint the point of failure or predict the point of failure. Though the execution time of the SQL queries give some picture of the point of break down but it is absolutely necessary to get the exact point of breakdown.

2.6 Best practices for Business Process management

The basic elements of application management include administration and monitoring of the application infrastructure, as well as administration and monitoring of the applications themselves. Not all enterprise J2EE deployments demand sophisticated application management. The extent of the demand for effective application management will be driven by the scale, maturity, and importance of applications to the enterprise. And as J2EE deployments grow in these dimensions, good solutions to address the administration and monitoring of the J2EE applications and infrastructure deployed by enterprise IT are being deemed a priority. And application deployments that integrate well with the J2EE management architecture will be more likely to be successful.

Beyond basic application management, business use of the applications in mission-critical processes drives the demand for management functionality to support the business process and decisions. Application and business-specific data needs to be monitored to ensure proper response to changing conditions and dependencies across systems. The business needs may be specific to the business, but the value of being able to expose and act on application and business data is widely appreciated by business and IT managers. This ability to expose application and business data for management requires internal instrumentation as discussed earlier.

The diversity of applications and application infrastructures has always made application management a challenge. Despite the best efforts of management vendors and others, there have been many hurdles in delivering effective application management to meet the needs of enterprise IT. A variety of management standards, technologies, and products have made small dents in the problem, but without widespread success. Meanwhile, the awareness of and business need for application management have been growing. The emergence of J2EE and other standardized application infrastructures with standard component models and “managed code” is in response to the strong need of enterprise IT for better control of application infrastructure. However, J2EE tools currently offered by the J2EE vendors address only a part of the application management problem.

An emerging management standard from the Java community is showing promise in directing us to a solution for both the core application management and business management needs. Java Management Extensions (JMX) is a standard being adopted by the Java industry to meet many of the application management challenges. The powerful model adopted by JMX for simpler instrumentation and integration with existing management standards makes it a good way to bring together the elements needed to address the management problem. Though JMX meets most of the requirements of management, it fails in case of clustered environment. A detailed analysis on this issue is made in a later chapter. Hence another strategy with Java Message Service (JMS) technology is also adapted to enable business-oriented management.

3 Business Process management with Java Management Extension (JMX)

Java Management Extension (JMX) API and the ManagedBean (MBean) can be harnessed to monitor the "inside" of the BPs at runtime.

3.1 Java Management Extension (JMX)

JMX is about providing a standard for managing and monitoring all varieties of software and hardware components from Java. Further, JMX aims to provide integration with the large number of existing management standards. The architecture of JMX has three levels.

The level closest to the application or the resource is called the instrumentation level. This level consists of four approaches for instrumenting application and system resources to be manageable, as well as a model for sending and receiving notifications. The middle level of the JMX architecture is called the agent level. This level contains a registry for handling manageable resources as well as several agent services. The third level is called the distributed services level. This level contains the middleware that connects JMX agents to applications that manage them. The architecture can be described as shown in the figure 3.
The overview of the JMX technology that follows is based on "JMX in action" by Benjamin.G.Sullins.

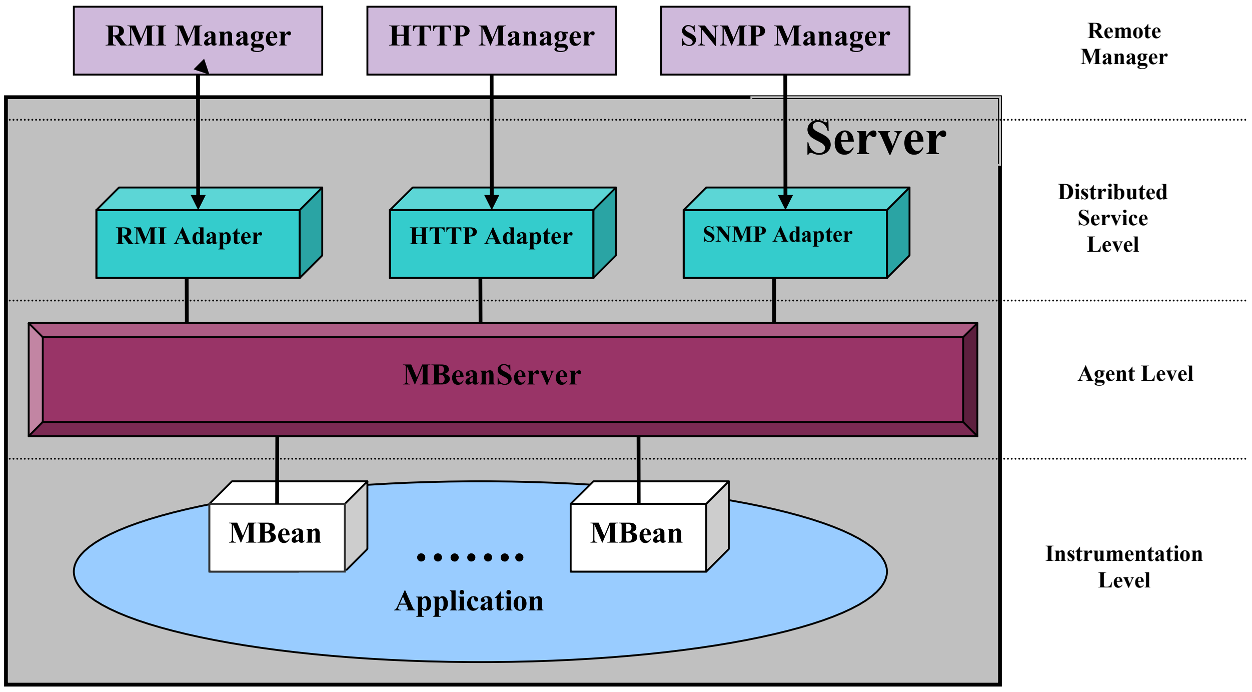


Figure3: ***JMX Architecture***

3.2 The Instrumentation Level

The instrumentation level defines the requirements for implementing JMX manageable resources. A JMX manageable resource can be virtually anything, including applications, service components, devices, and so on. The manageable resource exposes a Java object or wrapper that describes its manageable features, which makes the resource instrumented so that it can be managed by JMX-compliant applications. The user provides the instrumentation of a given resource using one or more managed beans or MBeans. There are four varieties of MBean implementations: standard, dynamic, model and open.

Each resource that is to be managed must provide a management interface, which consists of the attributes and operations it exposes so that it can be monitored and controlled by a management application. An MBean is an application or system resource that has been instrumented to be manageable through JMX. The management interface of an MBean is composed of the four following items:

- **Public constructors**
  MBeans can be dynamically loaded into JMX agents. Agents do this using any of the public constructors exposed by the MBean. Constructors are included in the definition of the management interface because a particular constructor could define specific behavior over the life of the MBean object. For instance, one constructor may tell the MBean to log all of its actions, and another may make it silent. Any way of altering the behavior of an MBean is included as part of its management interface.

- **Attributes**
  Attributes are the vital part of the management interface of an MBean. The attributes describe the manageable interface. A manageable resource is some application or resource exposed for management by an MBean.

- **Operations**
  Operations correspond to the actions that can be initiated on the manageable resource. Operations are methods like any other; they can have multiple parameters and optionally return a value.

- **Notifications**
  Notifications allow MBeans to communicate with registered listeners. In order to emit notifications, an MBean must implement the *javax.management.NotificationBroadcaster* interface. This interface provides methods for sending notifications, as well as methods for other objects to register as listeners on the implementing MBean.

3.2.1 Standard MBean

JMX provide a set of patterns to follow when instrumenting application resources as Standard MBeans. Standard MBeans are the simplest type of MBean to code from scratch. There are three patterns that are to be followed when instrumenting a resource as a Standard MBean:

- The management interface of the resource must have the same name as the resource's java class, followed by 'MBean'; it must be defined as a Java interface; and it must be implemented by the resource to be managed using the *implements* keyword.
- The implementing class must contain at least one public constructor.
- Getters and setters for attributes on the management interface must follow strict naming conventions.

The metadata required of every MBean is created automatically by the JMX infrastructure for standard MBeans. Before an MBean can be managed, it must be registered with a JMX agent. When a standard MBean is registered, it is inspected and metadata placeholder classes are created and maintained by the JMX agent on behalf of the MBean. The Java reflection API is used to discover the constructor on the MBean class, as well as other features. The attribute and operation metadata comes from the MBean interface and is verified by the JMX agent.

A simple StandardMBean:

```
Public interface SampleMBean {

      Public long getPrintTime();
      Public void setPrintTime(long time);

      Public void resetTime(long old, long new);

}
```

The above snippet of code demonstrates a simple *StandardMBean* with attribute *PrintTime* of type long and an operation *resetTime*. The resource that is being instrumented in the sample code above is a printer. The implementation of this interface actually instruments the resource.

3.2.2 Dynamic MBean

Standard MBeans are well suited for management interfaces that are relatively static. However, if a management interface must be defined for an existing resource, is likely to evolve over time, or for some other reason needs to be exposed at runtime, JMX provides an interface that allows doing just that. The main reason to use dynamic MBeans is to more easily instrument existing code that is written in a manner that conflicts with the standard MBean design pattern as discussed in the section 3.2.1. The dynamic MBean interface is determined not through introspection, but rather through a method call on the dynamic MBean itself. This method, called *getMBeanInfo()*, returns information about the management interface and is defined on the *DynamicMBean* interface; it is the portal through which a management application views what has been exposed on the management interface of a resource that has been instrumented as a dynamic MBean. This is demonstrated in figure 4. If there is a change in the resource from version 1 to version 2, there is no need to change the management interface.

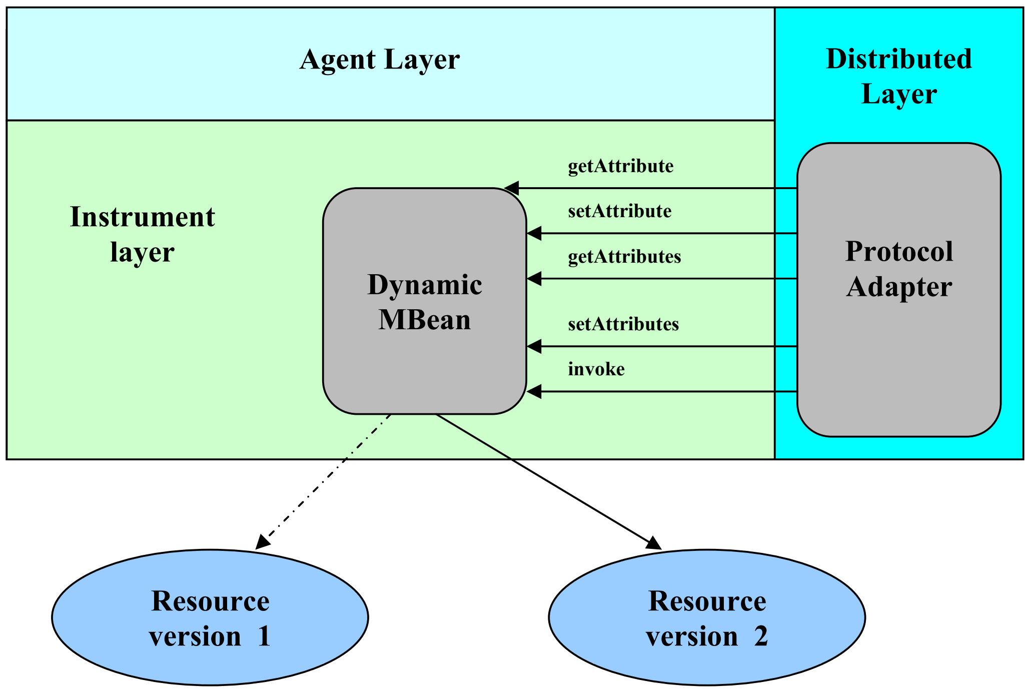


Figure 4: ***Dynamic MBean insulating an evolving implementation***

An MBean feature is an attribute, constructor, operation, parameter or notification of an MBean. Description of these features can also be provided that are visible to the management application. The feature description is a brief explanation of what a particular feature means when viewed from a management application. The feature's name usually indicates what it means, but this is not always the case.

As the dynamic MBean interface is exposed at runtime, rather than at compile time, the management interface is exposed through metadata classes. If the management interface is likely to change over time, dynamic MBeans offer a more flexible way to instrument a resource. The management interface is not statically bound to a dynamic MBean. Rather, the management interface is exposed dynamically. As such, it is conceivable that a dynamic MBean could expose a different interface from one instance to the next by reading which attributes and operations to expose from a configuration file.

When the MBean server is asked to register a dynamic MBean, no introspection is performed. Dynamic MBeans use metadata classes to expose their management interfaces. They make that metadata available through their management interface called *DynamicMBean*, which must be implemented by all dynamic MBeans. The *DynamicMBean* interface is as follows:

```
public interface DynamicMBean {
  public Object getAttribute(String attribute);
  public void setAttribute(Atribute attribute);
  public AttributeList getAttributes(String[] attrs);
  public AttributeList setAttributes(AttributeList attrs);
  Public Object invoke(String actionName, Object params[], String
sig[]);
  public MBeanInfo getMBeanInfo();
}
```

Essentially, the DynamicMBean interface provides a way for a management interface to do four things:

- Dynamically discover the management interface exposed by the MBean (*getMBeanInfo()*).
- Retrieve the value of one or more attributes on the management interface (*getAttribute()* and *getAttributes()* respectively).
- Set the value of one or more attributes on the management interface (*setAttribute()* and *setAttributes()* respectively).
- Invoke an operation on the management interface (*invoke()*).

Dynamic MBeans tell the MBean server that they are dynamic MBeans by exposing the *DynamicMBean* interface, but it is the use of the dynamic MBean metadata classes that ties it all together. There are six significant metadata classes:

- *MBeanInfo*
  The top-level container of metadata; each MBean requires only one instance of this class to completely describe its management interface. This class is a standard JMX class containing classes that describe individual parts of the overall management interface. *MBeanInfo* contains a metadata object for each of the parts that are described as follows.

- *MBeanAttributeInfo*
  Each instance of this class provides information about a single attribute.

- *MBeanParameterInfo*
  Each instance of this class provides information about a single parameter.

- *MBeanConstructorInfo*
  Each instance of this class provides information about a single constructor and may ^ contain one or more *MBeanParameterInfo* instances.

- *MBeanOperationInfo*
  Each instance of this class provides information about a single operation and may contain one or more *MBeanParameterInfo* instances.

- *MBeanNotificationInfo*
  Each instance of this class contains information about a group of notifications emitted by this MBean.

Thus dynamic MBeans provide their management interface at runtime. This ability equips dynamic MBean to manage evolving resources over time. Developers can easily adapt dynamic MBean as their resources change.

3.2.3 Model MBean

Model MBeans are generic MBeans that can be instantiated in the MBean server and configured by a user to manage any resource. The Model MBean's main difference from the Standard and Dynamic MBeans is that MBean class need not be developed, which means that without writing any MBean code, the resources can be instrumented using a management tool interfacing with a JMX agent. Other than instrumenting the resources, Model MBeans also provide the following features:

- **MBean Persistence**
  Model MBean has the ability to persist itself. By using its persistence mechanism, a Model MBean can survive the cycling of the JMX agent that contains it.

- **Notification logging**
  Model MBean has the ability to log each notification it emits. Using this mechanism, an accurate record of all the notifications that are sent by a particular MBean can be maintained.

- **Attribute value caching**
  Model MBeans can cache attribute values. This improves performance. For instance, a Model MBean can be configured to locally store the value of an attribute after it is first acquired. The subsequent requests for this attribute can be satisfied with the local copy. How often the cache is updated is determined by the caching policy associated with the specific attribute and configured by the user.

- **Operation delegation**
  Model MBean can have operations in its management interface that are invoked on objects other than its managed resource. When exposing a particular method for management, optional *Object* reference can be included in which to invoke the operation. The delegation helps in exposing operations that may interact with more than just single manageable resource.

- **Generic notification**
  Model MBean also provides methods to send out generic, purely informational notifications. In the model MBean implementation, there is a method that accepts a *String* argument to be sent out as notification.

Model MBeans are Dynamic MBeans and so use metadata to describe the features of the MBean. The UML diagram in figure 5 illustrates this.

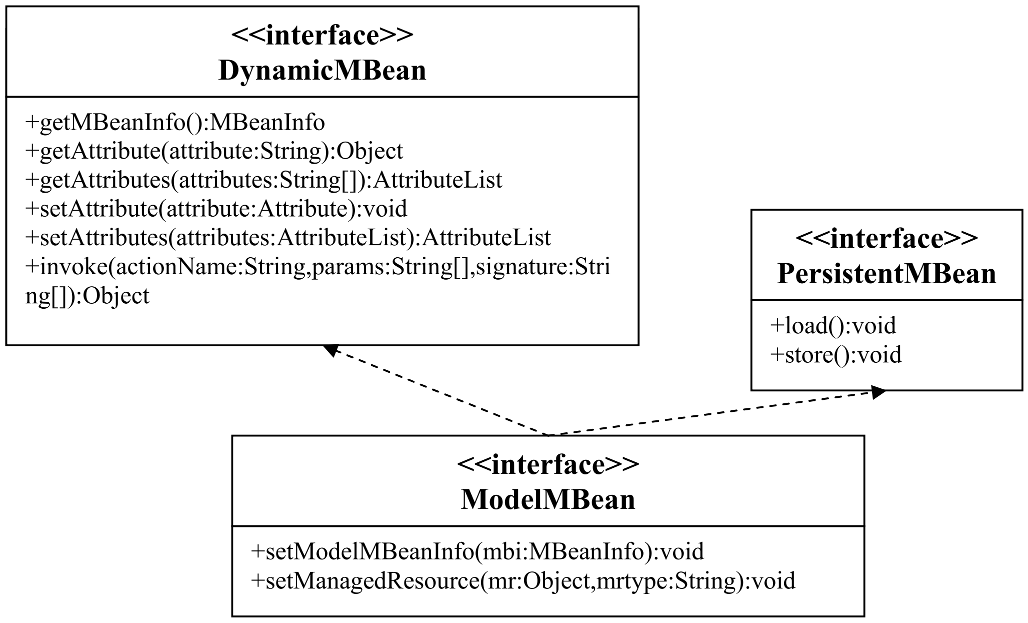

<<interface>>
DynamicMBean
+getMBeanInfo():MBeanInfo
+getAttribute(attribute:String):Object
+getAttributes(attributes:String[]):AttributeList
+setAttribute(attribute:Attribute):void
+setAttributes(attributes:AttributeList):AttributeList
+invoke(actionName:String,params:String[],signature:String[]):Object
<<interface>>
PersistentMBean
+load():void
+store():void
<<interface>>
ModelMBean
+setModelMBeanInfo(mbi:MBeanInfo):void
+setManagedResource(mr:Object,mrtype:String):void

Figure 5: ***UML diagram of the ModelMBean interface***

The *ModelMBean* provides the following two methods that initialize the MBean for outside use:

- **setManagedResource(Object resource, String resourceType)**
  This method sets the MBean's managed object. The object is the reference in which the operations will be invoked and attributes accessed. The *resourceType* parameter tells the MBean what type of Object reference is being passed in. It can have the value *ObjectReference*, *Handle*, *IOR*, *EJBHandle* or *RMIRefernce*.

- **setModelMBeanInfo(ModelMBeanInfo info)**
  The *ModelMBeanInfo* parameter is the metadata object collection that describes the management interface of this *ModelMBean*.

By extending *DynamicMBean* interface, all *ModelMBeans* are really *DynamicMBeans*. That being so, a *ModelMBean* defines its management interface at runtime like any other *DynamicMBean*. However, where *DynamicMBeans* are user-developed classes that construct their own *MBeanInfo* objects to define their management interface, *ModelMBeans* are standard JMX classes and must have their *MBeanInfo* created and placed inside them using the *setModelMBeanInfo* method.

However, there is one significant difference, Model MBeans offer the instrumentation developer a metadata class called *Descriptor*, which a collection of name/value pairs in which the name is a *String* and value is an *Object*. This allows for a much richer set of metadata to be exchanged with the agent level, other MBeans and management applications. When a resource's attributes are accessed or changed, or when an operation is invoked, the mechanism used by Model MBeans is *callback*. When the metadata for an MBean feature is created, a reference to the instance of the resource is stored with the metadata, along with the name of the attribute getter/setter or operation. When a management application manages the MBean, it simply uses this information to call back into the resource.

The resources that are instrumented using Model MBeans do not require any code changes. This is a significant advantage when instrumenting existing application or third-party resources that provide a well-defined API. Unlike Standard or Dynamic MBeans, the resource itself does not have to implement anything to be perfectly compliant JMX resource. All that is required is that somewhere in the code execution stream there must be a code that creates the necessary *Descriptor* and other metadata classes to instrument the resource. A logical place for this code is the resource itself, but JMX does not require this.

3.2.4 Open MBean

Open MBeans are used to instrument resources whose attributes are more complex than the fundamental types and whose operations take complex parameters. The key to this more open means to instrumentation lies in the set of data types defined by the JMX specification called Open MBeans. By using Open MBeans, application resources of any type can be instrumented and make the available to any agent or management application that does not have access to the *bytecode* for either the resource, attribute or operation parameter. The agent or management application can even be a non-java program.

3.3 Agent level

The main component of the agent layer is the MBean server. An MBean server is a java object that acts as a registry for MBeans; it is the heart of a JMX agent. The agent layer provides access to managed resources from the management application. A JMX agent can run in a JVM embedded in the machine that hosts the resources, or it can be remotely located. The agent does not require knowledge of the resources that it exposes or the manager application that uses the exposed MBeans. It acts as a service for handling MBeans and allows manipulation of MBeans through a collection of protocols exposed via connectors or adapters. The core component of a JMX agent is the MBean server, a managed object server in which MBeans are registered. A JMX agent also includes a set of services to manage MBeans, and at least one communications adaptor or connector to allow access by a management application.

3.3.1 MBean Server

A key component of the agent level is the managed bean server. Its functionality is exposed through an instance of the *javax.management.MbeanServer*. An *MBeanServer* is a registry for Mbeans that makes the MBean management interface available for use by management application. The Mbean never directly exposes the MBean object itself; rather, its management interface is exposed through metadata and operations available in the *MBeanServer* interface. This provides a loose coupling between management applications and the Mbeans they manage.

Mbeans can be instantiated and registered with the MbeanServer by the following:

- Another MBean.
- The agent itself.
- A remote management application.

The *javax.management.MBeanServerFactory* interface implementation should create or give an existing *MBeanServer*. *MBeanServer* implementation can be used to locate an MBean or register a new MBean. When an MBean is registered, it must be given an unique ObjectName. The ObjectName then becomes the unique handle by which management applications identify the object to perform management operation. *ObjectName* represents the object name of an MBean, or a pattern that can match the names of several MBeans. Instances of this class are immutable. An *ObjectName* consists of two parts, the domain and the key properties. The pattern will look like this:

```
domain: key1 = value1 , key2 = value2…
```

The operations available through the *MBeanServer* include the following:

- Discovering the management interface of MBeans
- Reading and writing attribute values
- Invoking operations defined by MBeans
- Registering for notification events
- Querying Mbeans based on their object name or their attribute values

The JMX agents support a query mechanism that can build and execute complex queries. Queries are submitted to a JMX agent for the purpose if retrieving a set of *ObjectInstance* objects. In essence, a query identifies all the Mbeans that conform to the rules of a given query. The two methods, *queryNames()* and *queryMBeans()*, both accept an *ObjectName* instance and a *QueryExp* instance. The ObjectName instance defines the scope of the query, and the QueryExp instance defines the constructed query expression. The queryMBeans method returns a set of ObjectInstance objects. The ObjectInstance class constains the ObjectName of an MBean and the MBean's defining class type. The queryNames method only returns a set of ObjectNames of MBeans. Both the methods execute the query in the same manner; only their return type differs.

3.4 Distributed services level

The general purpose of this level is to define the interfaces required for implementing JMX management applications or managers. The following points highlight the intended functionality of the distributed services level:

- Provides an interface for management applications to interact transparently with an agent and its JMX manageable resources through a connector.

- Exposes a management view of a JMX agent and its MBeans by mapping their semantic meaning into constructs of a data-rich protocol.

- Distributes management information from high-level management platforms to numerous JMX agents

- Consolidates management information coming from numerous JMX agents into logical views that are relavant to the end user's business operations

- Provides security.

It is intended that the distributed services level components will allow for cooperative management of networks of agents and their resources. These components can be expanded to provide a complete management application.

Protocol adaptors and connectors are the prime components of the distributed service level. These components are required to access the *MBeanServer* from outside the agent's JVM. Each adaptor provides a view via its protocol of all Mbeans registered in the *MBeanServer* the adaptor connects to. An example adaptor is an HTML adaptor that allows for the display of MBeans using a web browser.

Protocol adapters and connectors are very similar in that they serve the same overall purpose: to open a JMX agent to managing entities. The difference between them is how they go about it. Protocol adapters generally must listen for incoming messages that are constructed in a particular protocol like HTTP or SNMP. In this sense, protocol adapters are made up of only one component that resides in the agent at all times. Connectors, on the other hand, are made up of two components: one component resides in the JMX agent, and the other is used by client-side applications. Clients use the client-side connector component to contact the server-side component and communicate with a JMX agent. In this manner, connectors hide the actual protocol being used to contact the agent; the entire process happens between the connector's two components.

3.5 Notification Model

A notification in the context of JMX is a unit of information sent by a broadcaster through the JMX infrastructure to a listener, which interprets and processes the notification. A notification contains, at a minimum, the notification type, that uniquely identifies the notification), an Object reference to the notification broadcaster, and a sequence number, that uniquely identifies a particular occurrence of a specific notification type. Other optional information that can be sent in a notification includes a time stamp, a human-readable text message, and a reference to an object that permits additional processing of the notification to occur. The type of this object must be agreed upon by the listener and the implementation of the broadcaster.

The main components in the Notification model include

- Notification Broadcaster
- Notification
- Notification Listener
- Notification Filter

3.5.1 Notification Broadcaster

A notification broacaster is an Mbean that implements the *javax.management.NotificationBroadcaster interface*. The interface contains methods for adding and removing listeners. The methods allow objects to register as listeners for the notifications an MBean can emit. Listeners provide a callback method that broadcasts invoke in order to deliver a notification. The method *addNotificationListener* takes three arguments namely: *NotificationListener*, *NotificationFilter* and an *Object*. The first two are discussed later in this chapter. The last argument is an *Object* which is an *handback*. This value is sent back to the listener when a notification is delivered and should never be modified by the broadcaster. The handback object can be used to provide a broadcast with a context for the listener

The method *getNotificationInfo* returns an array of objects of type *MbeanNotificationInfo*. *MbeanNotificationInfo* isa member of the set of metadata objects used to describe the management interface of an MBean. It is used here separately to ensure that broadcasters provide information about the types of notification they emit.

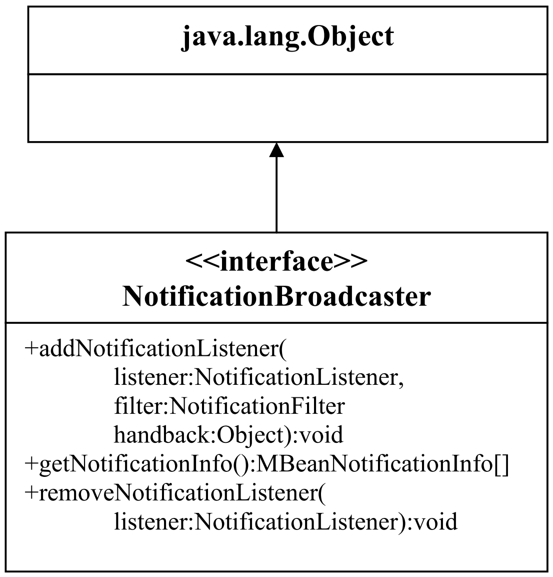

java.lang.Object
<<interface>>
NotificationBroadcaster
+addNotificationListener(
listener:NotificationListener,
filter:NotificationFilter
handback:Object):void
+getNotificationInfo():MBeanNotificationInfo[]
+removeNotificationListener(
listener:NotificationListener):void

Figure 6: ***UML diagram of the NotificationListener***

The other way to implement the notificationBroadcaster interface is to extend the *javax.management.NotificationBroadcasterSupport* class. By extending this class, one can inherit the implementation of the *NotificationBroadcaster* interface. In addition, *NotificationBroadcasterSupport* class provides an extra method called *sendNotification.* The method *sendNotification* provides mechanism for sending a *Notification* object to the registered listener. This method attempts to send its notification arguments to each registered listeners after first applying that listener's filter object. If the filter indicates that the listener should receive that notification, then it is sent.

3.5.2 Notification

JMX provides a standard notification class, *javax.management.Notification*. This class extends *java.util.EventObject* and is used as a super class for other notification class. The Notification class contains six member variables that are all accessible through getter methods. This class has several different constructors, each providing a different set of initialization arguments for these class members.

The class members of the *Notification* class are:

- **Message**
  A *String* object representing a message.

- **SequenceNumber**
  A number indicating the order in relation of events from the source. The source populates thie field if it intends to give the listeners the ability to sort incoming notifications. The notification model makes no guaranties that notifications will be received in the order they were sent.

- **TimeStamp**
  The timestamp of the notification, represented as a *long* value.

- **Type**
  The dot-separated String value indicating the type of the notification.

- **UserData**
  An object used to contain any data that a source wants to send to a notification listener.

- **Source**
  The source of the notification. This object contains an *ObjectName* or a reference to the object that generated the notification.

3.5.3 Notification Listener

The objects that are interested in receiving notifications must implement the *javax.management.NotificationListener* interface. The interface contains a single method: *handleNotification*. It takes two arguments: an instance of *Notification* and an instance of *Object*. Notification broadcasters invoke this method when they are ready to deliver a notification to the listener. The instance of *Notification* is the notification being sent,and the *Object* instance is the handback object registered by the listener. The method *handleNotification* of the listeners will be called by the broadcasters to which it is registered when a notification is emited.

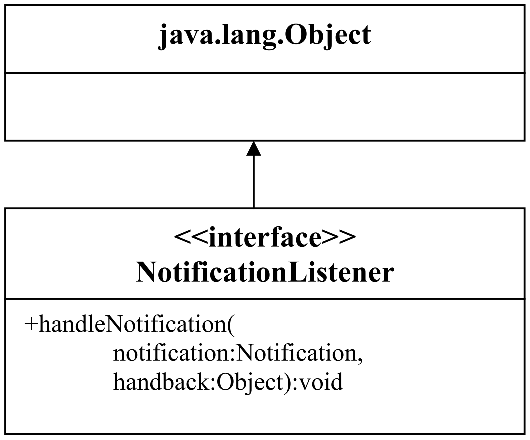


Figure 7: ***UML diagram of the NotificationListener interface***

3.5.4 Notification Filter

The only component remaining is the notification filter. Because MBeans can emit an infinite number of notification types, listeners can use a filter to ensure they receive only the specific notification types in which they are interested. A notification filter give notification listeners a way to sort through a potential barrage of notifications to receive only those notifications that are important to them. To be more accurate, notification broadcasters use a registered listener's filter to determine whether to send a notification to a listener.

The *NotificationFilter* interface declares only one method: *isNotificationEnabled*. This method accepts a *Notification* object that is about to be sent and returns a *boolean* value indicating whether a listener associated with this filter wants to receive the notification.

3.6 Architecture of the Monitor

This architecture is based completely on JMX. The design goals of the architecture are

1. The logging part at the business process should be as simple as possible. This should not make the business process development complicated.
2. The logging service that includes the strategy to publish information should be hidden from the business process developer.
3. The logging service should not affect the performance of the business process.
4. The architecture should provide space for the administrator to stop the logging service or set filter to the logging service at the business process level to stop it from publishing the information.
5. The monitor should also have the provision to set filter and see in the information that the administrator wants to see.

The messaging system is bases on the Notification model of the JMX. Figure 8 describes the architecture.

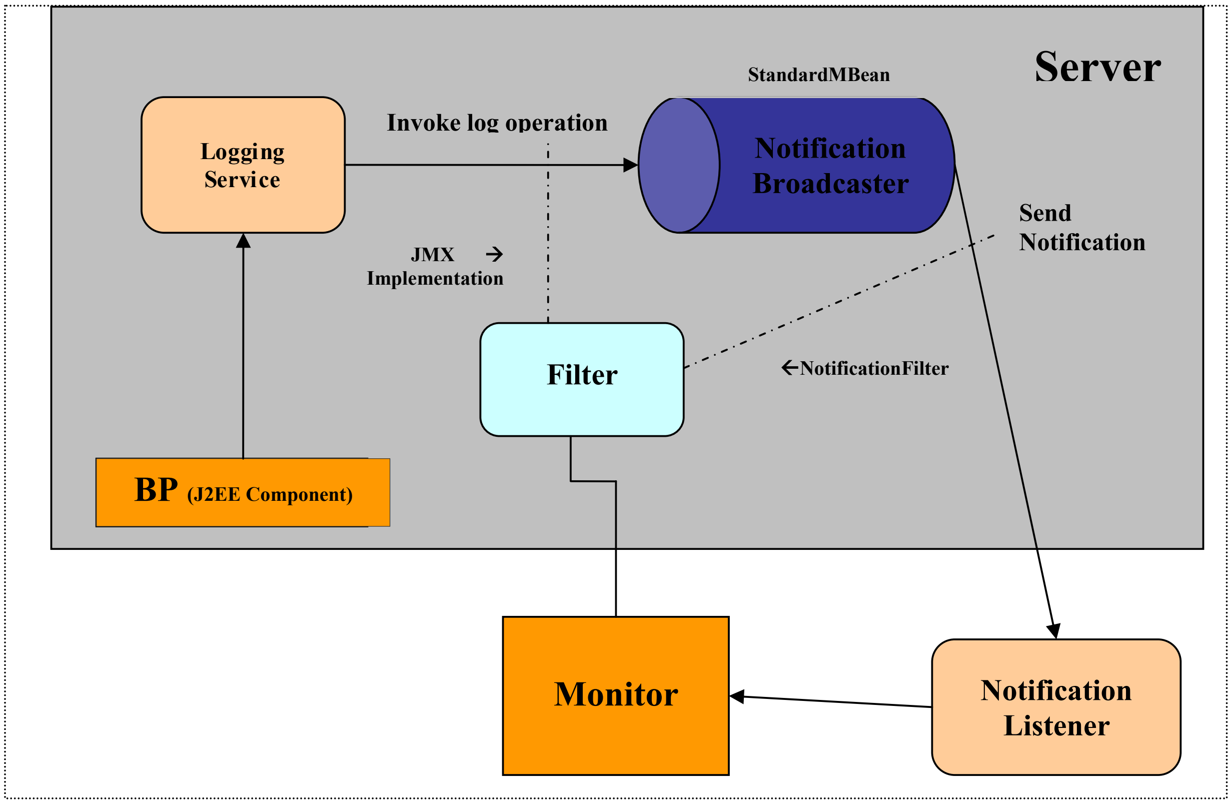


Figure 8: ***Architecture of the monitor based on JMX***

3.6.1 Logging Service

Logging service abstracts the implementation details from the business process. First task of the logging service is to obtain an instance of the MBeanBroadcaster. At the time of deployment, the broadcaster MBean is registered to the *MbeanServer*. For a given application server, there will be only one *MbeanServer* instance. *MbeanServer* is a singleton. The logging service should get an instance of the MbeanServer from the local application server, that is, the server in which the business process is deployed. Obtaining an instance of *MBeanServer* is application server specific. For instance, in JBoss,

```
MBeanServer server = (MBeanServer)
org.jboss.mx.util.MBeanServerLocator.locateJBoss();
```

To make the architecture application server independent, Java 2 Platform, Enterprise Edition Management Specification (JSR-77) is used. The goal of JSR77 is to abstract the fundamental manageable aspects of the J2EE architecture to provide a well defined model for implementing instrumentation and information access. In addition, this specification defines a standardized API for interoperating with J2EE components that participate in the monitoring and control of the platform's resources. The details of JSR-77 are discussed in the later chapter.

The J2EE Management EJB component (MEJB) provides interoperable access to the J2EE Management Model from any J2EE component on all platforms that implement the J2EE Management specification. The MEJB component incorporates the Java Management Extensions (JMX) API, a standard framework for Java object instrumentation. The MEJB component exposes the managed objects on any J2EE platform as JMX manageable resources as defined by the Java Management Extensions Instrumentation and Agent Specification (JSR003). The MEJB component provides local and remote access of the platform's manageable resources through the EJB interoperability protocol.

The business process, an EJB or servlet component, creates an instance of the logging service. For each bean, there should be only one instance of logging service. This insures that the bean does not make multiple requests to get a home instance of the MEJB. Lazy initialization of the logging service will make sure that there is only one instance of logging service, in particular one instance of the MEJB. The following code snippet shows how the logging service should be incorporated in an EJB:

```
protected LoggingService getLoggingService() throws Exception {
      if (loggingService == null) {
            loggingService = new
            LoggingService(getServiceLocator(),"SERVICE_NAME");
      }
      return loggingService;
}
```

The logging service adheres to the service locator pattern as described by Deepak Alur et al in Core J2EE Patterns.

Service Locator object is created to abstract all JNDI usage and to hide the complexities of initial context creation. Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control and improve performance by providing a caching facility. So this pattern provides a mechanism to abstract all dependencies and network details into the Service Locator.

The main functionality of the Service Locator in this context is to make sure that for a given LoggingService there is only one instance of the MBeanServer instance of an application server instance. An MBeanServer instance can be obtained through a Remote Method Connection (RMI) or through some proprietary connectors. All the management interfaces are obtained through the MBeanServer that is got through the Service Locator. The service locator pattern is described through Sequence diagram in the figure 9. In the figure the client is the instance that gets an access to the MBeanServer. The client in the context of this architecture is the LoggingService.

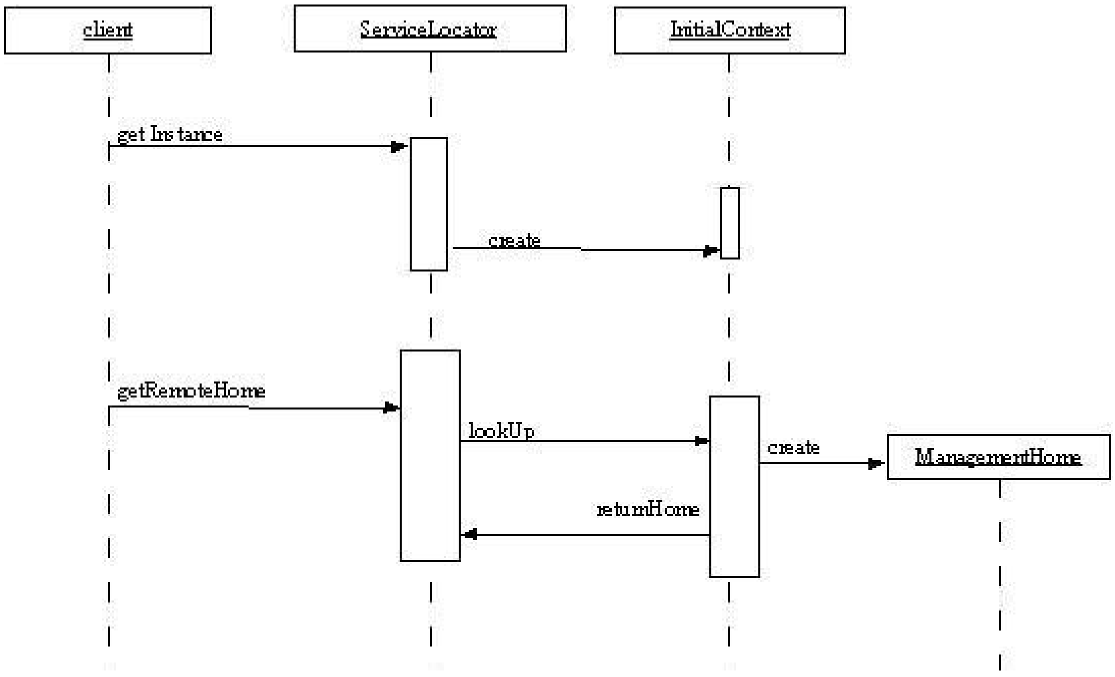


Figure 9: ***Sequence diagram for Service Locator for JMX implementation***

The logging service constructor will make the necessary JNDI lookups through the service locator pattern to get the home interface of the MEJB.

```
public LoggingService (ServiceLocator sl, String serviceName) {
      // necessary initialization
      mHome
      =(ManagementHome)serviceLocator.getRemoteHome("/ejb/mgmt/
      MEJB", ManagementHome.class);
      mejb = mHome.create();
      name = new ObjectName("ObjectNameOfTheBroadcaster");
}
```

The *log* method implementation invokes a method in the MBean which in turn broadcasts the information to the registered listeners. The *log* method packs all the arguments in a class and sends as argument to the *log* method of the broadcaster MBean.

3.6.2 StandardMBean interface

The MBean is the heart of the business process management architecture. The MBean expose the information that is passed on from the business process. The resource or the application that the MBean instruments in this case is the business process itself. The main purpose of the MBean is to notify all the log information from the business process to all the listeners that are registered themselves to this MBean to receive all the notifications. The StandardMBean interface looks as shown in the code snippet that follows:

```
Public interface LoggerMBean extends NotificationBroadcaster {
     public void log(Object obj);
}
```

3.6.3 Notification Broadcaster

Notification broadcaster is basically a *StandardMBean* that implements the *NotificationBroadcaster* interface. An instance of *Notification* class should be created based on the log information from the BP.

```
Notification notif = new Notification(LogRecord.getClass().getName(),
this, -1);
notif.setUserData(record);
```

An instance of *Notification* is what basically broadcast to the listeners that are registered with the broadcaster. The listeners could register with the broadcaster through the method *addNotificationListener* which accepts two main parameters: *NotificationListener* and *NotificationFilter*. The *log* method of the broadcaster will implement a method *sendNotification* that sends the *Notification* instance to all its listeners.

```
Public void log(LogRecord record) {
      //create an instance of Notification - notif
      sendNotification(notif);
}
```

The implementation of *sendNotification* should also check the *NotificationFilter* instance before broadcasting the information to the registered listeners.

3.6.4 Notification Listener

The notification listener is close to the monitor. The listener should implement the *NotificationListener* interface. The *handleNotification* is called each time broadcast is made. The handleNotification recives the *Notification* instance which actually is a wrapper class of the information that is to be logged. The *handleNotication* method implementation should retrieve the necessary information from the *Notification* object. The information thus retrieved is then produced on the management console.

4. Business process management with Java Message Service (JMS)

Enterprise Java Beans and Servlets are key parts of Java applications because they define the business logic that manages each business process component. Managing a business process will mean monitoring the EJBs or Servlets. This can be achieved by publishing the information at each critical point. One way of achieving this is through Java Message Service (JMS) developed by Sun Microsystems.

4.1 Java Message Service (JMS)

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages. The following overview of the JMS technology is based on the tutorial from Sun Microsystems.

Messaging enables distributed communication that is *loosely coupled*. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only what message format and what destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Enterprise messaging products are becoming an essential component for integrating intra-company operations. They allow separate business components to be combined into a reliable, yet flexible, system. In addition to the traditional Message Oriented Middleware (MOM) vendors, enterprise messaging products are also provided by several database vendors and a number of internet related companies. Java language clients and Java language middle tier services must be capable of using these messaging systems. JMS provides a common way for Java language programs to access these systems. JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product. It attempts to minimize the set of concepts a Java language programmer must learn to use enterprise messaging products. It strives to maximize the portability of messaging applications.

The JMS API enables communication that is not only loosely coupled but also

- Asynchronous: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
- Reliable: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

At the 1.2 release of the J2EE platform, a service provider based on J2EE technology was required to provide the JMS API interfaces but was not required to implement them. Now, with the 1.3 release of the J2EE platform, the JMS API is an integral part of the platform, and application developers can use messaging with components using J2EE APIs.

The JMS API in the J2EE 1.3 platform has the following features.

- Application clients, Enterprise JavaBeans (EJB) components, and Web components can send or synchronously receive a JMS message. Application clients can in addition receive JMS messages asynchronously. (Applets, however, are not required to support the JMS API.)
- A new kind of enterprise bean, the message-driven bean, enables the asynchronous consumption of messages. A JMS provider may optionally implement concurrent processing of messages by message-driven beans.
- Message sends and receives can participate in distributed transactions.

4.1.1 JMS API Architecture

A JMS application is composed of the following parts.

- A *JMS provider* is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the J2EE platform at release 1.3 includes a JMS provider.
- *JMS clients* are the programs or components, written in the Java programming language, that produce and consume messages.
- *Messages* are the objects that communicate information between JMS clients.
- *Administered objects* are preconfigured JMS objects created by an administrator for the use of clients.
- *Native clients* are programs that use a messaging product's native client API instead of the JMS API. An application first created before the JMS API became available and subsequently modified is likely to include both JMS and native clients.

Administrative tools allow you to bind destinations and connection factories into a Java Naming and Directory Interface (JNDI) API namespace. A JMS client can then look up the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

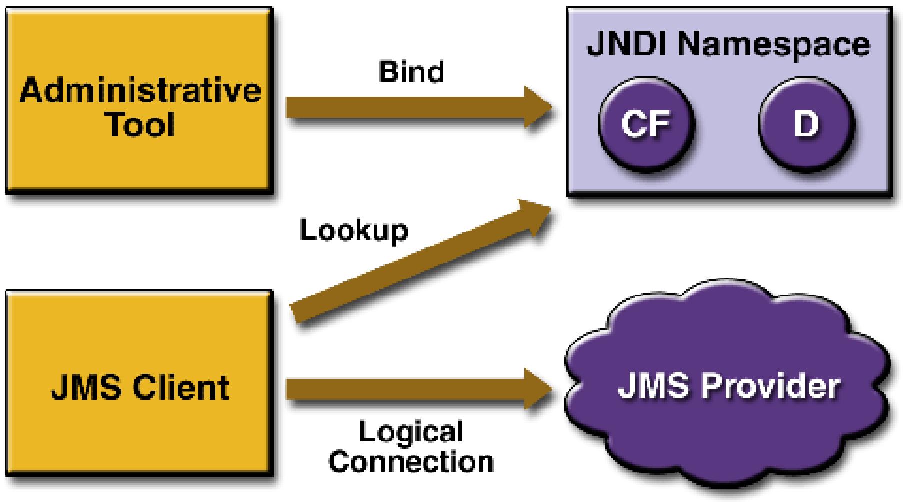


Figure 10: ***JMS API architecture***

4.2 Messaging Domains

4.2.1 Point-to-Point Messaging Domain

A point-to-point (PTP) product or application is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

PTP messaging has the following characteristics

- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

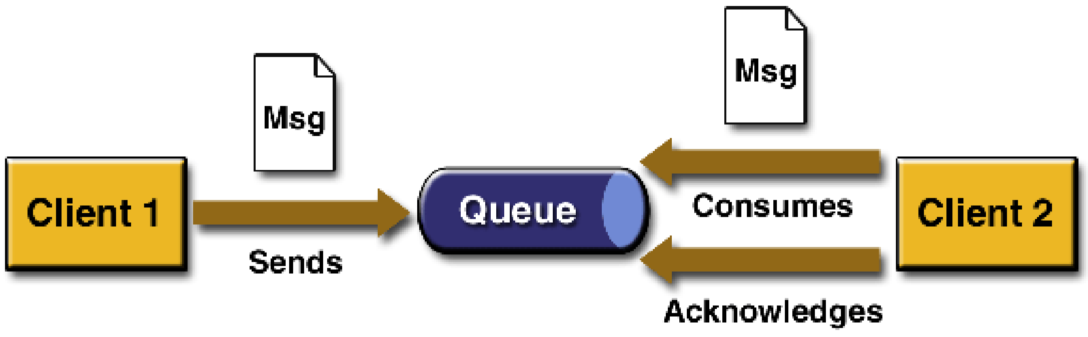


Figure 11: ***Point-to-Point messaging***

4.2.2 Publish/Subscribe Messaging Domain

In a publish/subscribe product or application, clients address messages to a topic. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.

Pub/sub messaging has the following characteristics.

- Each message may have multiple consumers.
- Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing clients to create *durable subscriptions*. Durable subscriptions can receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.

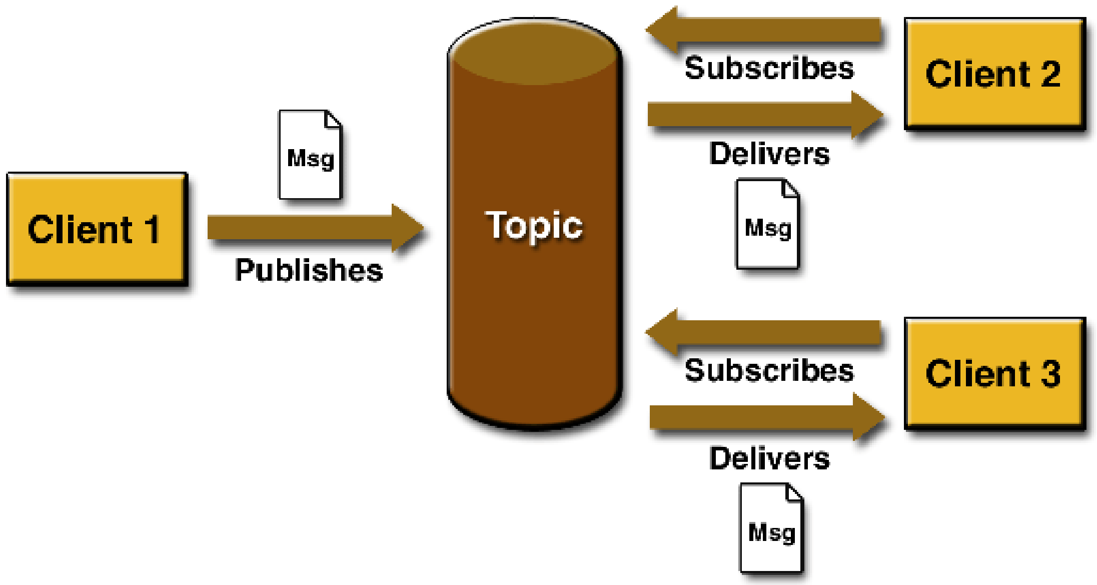


Figure 12: ***Publish/Subscribe Messaging Domain***

4.3 Message consumption

Messaging products are inherently asynchronous in that no fundamental timing dependency exists between the production and the consumption of a message. However, the JMS Specification uses this term in a more precise sense. Messages can be consumed in either of two ways:

- Synchronously**:** A subscriber or a receiver explicitly fetches the message from the destination by calling the *receive* method. The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- Asynchronously: A client can register a *message listener* with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's *onMessage* method, which acts on the contents of the message.

4.4 JMS API model

The basic building blocks of a JMS application consist of

- Administered objects
- Sessions
- Message producers
- Message consumers

- **Administered objects**

  It is expected that JMS providers will differ significantly in their underlying messaging technology. It is also expected there will be major differences in how a provider's system is installed and administered. If JMS clients are to be portable, they must be isolated from these proprietary aspects of a provider. This is done by defining JMS administered objects that are created and customized by a provider's administrator and later used by clients. The client uses them through JMS interfaces that are portable. The administrator creates them using provider-specific facilities.

  There are two types of JMS administered objects:

  - *ConnectionFactory* - This is the object a client uses to create a connection with a provider. This is denoted as CF in *figure 1*. The following code snippet shows how the connection is created:

    For pub/sub form:

    ```
    TopicConnectionFactory topicConnectionFactory =
    (TopicConnectionFactory) ctx.lookup("jndiName");

    TopicConnection topicConnection =
    topicConnectionFactory.createTopicConnection();
    ```

    For Point-To-Point form:

    ```
    QueueConnectionFactory queueConnectionFactory =
    (QueueConnectionFactory) ctx.lookup("jndiName");

    QueueConnection queueConnection =
     queueConnectionFactory.createQueueConnection();
    ```

- *Destination* - This is the object a client uses to specify the destination of messages it is sending and the source of messages it receives. This is denoted as D in *figure 1*. In the PTP messaging domain, destinations are called *queues.* In the pub/sub messaging domain, destinations are called *topics.* In addition to looking up a connection factory, a destination should also be looked up.

  For PTP form:

  ```
  Queue myQueue = (Queue) ctx.lookup("jndiName");
  ```

  For pub/sub form:

  ```
  Topic myTopic = (Topic) ctx.lookup("jndiName");
  ```

Administered objects are placed in a JNDI namespace by an administrator.

- **Sessions**

A *session* is a single-threaded context for producing and consuming messages. Sessions can be used to create message producers, message consumers, and messages. A session also provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

Sessions, like connections, come in two forms, implementing either the *QueueSession* or the *TopicSession* interface. Following code snippets show how a *Session* is created:

For PTP form:

```
QueueSession queueSession =
queueConnection.createQueueSession(true, 0);
```

For pub/sub form:

```
TopicSession topicSession =
topicConnection.createTopicSession(false, 0);
```

The first argument denotes the transactional; the second denotes whether the message should be acknowledged automatically or not and related issues.

- **Message producers**

A *message producer* is an object created by a session and is used for sending messages to a destination. The Point-To-Point (PTP) form of a message producer implements the *QueueSender* interface. The pub/sub form implements the *TopicPublisher* interface. *QueueSession* is used to create a sender for the queue, and *TopicSession* is used to create a publisher for the *Topic*:

```
QueueSender queueSender = queueSession.createSender(myQueue);

TopicPublisher topicPublisher =
topicSession.createPublisher(myTopic);
```

- **Message**

  The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

  A JMS message has three parts:

  - *Header* - A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages.
  - *Properties* – Properties can be set for the message in addition to the content. Properties can be used to provide compatibility with other messaging systems and it can also be used as filter parameters.
  - *Body* – The body is the payload or the content that should be communicated.

  Once a message is created with the description, the message is to be sent through the message producer which is shown as follows:

  ```
  queueSender.send(message);

  topicPublisher.publish(message);
  ```

- **Message consumer**

  A *message consumer* is an object created by a session and is used for receiving messages sent to a destination. A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination. The PTP form of message consumer implements the *QueueReceiver* interface. The pub/sub form implements the *TopicSubscriber* interface.

  ```
  QueueReceiver queueReceiver = queueSession.createReceiver(myQueue);

  TopicSubscriber topicSubscriber =
  topicSession.createSubscriber(myTopic);
  ```

The message could be received synchronously or asynchronously.

- Synchronous

  With either a *QueueReceiver* or a *TopicSubscriber*, *receive* method is used to consume a message synchronously.

  ```
  queueConnection.start();
  Message m = queueReceiver.receive();
  ```

- Asynchronous

  A *message listener* is an object that acts as an asynchronous event handler for messages. This object implements the *MessageListener* interface, which contains one method, *onMessage*. The message listener is registered with a specific *QueueReceiver* or *TopicSubscriber* by using the *setMessageListener* method. The following code demonstrates the registration of *JmsListener* class which implements *MessageListener*:

```
JmsListener jmsListener = new JmsListener();

topicSubscriber.setMessageListener(jmsListener);
```

And

```
queueReceiver. setMessageListener(jmsListener);
```

Once message delivery begins, the message consumer automatically calls the message listener's onMessage method whenever a message is delivered. The onMessage method takes one argument of type Message. A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on whether the listener is set by a *QueueReceiver* or a *TopicSubscriber* object. A message listener does, however, usually expect a specific message type and format. Moreover, if it needs to reply to messages, a message listener must either assume a particular destination type or obtain the destination type of the message and create a producer for that destination type.

4.4.1 Message selectors

Message selectors allow a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than to the application. A message selector is a *String* that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The *createReceiver*, *createSubscriber*, and *createDurableSubscriber* methods each have a form that allows specifying a message selector as an argument when a message consumer is created. The message consumer then receives only messages whose headers and properties match the selector. A message selector cannot select messages on the basis of the content of the message body.

4.5 Architecture of the Monitor

The main design goal of the architecture should be to provide maximum information without degrading the performance. The following points should be noted:

1. The logging part at the business process should be as simple as possible. This should not make the business process development complicated.
2. The logging service that includes the strategy to publish information should be hidden from the business process developer.
3. The logging service should not affect the performance of the business process.
4. The architecture should provide space for the administrator to stop the logging service or set filter to the logging service at the business process level to stop it from publishing the information.
5. The monitor should also have the provision to set filter and see in the information that the administrator wants to see.

The following figure details the strategy to publish the information:

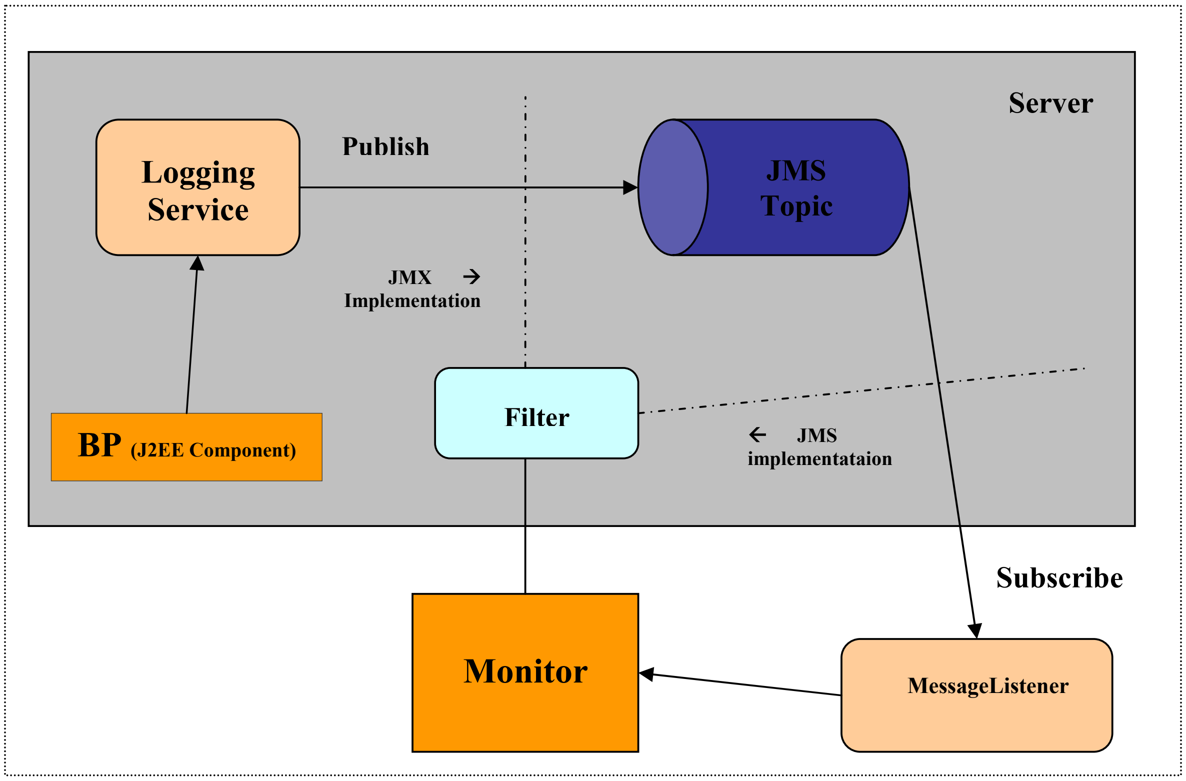


Figure 13: *Architecture of the monitor*

4.5.1 Logging service

The logging service is the liaison between the business process and the JMS queue/topic or any other means that may be used for publishing the information. The underlying technology is hidden from the business process. The business process, in our case the EJB, creates an instance of the logging service. For each bean, there should be only one instance of logging service. This insures that the bean does not make multiple connections to the JMS queue/topic. Establishing a connection to JMS is expensive and should be handled properly. Lazy initialisation of the logging service will make sure that there is only one instance of logging service, in particular one connection to the JMS queue/topic. The following code snippet shows how the logging service should be incorporated in an EJB:

```
protected LoggingService getLoggingService() throws Exception {
      if (loggingService == null) {
            loggingService = new
            LoggingService(getServiceLocator(),"SERVICE_NAME");
      }
      return loggingService;
}
```

The logging service adheres to the service locator pattern as described by Deepak Alur et al in Core J2EE Patterns.

J2EE clients interact with service components, such as EJB and JMS components, which provide business services and persistence capabilities. All J2EE application clients use JNDI common facility to look up and create JMS components. For JMS applications, the administered object can be a JMS *ConnectionFactory* for a Queue/Topic or a JMS Destination which could be a Queue/Topic. Locating a JNDI-administered service object is common to all clients that need to access that service object. That being the case, it is easy to see that many types of clients repeatedly use the JNDI service, and the JNDI code appears multiple times across these clients. This results in unnecessary duplication of code in the clients that need to look up services. Moreover, creating a JNDI initial context object and performing a lookup on a JMS *ConnectionFactory/Destination* object utilizes significant resources. If multiple clients repeatedly require the same object, such duplication effort can negatively impact application performance.

- **Service Locator for JMS Topic**

The service locator looks up the *TopicConnectionFactory* object using its JNDI name. The TopicConnectionFactory is cached by the service locator for future use. This avoids repeated JNDI calls to look it up when needed again. The client can then use it to create a *TopicConnection.* The class diagram for this strategy is shown in figure 14.

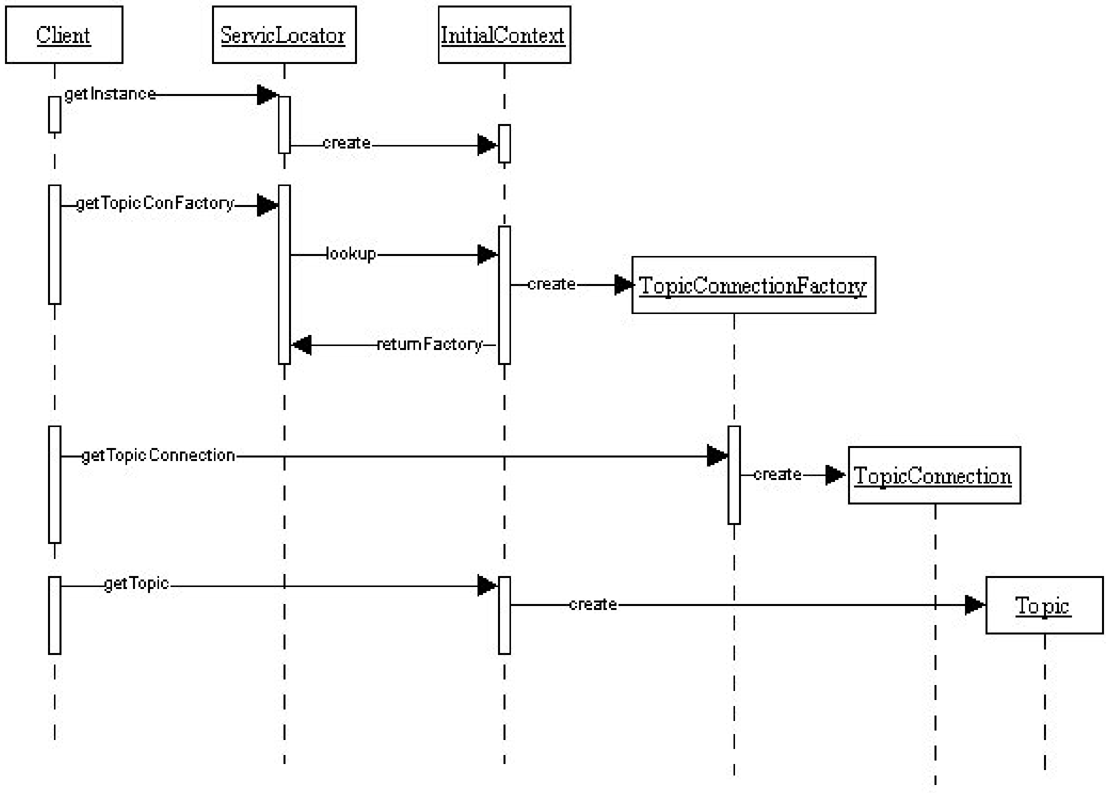


Figure 14: ***Sequence diagram for Service Locator for JMS implementation***

The logging service constructor will make the necessary JNDI lookups through the service locator pattern to get the *TopicConnectionFactory* and the *Topic* objects.

```
public LoggingService (ServiceLocator sl, String serviceName)
throws Exception {
      // necessary initialization
      topicConnectionFactory =
      serviceLocator.getEnvTopicConnectionFactory("NAME_TCF");
      topic = serviceLocator.getEnvTopic("NAME_TOPIC");
}
```

Now having got the instances that are necessary to initiate the connection and publish the information, the EJB component can publish the information through *log* method of *LoggingService*. The *log* method publishes the information on to the queue based on the filter parameter set both at the monitor end and the server end.

4.5.2 JMS Topic

As the business processes are real time applications, they should be monitored as and when there is a call to the business process. *Topic* is more suitable for this situation than a *Queue*. The *Queue* will hold the information until the point when the monitor consumes the information. This will be huge waste of resource considering the volume of clients that may be accessing the EJB that publishes the information. There may also be more than one monitor who might be interested in the functioning of the business process. Hence *Topic* will be more appropriate for this strategy.

4.5.3 Filter

The filter has to be designed both at the server end and at the client (monitor) end.

- **Server**

This is discussed in detail in the later chapter. The basic idea to have the filter to is to improve or not to hinder with the performance of the business process when the business process is not monitored. The filter is designed using Java Management Extension (JMX).

- **Client**

The client should have a filter option to better analyze the business process. There may be case where the administrator might be interested at the business process that throws Exception at the logic level or might be interested at one particular user who uses the business process. Considering the volume of information that can be retrieved from the *Topic*, which is proportional to the volume of clients that access the business process, the filter should be set at the *Topic* rather than at the client. This brings down the amount of communication traffic between the client and the *Topic*. The message selector, which is described earlier, is used for this strategy. The filter parameters are set at the *properties* of the *message*. Following snippet shows the filter parameters representing the *Level* of the business process, the name of the process and the user who has accessed the process and finally the payload which is the business object itself:

```
message.setIntProperty("level", record.getLevel());
message.setStringProperty("service", record.getServiceName());
message.setStringProperty("user", record.getUser());
message.setStringProperty("message", record.getMessage());
message.setObject(record);
```

At the monitor end, a query is built based on the needs of the administrator before retrieving the information. The following code snippets demonstrate the usage of the message selector:

```
String selector = "level = SEVERE and service = BankAccountEntity";
topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
subscriber = topicSession.createSubscriber(topic, selector, true);
```

The above piece of code broadcasts only the *BankAccountEntity* business record that has SEVERE *level* which is described by the *message* property to the subscriber.

4.5.4 Monitor

Monitor is the client who subscribes to the *Topic* to retrieve the information. *LoggingServiceConsumer* object is the mediator between the client and the *MessageListener*. This *LoggingServiceConsumer* also provides the provision to add as many *Handler*s as needed which can process/log the information from the *Topic*. The *Handler*s may use a rich Graphic User (GUI) interface or log the information to a file. With the ground structure of the monitor, new handlers can be just added to the *LoggingServiceConsumer* without changing the existing code. This is detailed in the sequence diagram as shown in the figure 15.

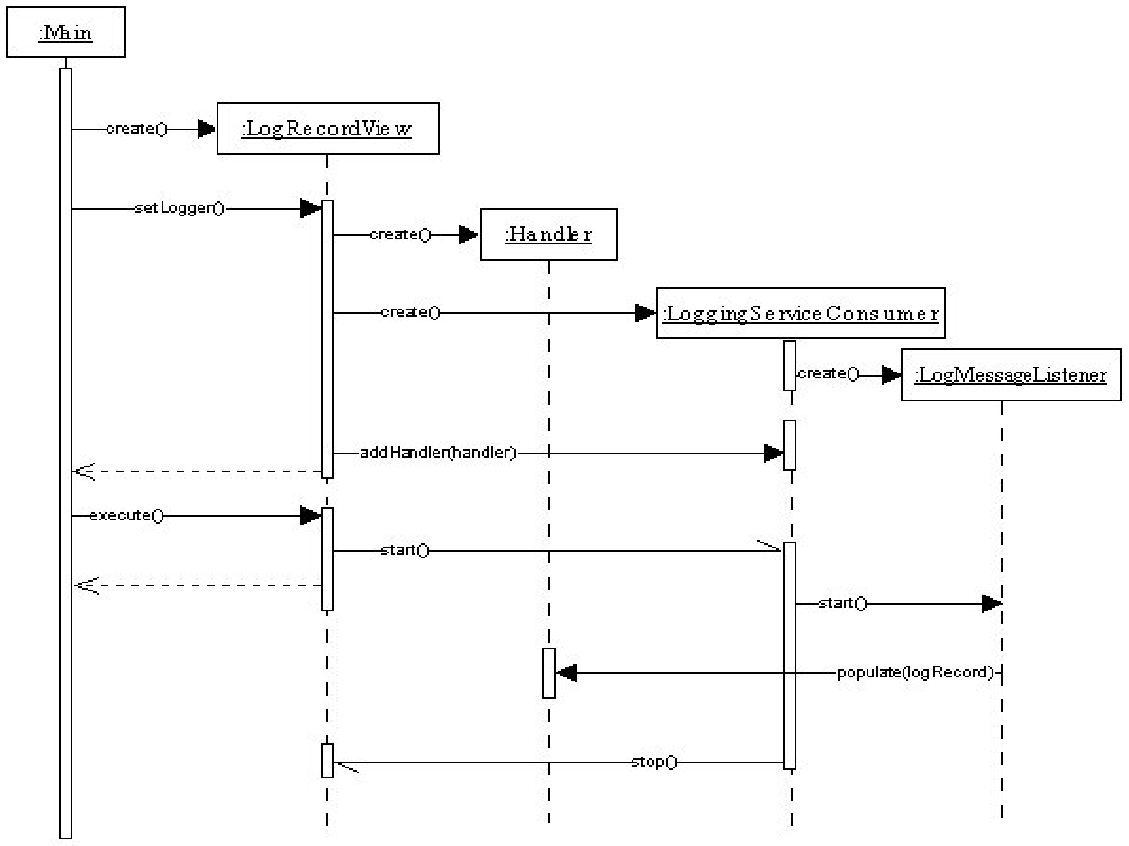
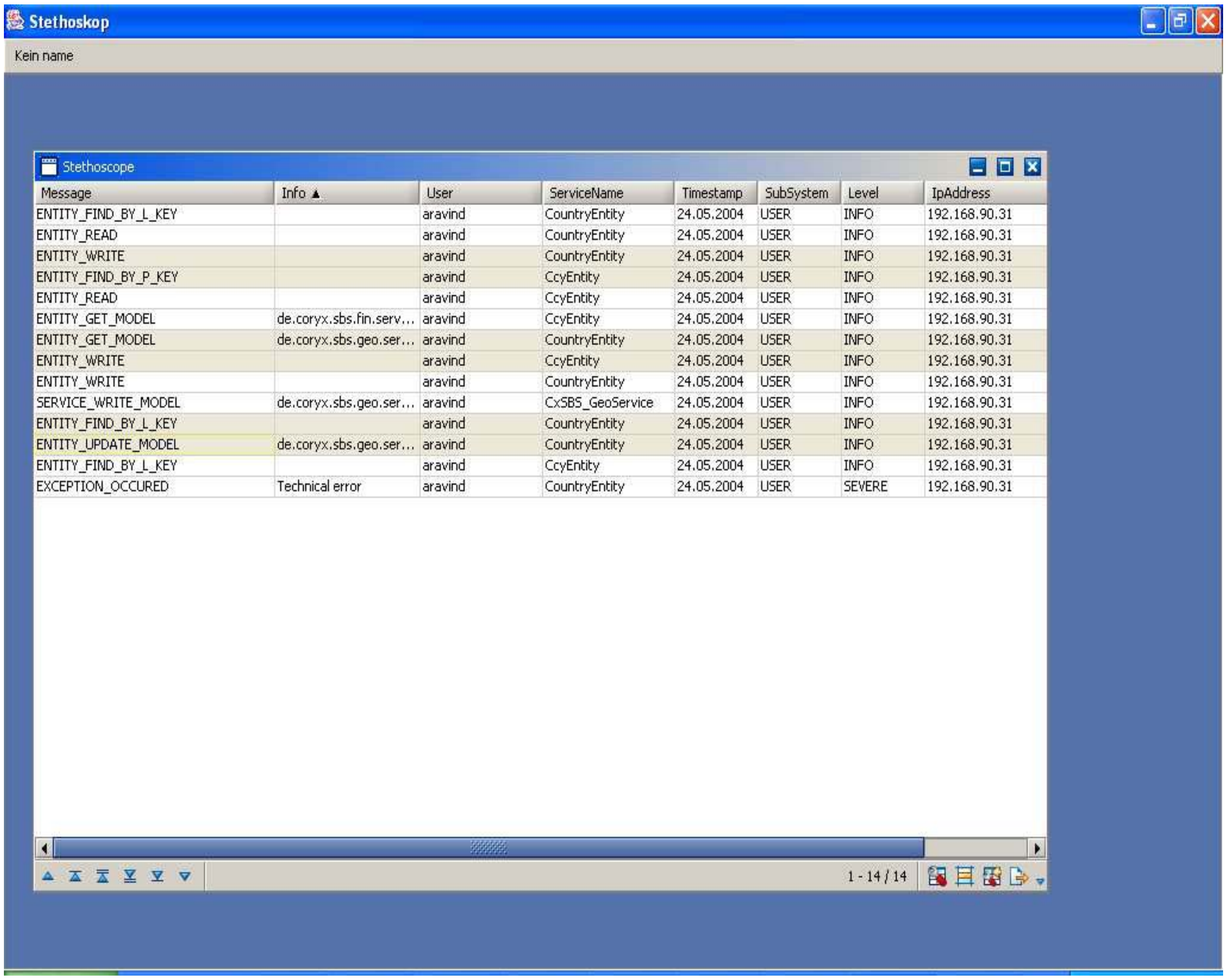


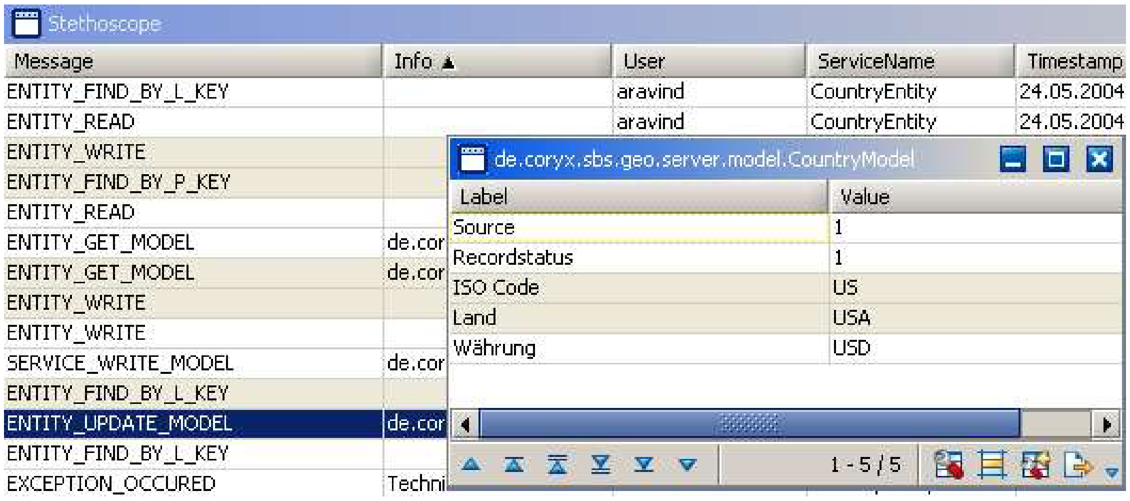
Figure 15: ***Sequence diagram of the monitor***

The figure 16 shows the information that is retrieved from the *Topic* with the *Handler* that displays the result in a rich GUI. The information is monitored for the service 'CountryEntity'. The message field shows the sequence of events that happen when the CountryEntity service is called. The last field shows an Exception which is of *Level* Severe. The business object of the particular service can also be viewed by double clicking on the service row which is shown in the figure 17.



Message	Info ▲	User	ServiceName	Timestamp	SubSystem	Level	IpAddress
ENTITY_FIND_BY_L_KEY		aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_READ		aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_WRITE		aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_FIND_BY_P_KEY		aravind	CcyEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_READ		aravind	CcyEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_GET_MODEL	de.coryx.sbs.fin.serv...	aravind	CcyEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_GET_MODEL	de.coryx.sbs.geo.ser...	aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_WRITE		aravind	CcyEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_WRITE		aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
SERVICE_WRITE_MODEL	de.coryx.sbs.geo.ser...	aravind	CxSBS_GeoService	24.05.2004	USER	INFO	192.168.90.31
ENTITY_FIND_BY_L_KEY		aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_UPDATE_MODEL	de.coryx.sbs.geo.ser...	aravind	CountryEntity	24.05.2004	USER	INFO	192.168.90.31
ENTITY_FIND_BY_L_KEY		aravind	CcyEntity	24.05.2004	USER	INFO	192.168.90.31
EXCEPTION_OCCURED	Technical error	aravind	CountryEntity	24.05.2004	USER	SEVERE	192.168.90.31

Figure 16: ***GUI of the Monitor***


Stethoscope

Message	Info ▲	User	ServiceName	Timestamp
ENTITY_FIND_BY_L_KEY		aravind	CountryEntity	24.05.2004
ENTITY_READ		aravind	CountryEntity	24.05.2004
ENTITY_WRITE				
ENTITY_FIND_BY_P_KEY				
ENTITY_READ				
ENTITY_GET_MODEL	de.cor			
ENTITY_GET_MODEL	de.cor			
ENTITY_WRITE				
ENTITY_WRITE				
SERVICE_WRITE_MODEL	de.cor			
ENTITY_FIND_BY_L_KEY				
ENTITY_UPDATE_MODEL	de.cor			
ENTITY_FIND_BY_L_KEY				
EXCEPTION_OCCURED	Techni			

de.coryx.sbs.geo.server.model.CountryModel

Label	Value
Source	1
Recordstatus	1
ISO Code	US
Land	USA
Währung	USD

1 - 5 / 5

Figure 17: ***Business Object view for the selected Service***

4.6 LoggingService incorporated in Business Process

The business process may contain one or many EJB or Servlet components. The components are made "manageable" by using simple calls to the LoggingService interface whose implementation is completely hidden from the BP. The following code demonstrates the simple EJB that is made manageable:

```
Public SampleBean extends EJBOject {

        //logging the information at the point when an instance of a
        //bean is created

        Public Object ejbCreate() {
        //just a string object is logged in this case. But user defined
        //information can also be logged
          getLoggingService().log("Instance Created");
        }

        Public void businessMethod() {
        //the name of the method and the timestamp is logged
              getLoggingService().log("Start    of       businessMethos",
        timestamp);
              //actual business logic starts
              …
              …
              //business logic ends
              //information is logged
              getLoggingService().log("end    of        businessMethos",
        timestamp);

        }

        //LoggingService is a singleton for a given instance of a Bean
        Public LoggingService getLoggingServce() {
              if (loggingService == null)
                    loggingService = new LoggingServiceImpl();
              return loggingService;
        }
}
```

5 Performance analysis

The management models implemented with JMX/JMS are in the production phase. The J2EE applications are internally implemented. The management implementation is a performance overhead when the application is not monitored. Hence there is a strong need to turn off the management process when not needed without halting any business process or redeployment of the business process. The management architecture should implement a filter which enables the administrator to turn on/off the monitoring process to improve the performance.

5.1 Filter options

The filter options can be set at a point that is common to both the management console and the management implementation. There are various options to implement the “common point”. But as this closely related to the business process itself, it is necessary to choose the best option that would perform better requiring lesser resource to manipulate.

5.1.1 Database

The easiest to implement the “turn off/on” mechanism is through a database. The database here is the only point where the administrator has control over the management process. The simple filter console can be incorporated in the management console. The filter parameters are set by the administrator and are read by the management process before logging the information from the business process. The database need not necessarily be in the same server. The database can be in a remote server. Though this is a very simple implementation, the amount of resource consumed is considerable. Establishing a connection to a database is quite expensive. Though this is a better option, but there are still simpler ways to implement the filter.

5.1.2 File system

A simple file system can effectively handle the situation without much effort. But there is lot of disadvantages in accessing a file system. The file in which the filter parameter to be set must be remotely available to the administrator. In a J2EE environment, the better way to access a file system is through a resource adapter which is Java 2 Connector Architecture (JCA) compliant. The business process can contain EJB components. The EJB specification forbids the use of I/O packages. The reasons for prohibiting the I/O packages to access file system from EJB are

- Access to file system is not transactional.
- Accessing file system is a potential security hole.

Hence both at the implementation level and at the performance level, this approach fails miserably.

5.2 JMX implementation of filter option

A simple architecture with Java Management Extension is devised to include the filter option in the management model. The main goals of this architecture:

- To enable easy access of the "common point" to both the management console and the management process.
- Consume less resource to get access to the "common point"

The architecture is detailed in the figure that follows.

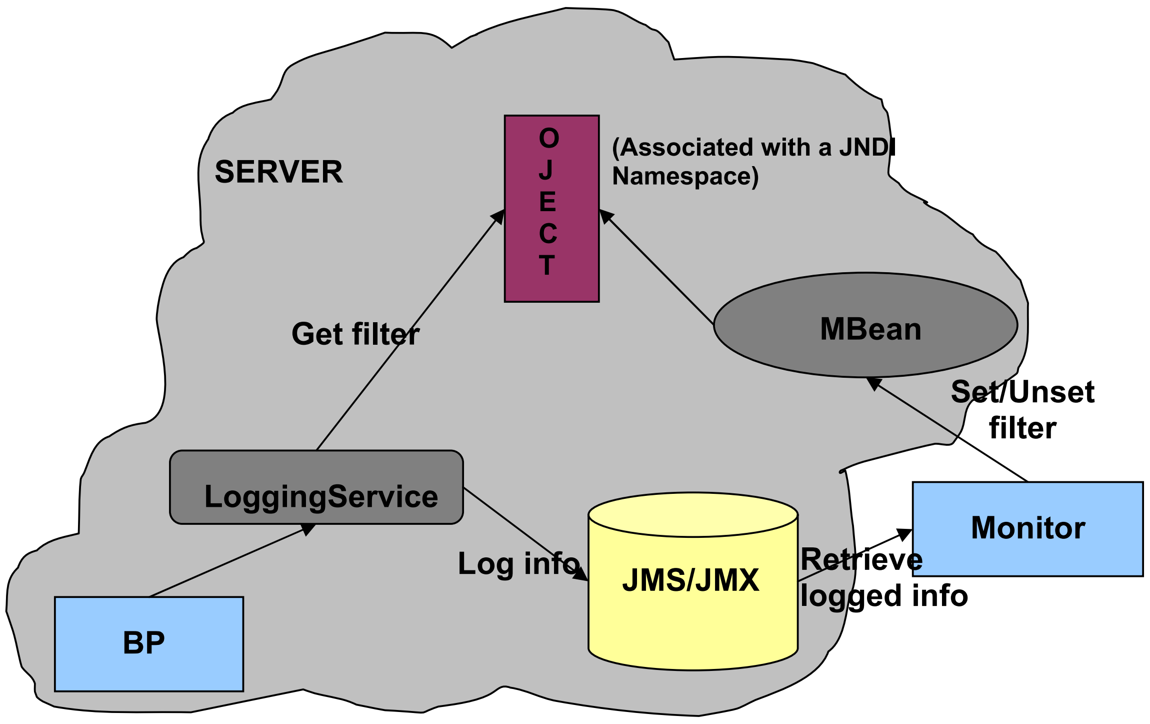


Figure 18: ***Architecture for implementing filter options.***

5.2.1 Filter repository

The shortcomings that are discussed earlier are overcome with a simple object that holds the filter data. The object may reside locally or remotely. The object is a simple *HashMap* that holds the filter parameters. This object made available both to the management console and to the business process itself through the *LoggingService* interface.

5.2.2 Persistence of the filter repository

JNDI, acronym for Java Naming and Directory Interface, is Sun's standard API for interfacing with directory and naming servers. JNDI's object-storage capabilities allow it to play the role of resource administrator in distributed applications and to provide simple, manageable object persistence.

According to the JNDI specification, service providers are encouraged, but not required, to support object storage in one of the following formats:

- **Serialized data**

  The most obvious approach to storing an object in a directory is to store the serialized representation of an object. The only requirement is that the object's class should implement the *Serializable* interface.

  Serialization is the technique by which an object can store and restore its state, usually to and from a stream of bytes. When an object is serialized, it is basically broken down into its most primitive values like integers, booleans and strings. These primitive values must be managed in a predetermined format and order so that the object can be *deserialize*d, thus restoring the object to its original state. Support for serialization allows an object to be *persistent.* In the simplest terms, this means that an object can survive from one program instance to another.

  Thus when an object is serialized, its state becomes transformed into a stream of bytes. The service provider takes the stream of bytes and stores it in the directory. When a client looks up the object, the service provider reconstructs it from the stored data.

- **Reference**

  Sometimes it is not appropriate to serialize an object. If the object provides a service on a network, for example, it does not make sense to store the state of the object itself. An example is a connection to an external resource (one outside the scope of the Java Virtual Machine) such as a database or file.

- **Attributes in a directory context**

  This approach provides directory functionality instead of only naming functionality, allowing storage of an object as a collection of attributes on a *DirContext* object. A *DirContext* instance differs from a *Context* instance in that it may have attributes. This approach is useful when the object must be accessible by non-Java applications.

In our case, first and the last option can be used. In case of the last approach, an object that implement the *DirContext* interface must be created and contain the code necessary to write their internal state as an Attributes object. An object factory to reconstitute the object must also be created. Though this approach also help to solve the issue, but the first approach is simple and clean with minimum complexity.

The Object that is associated with a namespace and stored in a JNDI tree must be changed by the management tool. The object is serialized and made accessible to the management console. When a filter parameter is to be changed, the object, in effect the *HashMap* is changed, the changes are to be made persistent. This requires the parameters of the object, which is associated with the same namespace, have to be changed to make the changes made by a management console available to other services that may access the filter parameters. New Object can be reassociated to the namespace. Manipulating a JNDI tree require administrative privileges. The management console need not necessarily have administrative privileges of an application server. Hence a solution has to be achieved to manage the filter parameters locally on an application server.

5.2.3 Manipulation of filter parameters through MBeans

The main functionality of MBean is to manage the Object that holds the filter parameters. As the filter parameters remain static for a given implementation, *StandardMBean* will be sufficient for exposing the management interface. The management interface will be simple get and set methods for each filter parameters. The MBean is shown in the code snippet that follows:

```
Public interface FilterMBean extends StandardMBean {

      //the getter and setters for the filter parameters

      public void setUser(String user);
      public String getUser();
      ..
      ..
}
```

All application servers start all the MBeans that are registered to the *MBeanServer* at the start up of the server. The list all the MBeans are usually obtained from a configuration file, which in most cases is XML file. For a given application server instance there will be only one instance of *MBeanServer*. The *MBeanServer* holds references to the instances of the MBeans that are registered to it. Hence for a given application server instance, there will be only one instance of the *MBeanServer* and MBeans. Hence the best design would require the MBean to get an instance of the serialized object that holds the filter parameters at the start up itself. Hence for a every get method, the JNDI tree is not looked up every time instead the parameter is returned from the local copy of the filter object.

The filter parameters can also be changed by the management console. In that case the new filter parameter is set to the filter object and rebound to the JNDI tree and the local copy is changed. Though it would be a valuable argument to question the use of the JNDI association, the main purpose of usage of the JNDI namespace is to make the application server remember the last changed value of the filter parameters even after a restart of the application server. The following code snippet explains more clearly:

```
Public Filter implements FilterMBean {

      //constructor which is called at the start up of the server

      Public FilterMBean() {
            //get the instance of the filter Object from the JNDI
            //name space
            context = new InitialContext();
            obj       =  context.lookup("NAMESPACE");
      }

      //for the filter parameter "User"

      public void setUser(String user) {
            obj.getHasMap().put("USER", user);
            context.rebound("NAMESAPCE", obj);
      }

      public String getUser() {
            //return the result from the local copy of the filter
            //object
            return (String) obj.get("USER");
      }

}
```

The *LoggingService* can get access to the filter parameters by directly accessing the JNDI tree. The *LoggingService* just reads the data and does not manipulate the data. The *LoggingService* constructor will now be as shown in the following code snippet:

```
public LoggingService (ServiceLocator sl, String serviceName)
throws Exception {

//get the filter parameters from the JNDI namespace
      filterObject = serviceLocator.getObject("NAMESPACE");

//check the filter parameters
      if(isLoggingEnabled(filterObject))

            // necessary initialization required for JMX/JMS
            //implementation
            logging = true;

      else

            //a flag is set false which will be checked for
            //each log call
            logging = false;

}
```

The log method that is called by the business methods to log information is logged based on a flag value set at the point of creation of an instance of the *LoggingService*. The log flag determines whether to log the information or not.

5.3 Comparison of JMX and JMS architectures

The architectures implemented with JMX and JMS technologies are suited much in the situations where one of the architectures fails, the other fills the void. But it is hard for any one implementation to work efficiently in all situations. Some of the situations are described as follows.

5.3.1 Performance

Performance here effective means the fastness or the responsive time for an execution. The performance is decided on the means which the *LoggingService* uses to accesses the resource to log the information.

- **JMS implementation**

  In the JMS implementation, the LoggingService has to get a reference to the instance of the *Topic* to publish the information. The JNDI look up is quite expensive if the *Topic* is located at a remote server. The *LoggingService* has to get a reference of the *ConnectionFactory* before getting a reference to the *Topic*.

- **JMX implementation**

  In the JMX implementation, the *LoggingService* has to get an instance of the *MBeanServer*. The *MBeanServer* instance that the *LoggingService* holds is actually a stub of the *MBeanServer*. Most of the application servers are compliant to the J2EE Management Model (JSR -77) standard, it is not even necessary to get a stub of the *MBeanServer*. The *MBeanServer* stays at the remote/local location, but a stateless session bean that interacts with the *MBeanServer* is exposed to the *LoggingService*. Hence the *LoggingService* gets the stub of the *Management* bean. This just requires single lookup in the JNDI tree and the Bean exposes the management interfaces that incorporate the notification model.

Though a single or double lookups do not make much difference at the performance, but this depends very much on how complicated the BP is. Hence JMX implementation is slightly better than the JMS implementation as for as performance is concerned.

5.3.2 Server Cluster

A server cluster consists of multiple copies of the server program running simultaneously and working together to provide increased scalability and reliability. A cluster appears to clients to be a single server instance. The server instances that constitute a cluster can run on the same machine, or be located on different machines. A cluster's capacity can be increased by adding additional server instances to the cluster on an existing machine, or by adding machines to the cluster to host the incremental server instances

A server cluster provides these benefits:

- Scalability

  The capacity of an application deployed on a server cluster can be increased dynamically to meet demand. Server instances can be added to a cluster without interruption of service—the application continues to run without impact to clients and end users.

- High-Availability

  In a server cluster, application processing can continue when a server instance fails. An application component can be clustered by deploying them on multiple server instances in the cluster—so, if a server instance on which a component is running fails, another server instance on which that component is deployed can continue application processing.

The choice to cluster server instances is transparent to application developers and clients. However, understanding the technical infrastructure that enables clustering will help programmers and administrators maximize the scalability and availability of their applications.

The key clustering capabilities that enable scalability and high availability:

- Failover

  Simply put, failover means that when an application component doing a particular "job"—some set of processing tasks—becomes unavailable for any reason, a copy of the failed object finishes the job.

  For the new object to be able to take over for the failed object:

  - There must be a copy of the failed object available to take over the job.
  - There must be information, available to other objects and the program that manages failover, defining the location and operational status of all objects—so that it can be determined that the first object failed before finishing its job.
  - There must be information, available to other objects and the program that manages failover, about the progress of jobs in process—so that an object taking over an interrupted job knows how much of the job was completed before the first object failed, for example, what data has been changed, and what steps in the process were completed.

Hence the state of the particular component that is processed at one server instance has to be replicated to other server nodes that are in the cluster.

- Load balancing

  Load balancing is the even distribution of jobs and associated communications across the computing and networking resources in your environment. For load balancing to occur:

- There must be multiple copies of an object that can do a particular job.
- Information about the location and operational status of all objects must be available.

Many application servers allow objects to be clustered—deployed on multiple server instances—so that there are alternative objects to do the same job. Server shares and maintains the availability and location of deployed objects using multicast, IP sockets, and JNDI.

In clustered environment, the components of a given application server are also clustered. The components are mostly EJBs or Servlets. Many application servers allow clustering of EJBs and Servlets.

In case of clustered environment, the management must also be compliant. Almost all application servers support or partially support clustering of JMS *Topic*. Both the cases fail-over and load balancing are supported by most application servers. But the components of JMX, mainly the *MBeanServer* and the *MBeans* are not cluster-wise visible. They are tied up to the single instance of the application server.

Hence in a single application server environment, the architecture based on JMX is preferred considering the performance supremacy over the JMS implementation. But in a clustered environment, the JMX implementation fails and the JMS implementation is well suited.

6 System-level Management

The system-level management means managing the system level resources that the business components use. The system-level resources include mainly the connection pools or the actual connection the business components make to the back end system which could be a database or an ERP system, the thread pools, the CPU utilization, etc. The business process performance depends very much on the resources. Hence there is a need to manage the resources to have an effective of the BP.

The J2EE based application servers have their own way of handling the resources. Hence the resource management is quite vendor-specific. Many application servers are based on JMX or have an implementation of JMX to expose the management interfaces of the system resources through MBeans. In effect, to manage the system resources means to understand the architecture of each application server and expose the management interface to the custom management console. Before discussing on how to manage system resources, the architecture of the application server has to be analyzed.

6.1 Architecture of JBoss Application Server

The JBoss server and container are completely implemented using component-based plug-ins. The modularization effort is supported by the use of JMX. Using JMX, industry-standard interfaces help manage both server components and the applications deployed on it. JMX provides a common spine that allows the user to integrate modules, containers and plug-ins. Figure 19 shows the role of JMX as an integration spine or bus into which components plug.

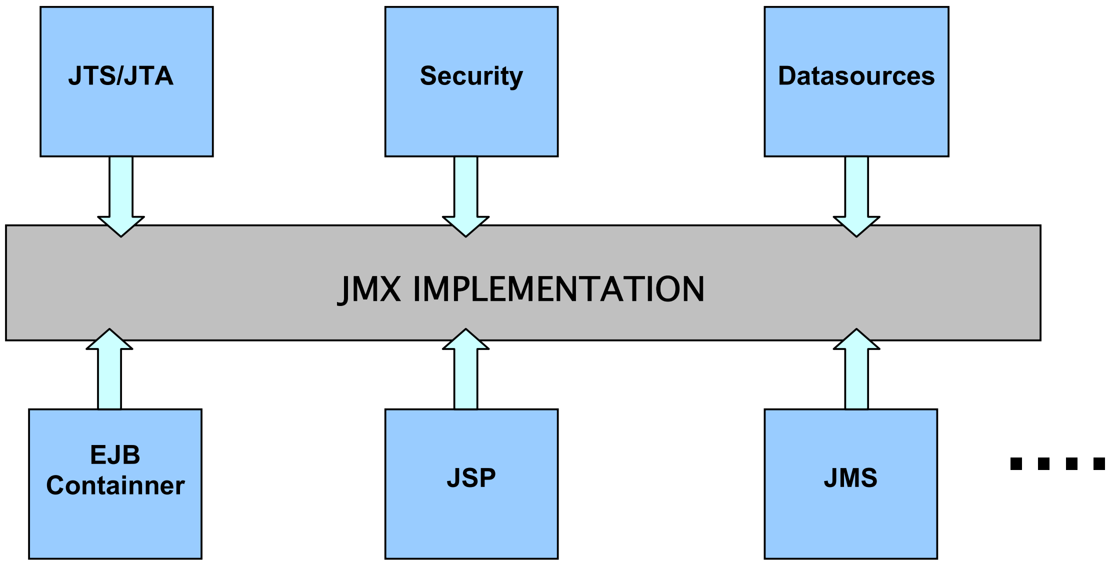


Figure 19: ***JBoss JMX integration bus and the standard JBoss components***

When JBoss starts up, one of the first steps performed is to create an MBean server instance. The JMX MBean server in the JBoss architecture plays the role of a microkernel aggregator component. All other manageable MBean components are plugged into JBoss by registering with the MBean server. The kernel in that sense is only an aggregator and not a source of actual functionality. The functionality is provided by MBeans, and infact all major JBoss components are manageable MBeans interconnected through MBean server.

6.2 J2EE server management with JSR 77

The J2EE specification should ease enterprise computing; we should be able to simply develop enterprise applications and deploy them into a J2EE-compliant product. But the reality is different because the J2EE specification does not go far enough. Many application server features are vendor specific, and, to avoid vendor lock-in, we need further standardization. One particular aspect of standardization is J2EE server management, an aspect covered by the J2EE Management Specification, Java Specification Request (JSR) 77. The J2EE Management Specification abstracts the manageable parts of the J2EE architecture and defines an interface for accessing management information. This helps in integrating J2EE server management with the custom management console.

JMX enables Bean developers to also provide a management interface, which the user can then incorporate into his management tool. What JMX does not do is to specify the meaning behind the management interface. Therefore, the client has to investigate the interface at runtime and then provide a management UI dynamically. JSR-77 now specifies the management interface; therefore, the management tool can be more specific for the J2EE environment. In addition, it defines how the information is made available to the remote client. The following description of the JSR-77 is based on the J2EE Management specification of Sun Microsystems, Inc.

6.2.1 Differences between JMX and JSR 77

Note that JSR-77 does not provide Java classes for its implementations. It does provide a model and a meta-model of how the data is presented to the client. The model describes how the data is grouped together into *objects* and how the objects are related to each other (inclusive cardinalities).

Since JSR-77 does not provide classes, the term object should not be associated with Java instances. Here, it just means a group of information representing a logical object of the J2EE server, such as a Web module, a JNDI service, a computer, and so forth. Because of the general nature of JSR-77, it does not have to represent physical objects when the application server is built differently.

Also noteworthy is the fact the JSR-77 does not have a notion of an agent. Each J2EE server provides one or more points of access for remote clients. The Management EJB is required but JSR-77 also contains specifications for the SNMP and CIM protocols.

6.3 J2EE Management Model

A J2EE server is a concrete system, and JSR 77 defines a concrete object model. This J2EE management model contains managed objects models for all well-known concepts from the J2EE world, such as a Java Virtual Machine (JVM), EJB and EJB module. The term managed object refers to the definition of a unit of management information. Management instrumentation provides the "glue" which takes the information available on an entity to be managed and makes it appear as a collection of managed objects.

6.3.1 Managed objects

All managed objects derive common features from a base model called *J2EEManagedObject*, shown in figure 20. All managed objects in the J2EE platform must include the attributes of the *J2EEManagedObject* model. All managed objects must have a unique name.

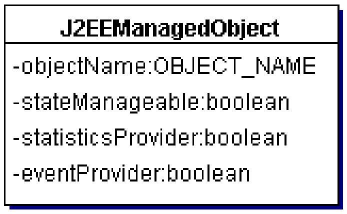


Figure 20: ***J2EEManagedObject***

An important attribute of J2EEManagedObject is the *objectName* of type OBJECT_NAME; an *objectName* identifies an object. Here's an example of an object name's string presentation:

```
jboss.management.single:name=localhost,j2eeType=JVM,J2EEServer=Single
```

This object name is from JBoss. The string to the left of the colon (:) is the domain name, a string used as a namespace. Three key attributes follow the colon:

- name, the object's actual name
- j2eeType, the managed object's concrete type name (in the example above, a JVM with the name localhost)
- The relationship to a parent—the parent is the J2EEServer with the name Single

Other attributes for the *J2EEManagedObject*: *stateManageable*, *statisticsProvider*, and *eventProvider*. All these attributes are of type Boolean and indicate the existence of other features. If *stateManageable* is true, the object provides additional operations to start and stop its services. If *statisticsProvider* is true, an object can provide runtime statistics. If *eventProvider* is true, an event provider object enables a client to register for events and receive event notifications. Specialized models derive from the *J2EEManagedObject* model. Though figure 21 shows only a part of the hierarchy, it includes some well-known J2EE concepts.

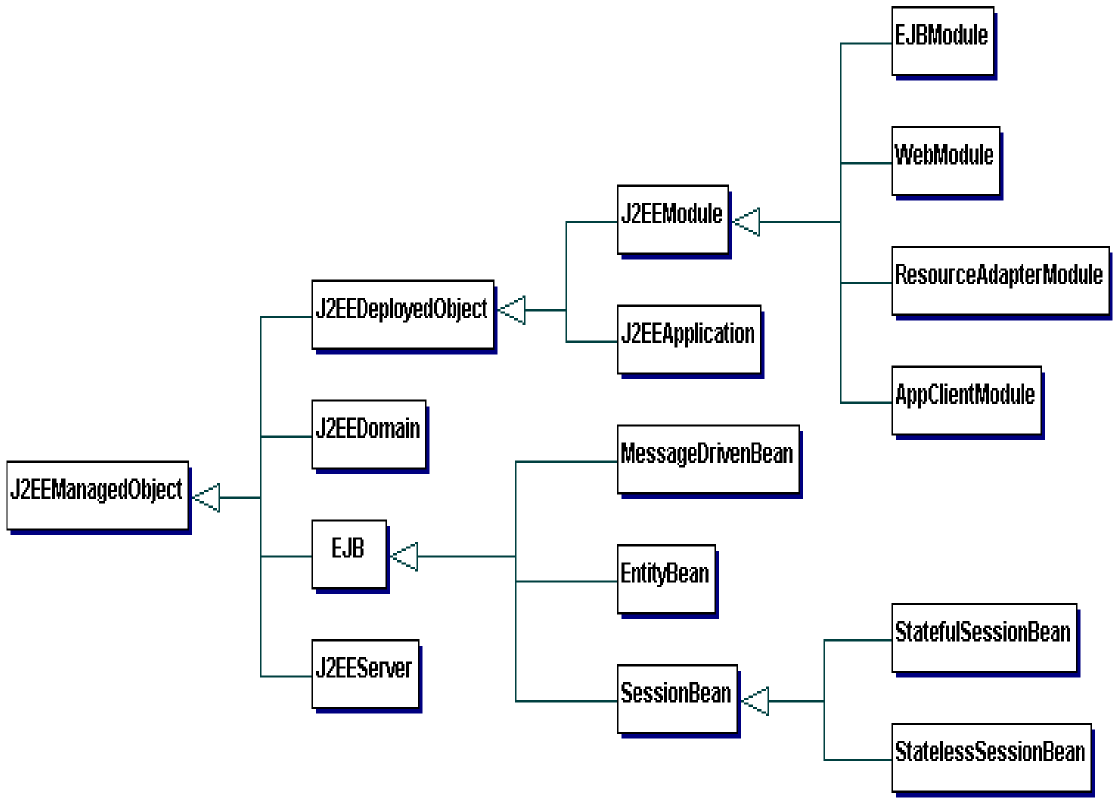


Figure 21: ***JSR-77 objects overview***

6.3.2 Events

Just to prevent some confusion: events here are similar in concept to the JMX's notifications concept. Any server-side event provider can send the client a message. This prevents the client from polling the server to see if any events are in the queue, and the client can inform the users immediately when the event is received.

Basically any Managed Object can send an event by indicating that it is an event provider. Then the client can register a listener for this Managed Object and receive an event when the object sends one. Currently, it is up to the implementation of the server-side to define how the events are transferred. It can be RMI or JMS, because they come with the J2EE server.

When an object indicates that it is an event provider, it implements the *EventProvider* object, containing a list of types of events it emits. As a reminder, "implements" here means not the implementation of an interface, but merely the providing of another attribute.

6.3.3 State Management

State management refers to the management facilities that are provided by compliant J2EE platforms to manage the state of a J2EE platform and the components that comprise it. The management facilities allow Management Applications to get the current state of the platform and its components, find out how long the platform and components have been running, and start and stop the platform components. The StateManageable model specifies the operations and attributes that must be implemented by a managed object that supports state management. A managed object that implements the StateManageable model is termed a State Manageable Object (SMO). An SMO generates events when its state changes.

When an object indicates that it can manage a state, it implements the *StateManageable* object containing an integer indicating the state and the timestamp for when the object was last started. In addition, it implements the *start()*, *startRecursive()*, and *stop()* methods. This *start()* method starts the object; the *startRecursive()* also starts all state-manageable children of this object after the object is started; and the *stop()* method first stops the object and then all of the state-manageable children; therefore, we have no *stopRecursive()* method.

6.3.4 Performance Monitoring

The Performance Data Framework specifies a performance data model as well as performance data requirements of the J2EE Management Model. The Performance Data Framework consists of the *StatisticsProvider* model, which any managed object may implement, the *Stats* interfaces, which specify standard performance attribute semantics for each managed object type, and the *Statistic* interfaces which provide specific interfaces for representing the common performance data types.

The *StatisticsProvider* model must be implemented by all managed objects that provide performance data. A managed object that implements the *StatisticsProvider* model must have its *statisticProvider* attribute set to "true". The stats attribute references the specific *Stats* interface that corresponds to the managed object type that is providing *Statistics*. For example, an *EntityBean* managed object that implements *StatisticsProvider* will have a reference in the stats attribute to an object that implements the *EntityBeanStats* interface. The detail for the stats attribute includes a table of the appropriate *Stats* interface that each managed object must implement if it provides performance data.

6.4 J2EE Management EJB Component

The J2EE Management EJB component (MEJB) provides interoperable access to the J2EE Management Model from any J2EE component on all platforms that implement the J2EE Management specification. The MEJB component incorporates the Java Management Extensions (JMX) API, a standard framework for Java object instrumentation. The MEJB component exposes the managed objects on any J2EE platform as JMX manageable resources. The MEJB component provides local and remote access of the platform’s manageable resources through the EJB interoperability protocol.

All compliant J2EE products must provide an implementation of an Enterprise Session bean component which implements the interfaces specified so far. The MEJB component may be automatically deployed during server installation. A compliant J2EE product must deploy an MEJB component before installation of that product can be considered complete. The MEJB component provides access to the managed object instances of all the available managed objects in one or more management domains. Compliant implementations of the MEJB component must provide access to all managed object instances required by the J2EE Management Model. All attributes required by the J2EE Management Model for a standard J2EE managed object type must be accessible from the MEJB component. All operations required by the J2EE Management Model for a standard J2EE managed object type must be able to be invoked from the MEJB component.

This MEJB is a stateless session bean, which is available under the JNDI name *ejb/mgmt/MEJB*. The following code snippet obtains the MEJB:

```
Context ctx = new InitialContext();
Object objref = ctx.lookup("ejb/mgmt/MEJB");
ManagementHome home = (ManagementHome)
    PortableRemoteObject.narrow(objref,ManagementHome.class);
Management mejb = home.create();
```

The MEJB's interface is very simple. It provides methods to query objects, to get all metadata for an object, to get and to change attributes, and more. Figure 22 shows the MEJB remote interface.

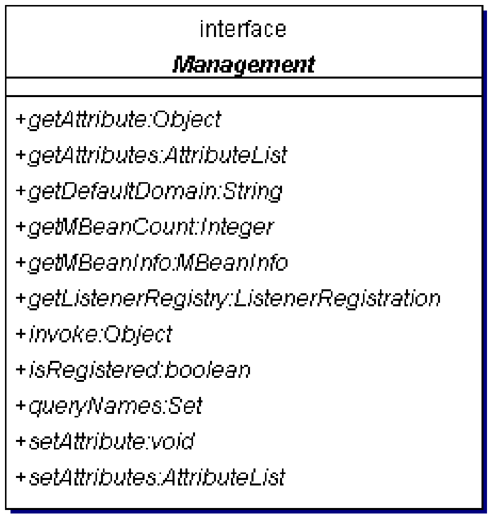


Figure 22: ***MEJB remote interface***

To find a managed object, the *queryNames()* method must be used, which returns a *java.util.Set* containing object names that identify the objects matching the query parameters. The class *ObjectName* comes from JMX. *queryNames()*'s first parameter is an *ObjectName* that specifies a query string. A second parameter is a *QueryExp*, which is used for more specific searches. For simple queries, the second parameter can be null. The following code uses *queryNames()*:

```
Set names = mejb.queryNames(new ObjectName("domain :j2eeType=EJBModule,*"),
  null);
```

The attributes provide the information of the managed objects. The values of the attribute can be read as shown in the following code snippet:

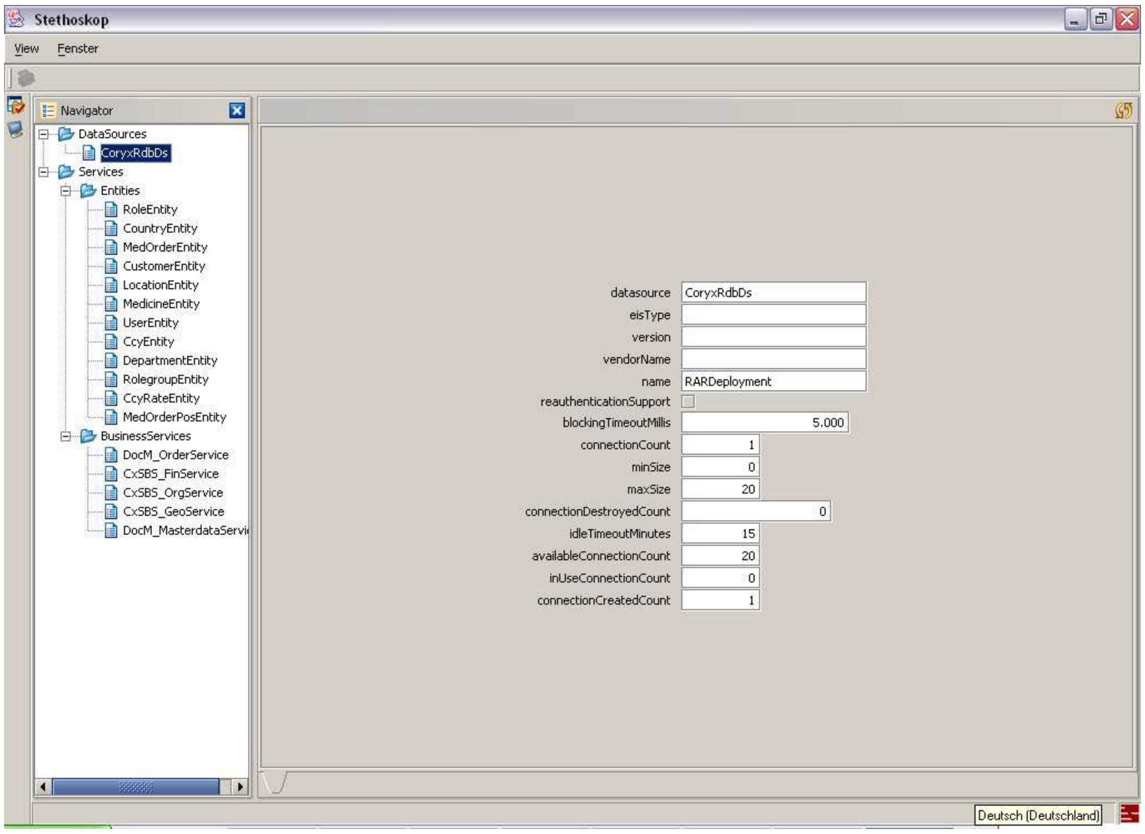
```
Iterator itr = names.iterator();
while(itr.hasNext()) {
  ObjectName name = (ObjectName)itr.next();
  ObjectName[] ejbs = (ObjectName[])mejb.getAttribute(name, "ejbs");
}
```

Operation of a managed object can be invoked through the methods provided by the Management interface. The following code demonstrates how to invoke an operation of a managed object.

```
mejb.invoke(name, “test”, new Object [] {}, new String [] {});
```

name is the *ObjectName* of a Managed Object and *“test”* is the name of the operation of the Managed Object that should be invoked. The other two parameters are the arguments of the operation and their signatures. The *invoke* operation returns the return value from the operation that is being invoked.

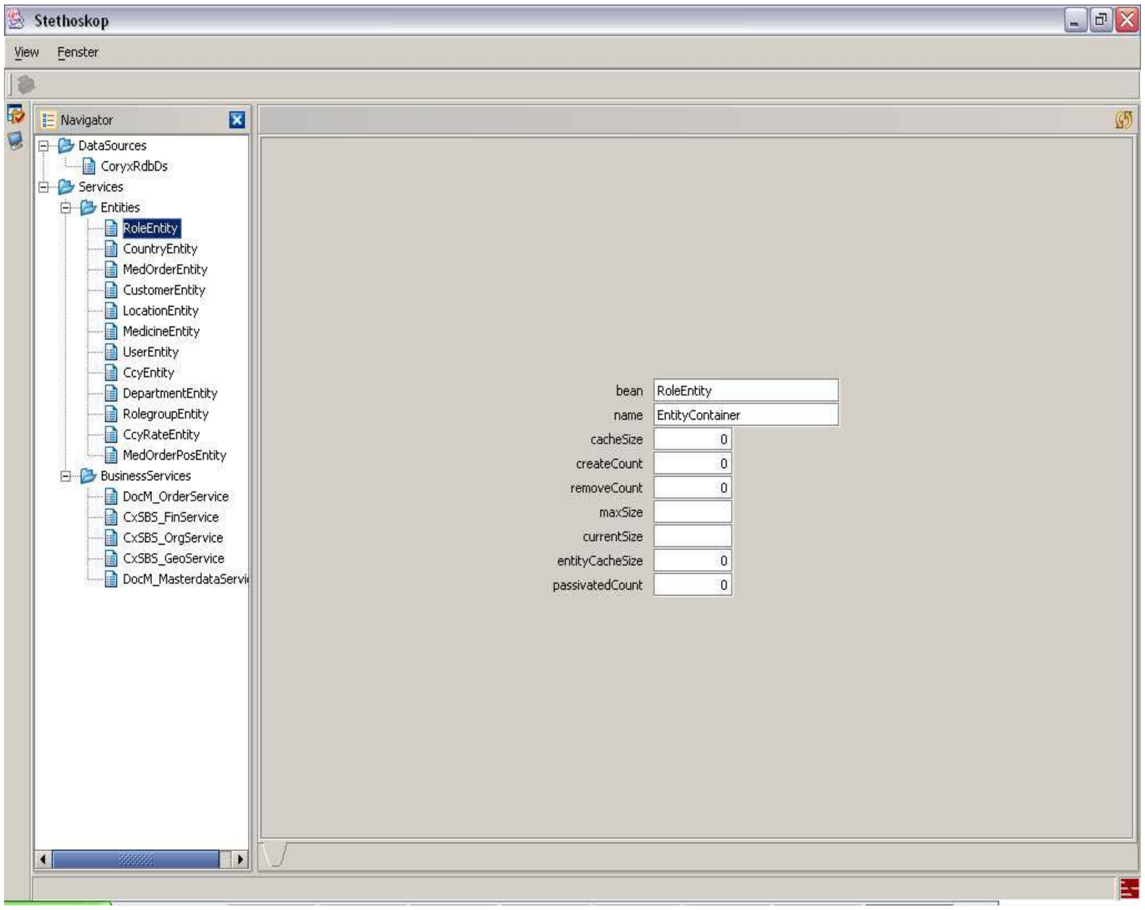
The system-level management can be achieved by managing the resources that are used by the enterprise applications. The enterprise applications, in most cases, are the EJBs. The cache utilization of the Entity bean can also be monitored. With the help of the JSR-77 specification, the MBeans that are specific to the application servers are exposed. The figure 23 and figure 24 shows screen shots of data sources and services (EJBs) management respectively.



- DataSources
  - CoryxRdbDs
- Services
  - Entities
    - RoleEntity
    - CountryEntity
    - MedOrderEntity
    - CustomerEntity
    - LocationEntity
    - MedicineEntity
    - UserEntity
    - CcyEntity
    - DepartmentEntity
    - RolegroupEntity
    - CcyRateEntity
    - MedOrderPosEntity
  - BusinessServices
    - DocM_OrderService
    - CxSBS_FinService
    - CxSBS_OrgService
    - CxSBS_GeoService
    - DocM_MasterdataServi

datasource	CoryxRdbDs
eisType	
version	
vendorName	
name	RARDeployment
reauthenticationSupport	☐
blockingTimeoutMillis	5.000
connectionCount	1
minSize	0
maxSize	20
connectionDestroyedCount	0
idleTimeoutMinutes	15
availableConnectionCount	20
inUseConnectionCount	0
connectionCreatedCount	1

Figure 23: ***Management of Datasources***



bean	RoleEntity
name	EntityContainer
cacheSize	0
createCount	0
removeCount	0
maxSize	
currentSize	
entityCacheSize	0
passivatedCount	0

Figure 24: ***Management of the deployed services***

7 Management of Tomcat

Tomcat is the Reference Implementation (RI) for the Java Servlet and JavaServer Pages (JSP) technologies. It is the official reference implementation for these complementary technologies. Tomcat is a servlet container with a JSP environment. A servlet container is a runtime shell that manages and invokes servlets on behalf of users.

7.1 JMX and Tomcat

Tomcat has an implementation of JMX for instrumentation of the resources and the applications deployed in an instance of Tomcat. Tomcat 4.1 binary release comes bundled with the open source MX4J version 1.1.1 implementation of the JMX 1.1 Specification. The Tomcat 5.0 binary release comes bundled with Sun's JMX 1.2 reference implementation, along with Sun's JMX Remote API 1.0 reference implementation. Both MX4J and JMX Remote API address the connectivity to a remote JMX Agent and expose the MBeans that manage the resources. Tomcat's JMX implementation is based on Model MBeans. The *Modeler* component of the Jakarta Commons subproject offers convenient support for configuring and instantiating Model MBeans (management beans), as described in the JMX Specification.

7.2 Modeler component

Modeler component is typically used within a server-based application that wants to expose management features via JMX. Model MBeans are very powerful - and the JMX specification includes a mechanism to use a standard JMX-provided base class to satisfy many of the requirements, without having to create custom Model MBean implementation classes manually. However, one of the requirements in creating such a Model MBean is to create the corresponding metadata information (i.e. an implementation of the *javax.management.modelmbean.ModelMBeanInfo* interface and its corresponding subordinate interfaces). Creating this information can be tedious and error prone. The *Modeler* package makes the process much simpler, because the required information is constructed dynamically from an easy-to-understand XML description of the metadata. Once the metadata is defined, and registered at runtime in the provided *Registry*, *Modeler* also supports convenient factory methods to instantiate new Model MBean instance. Registry is the registry for modeler MBeans. This is the main entry point into modeler. It provides methods to create and manipulate model MBeans and simplify their use.

Model MBeans effectively manage the resources in Tomcat. Having discussed the Modeler package which in creating the *ModelMBean*, it now easy to add a feature to the Tomcat and expose the management interface in the standard JMX way. One of the features that is added to the Tomcat is a way to remotely make the log information available to the management console. The web modules decide the location and name of the log file in which the log information is to be stored. The Tomcat exposes the file name and the location but not the content of the log file.

7.3 Custom service in Tomcat

Modeler package helps in adding a custom service to Tomcat. *Modeler* requires a configuration file that describes the metadata ultimately need to construct the *javax.management.modelmbean.ModelMBeanInfo* structure that is required by JMX. The configuration file is an XML file which should conform to the mbeans-descriptors.dtd DTD that defines the acceptable structure.

Fundamentally, <*mbean*> element is to be constructed for each type of Model MBean that a registry will know how to create. Nested within this element will be other elements describing the constructors, attributes, operations, and notifications associated with this MBean. The mbeans-descriptor file for the log information MBean is as shown below:

```
<mbeans-descriptors>
 <mbean name="LogService"
        description="Service to read logged info"
        domain="LogService"
        type="de.coryx.apps.crc.server.LogService">

    <operation name="openLogFile"
               description="Open logfile and read the content"
               impact="ACTION"
               returnType="void">
      <parameter name="directory"
                 description="Directory of the logfile"
                 type="java.lang.String"/>
      <parameter name="fileName"
                 description="Name of the logfile"
                 type="java.lang.String"/>
    </operation>

    <attribute name="logDetails"
               description="logged info"
               type="java.lang.String"
               writeable="false"/>
 </mbean>
</mbeans-descriptors>
```

This MBean represents an instance of *de.coryx.apps.crc.server.LogService*, which is an entity representing the logged information which is read from the log file for a given web module. This MBean advertises support for only one attribute, which is the logged information "loDetails" itself, that roughly correspond to JavaBean properties. By default, attributes are assumed to have read/write access. For this particular MBean, the logDetails attribute is read-only (writeable="false"). Finally, this MBean supports only one operation (*openLogFile*), which accepts two parameters: *directory* and *fileName*, which roughly correspond to JavaBean methods on the underlying component. The operation, *openLogFile,* returns void.

In general, *Modeler* provides a standard *ModelMBean* implementation that simply passes on JMX calls on attributes and operations directly through to the managed component that the *ModelMBean* is associated with. For special case requirements, a subclass of *BaseModelMBean* can be defined that provides override methods for one or more of these attributes (i.e. the property getter and/or setter methods) and operations (i.e. direct method calls).

The *BaseModelMBean* is the basic implementation of the *ModelMBean* interface, which supports the minimal requirements of the interface contract. This can be used directly to wrap an existing java bean, or anywhere an MBean would be used. The *String* parameter passed to the constructor will be used to construct an instance of the real object that is wrapped.

The metadata information, and the corresponding Model MBean factory, is represented at runtime in an instance of *Registry* whose contents are initialized from the configuration file prepared as was described above. Typically, such a file will be included in the JAR file containing the MBean implementation classes themselves, and loaded as follows:

```
URL url= this.getClass().getResource
             ("/de/ceryx/apps/crc/server/mbeans-descriptors.xml");
Registry registry = Registry.getRegistry();
registry.loadMetadata(url);
```

Besides using the configuration file, it is possible to configure the registry metadata by hand, using the *addManagedBean()* and *removeManagedBean()* methods. However, this standard support for loading a configuration file is convenient and sufficient. *Modeler* will also look for an *mbeans-descriptors.xml* file in the same package with the class being registered and in its parent. If no metadata is found, modeler will use a number of simple patterns to determine a reasonable metadata.

```
LogService service = new LogService(); //managed component instance
MBeanServer mserver = registry.getMBeanServer();
String oname = "LogService:name=LogService";
registry.registerComponent( service, oname,
service.getClass().getName());
```

The *registerComponent* method of the *Registry* instance takes three arguments: instance of managed bean, the *ObjectName* of the MBean and the type of the MBean. The service can be made to run at the start up of the server by adding a *Listener* to the server.xml file of the server. The Listener controls the life of the MBean. The above snippet of code can be used at the start up of the server.

7.4 Managing and Monitoring Tomcat

There are many MBeans that are exposed by the Tomcat that does the task of managing and monitoring the server instance, the applications that are deployed and the resources. The Tomcat adheres to the JMX specification and hence the management interfaces can be acquired as described in the earlier chapters. The remote client could get access to the management interface through Java Specification Request (JSR) 160.

The Java Management Extension (JMX) API is defined and under maintenance release of the Java Specification Request (JSR) number 3. JMX defines the API for management of Java applications, and those API are local to the application. This means JMX specification does not provide a solution to access the management interfaces remotely. To fill this gap, JSR 160 extends JSR 3 by providing a standard API to connect to remote JMX-enabled applications. Currently, JSR 160 has defined a mandatory connector based on RMI (that supports both RMI/JRMP and RMI/IIOP), and an optional one based on sockets and Java serialization (JMXMP). JSR 160 thus provides a standard way to connect to remote JMX-enabled applications using RMI.

7.4.1 Java Specification Request 160

Java Specification Request (JSR) 3 defines the JMX specification. What is standardized by JSR 3 is the way in which resources are instrumented within a management agent based on Java technology, and a certain number of agent-local services based on that instrumentation. Although JSR 3 defines terminology for remote access to instrumentation, it does not standardize any particular remote access API or protocol. Many solutions exist for exporting JMX API instrumentation either through existing management protocols such as the simple network management protocol (SNMP) or through proprietary protocols. This JSR (JSR 160) standardizes one such solution.

7.4.2 Connectors

The JMX specification defines the notion of connectors. A connector is attached to a JMX API MBean server and makes it accessible to remote Java technology-based clients. The client end of a connector exports essentially the same interface as the MBean server. A connector consists of a connector client and a connector server. A connector server is attached to an MBean server and listens for connection requests from clients. A connector client takes care of finding the server and establishing a connection with it. A connector client will usually be in a different Java Virtual Machine from the connector server, and will often be running on a different machine.

Many different implementations of connectors are possible. In particular, there are many possibilities for the protocol used to communicate over a connection between client and server. This standard defines a standard protocol based on Remote Method Invocation (RMI) that must be supported by every conformant implementation. It also defines an optional protocol based directly on TCP sockets, called the JMX Messaging Protocol (JMXMP). An implementation of this standard can omit the JMXMP connector.

7.4.3 Connection Establishment

A connector client is represented by an object that implements the *JMXConnector* interface. If the client has the address (*JMXServiceURL*) of the connector server to which it wants to connect, it can use the *JMXConnectorFactory* to make the connection. A *JMXServiceURL* is a string of the form:

```
service:jmx: <protocol>://[[[ <host>]: <port>]/ <path>]
```

where protocol is a short string that represent the protocol such as "rmi", "iiop", "jmxmp" or "soap", while host, port and path are optional. A *JMXServiceURL* can be seen as the "address" of a *JMXConnectorServer*, and it is the mean by which a *JMXConnector* can connect to a *JMXConnectorServer.*

For example, an application *app1* that includes an MBean server might export that server to remote managers as follows:

1. Create a connector server *cServer*
2. Get *cServer's* address *addr*, either by using the *JMXServiceURL* that was supplied to its constructor to tell it what address to use, or by calling *cServer.getAddress()*
3. Put the address somewhere the management applications can find it, for example in a directory or in an SLP service agent

A manager can start managing app1 as follows:

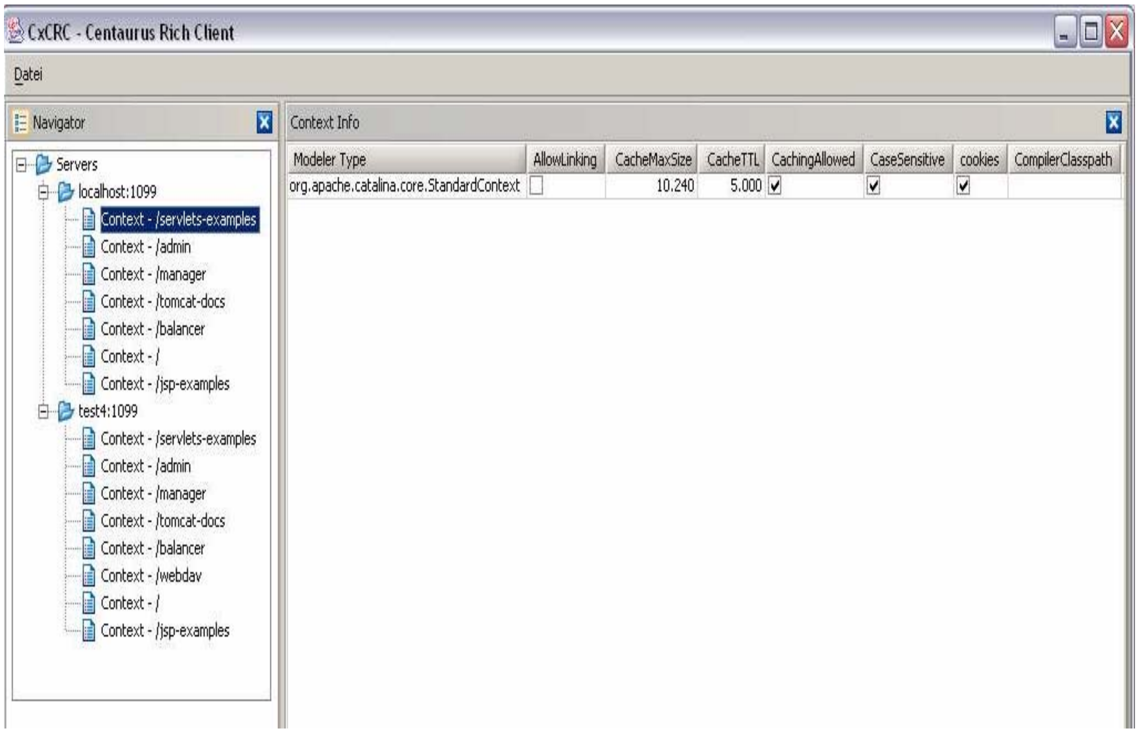
1. Retrieve *addr* from where it was stored in step 3 above
2. Call *JMXConnectorFactory.connect(addr)*

In our case, the application *appl* is the Tomcat server. For a given instance of the Tomcat server, there may be one or more MBean server and a connector server is attached to each MBean server. The client end will have the following code to establish a connection to the instance of Tomcat:

```
// The address of the connector server
JMXServiceURL address = new
JMXServiceURL("service:jmx:rmi://test4:1099");

// The creation environment map, null in this case
Map creationEnvironment = null;

// Create the JMXConnector
JMXConnector cntor = JMXConnectorFactory.newJMXConnector(address,
creationEnvironment);

// The connection environment map, null in this case
// May contain - for example - user's credentials
Map connectionEnvironment = null;

// Connect
cntor.connect(connectionEnvironment);

// Obtain a "stub" for the remote MBeanServer
MBeanServerConnection mbsc = cntor.getMBeanServerConnection();
```

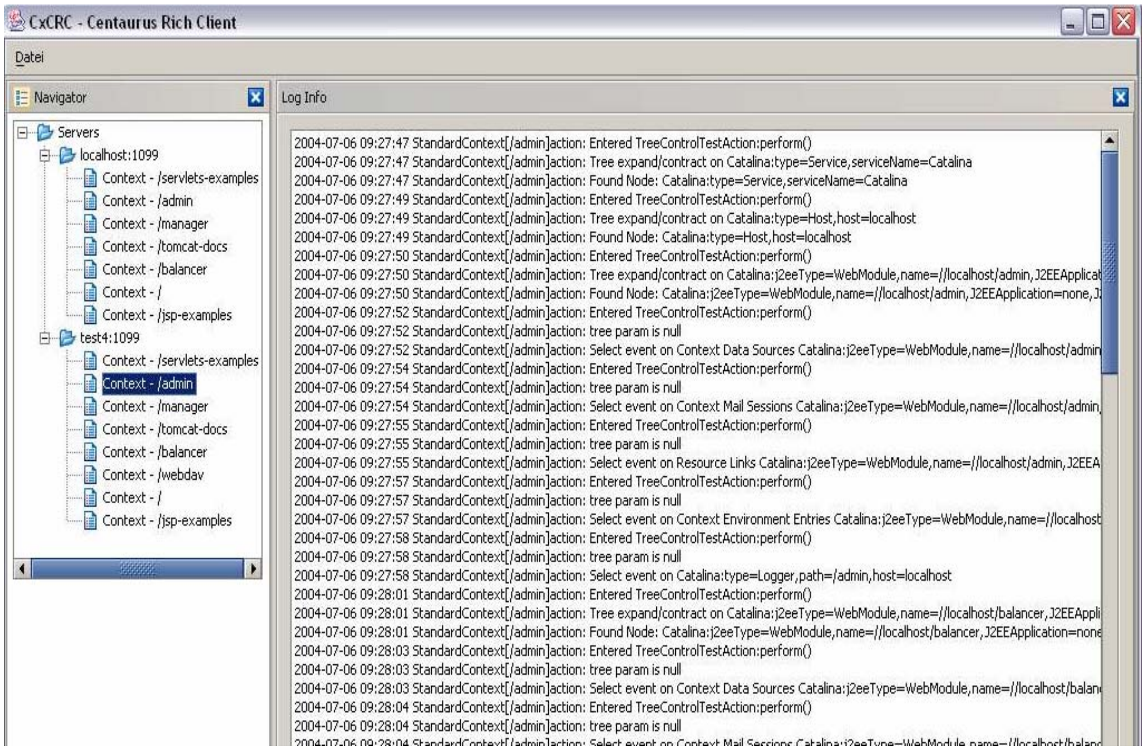
Once the stub is obtained, the server can be queried for the management interfaces and metadata with the *ObjectName* which uniquely identifies each MBean.

Though Tomcat comes with a web-based management console, the main idea is to manage multiple instances of Tomcat servers under single interface. The following screen shots demonstrate the multiple instance management.



Modeler Type	AllowLinking	CacheMaxSize	CacheTTL	CachingAllowed	CaseSensitive	cookies	CompilerClasspath
org.apache.catalina.core.StandardContext	☐	10.240	5.000	✓	✓	✓	

Figure 25: ***Management attributes for a web-module in an instance of Tomcat server***



```
2004-07-06 09:27:47 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:47 StandardContext[/admin]action: Tree expand/contract on Catalina:type=Service,serviceName=Catalina
2004-07-06 09:27:47 StandardContext[/admin]action: Found Node: Catalina:type=Service,serviceName=Catalina
2004-07-06 09:27:49 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:49 StandardContext[/admin]action: Tree expand/contract on Catalina:type=Host,host=localhost
2004-07-06 09:27:49 StandardContext[/admin]action: Found Node: Catalina:type=Host,host=localhost
2004-07-06 09:27:50 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:50 StandardContext[/admin]action: Tree expand/contract on Catalina:j2eeType=WebModule,name=//localhost/admin,J2EEApplicat
2004-07-06 09:27:50 StandardContext[/admin]action: Found Node: Catalina:j2eeType=WebModule,name=//localhost/admin,J2EEApplication=none,J
2004-07-06 09:27:52 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:52 StandardContext[/admin]action: tree param is null
2004-07-06 09:27:52 StandardContext[/admin]action: Select event on Context Data Sources Catalina:j2eeType=WebModule,name=//localhost/admin
2004-07-06 09:27:54 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:54 StandardContext[/admin]action: tree param is null
2004-07-06 09:27:54 StandardContext[/admin]action: Select event on Context Mail Sessions Catalina:j2eeType=WebModule,name=//localhost/admin,
2004-07-06 09:27:55 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:55 StandardContext[/admin]action: tree param is null
2004-07-06 09:27:55 StandardContext[/admin]action: Select event on Resource Links Catalina:j2eeType=WebModule,name=//localhost/admin,J2EEA
2004-07-06 09:27:57 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:57 StandardContext[/admin]action: tree param is null
2004-07-06 09:27:57 StandardContext[/admin]action: Select event on Context Environment Entries Catalina:j2eeType=WebModule,name=//localhost
2004-07-06 09:27:58 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:27:58 StandardContext[/admin]action: tree param is null
2004-07-06 09:27:58 StandardContext[/admin]action: Select event on Catalina:type=Logger,path=/admin,host=localhost
2004-07-06 09:28:01 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:28:01 StandardContext[/admin]action: Tree expand/contract on Catalina:j2eeType=WebModule,name=//localhost/balancer,J2EEAppli
2004-07-06 09:28:01 StandardContext[/admin]action: Found Node: Catalina:j2eeType=WebModule,name=//localhost/balancer,J2EEApplication=none
2004-07-06 09:28:03 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:28:03 StandardContext[/admin]action: tree param is null
2004-07-06 09:28:03 StandardContext[/admin]action: Select event on Context Data Sources Catalina:j2eeType=WebModule,name=//localhost/balan
2004-07-06 09:28:04 StandardContext[/admin]action: Entered TreeControlTestAction:perform()
2004-07-06 09:28:04 StandardContext[/admin]action: tree param is null
2004-07-06 09:28:04 StandardContext[/admin]action: Select event on Context Mail Sessions Catalina:j2eeType=WebModule,name=//localhost/balanc
```

Figure 26: Log ***information of a web-module in an instance of Tomcat server***

8 Conclusion

Management and monitoring are essential parts of an effective e-business. Most application developers view management as an afterthought when building and delivering their applications. In some cases, developers try to anticipate administration and monitoring needs of their users, and build such administration, logging and instrumentation into their applications. This often uses a variety of proprietary mechanisms for consoles, log files, and instrumentation. While this may fulfil the immediate needs anticipated by application developers, it is usually a poor fit with the real requirements of enterprises that have to deal with a complex integration of many such applications. If each application has its own way to do administration and monitoring, handling the diversity of management tools becomes its own challenge.

Hence a generic effective way of managing the J2EE application both at the development and production phase has to be devised at the development phase itself. Various methodologies, some of which are current industry standard, are analyzed and better solution is approached through the technologies: JMS and JMX. The core need for management is the notification model. The notification model triggers an event as when an action is performed. The action could be a database access, execution of a critical part of business logic. The action is the breakpoint which is set by the developer. The events that are triggered hold the performance metrics or the business data itself for management and monitoring. The bottlenecks, if any arose, can be determined beforehand by analyzing the data received from the application through the notification model. The notification model has been developed with JMX and JMS to work seamlessly both in the local environment and the distributed environment. The application can also run in a clustered environment. JMX has some pitfalls in the clustered environment and JMS fills that gap. Most J2EE application servers in the market partially or fully support clustering of JMS.

As the management and monitoring part comes as a part of the application itself, a simple off/on option of the management has also been devised. The little overhead that comes due to the management factor can be easily removed when not monitored without changing a single line of the existing code.

The future work will include an auto-generation framework that will generate the necessary code, relieving the developer from manually writing the management code at critical points of the business logic. A simple XML file that specifies the critical parts of the business logic, as set by the developer, and the type of management code that is to be generated. The generator reads the XML file, analyses the business logic and generates the code as specified without touching the business logic. The generated should also be able to extract the business logic specific data also.

References

[1] *Java 2 Platform Enterprise Edition Specification*, Version 1.4, Sun Microsystems, Inc., November 2003.

[2] *Java Management Extensions (JMX) Instrumentation and Agent Specification*, Version 1.2, Sun Microsystems, Inc., October 2002.

[3] *Java Management Extensions (JMX) Remote API Specification*, Version 1.0, Sun Microsystems, Inc., October 2003.

[4] *Java Platform, Enterprise Edition Management Specification - JSR-77*, Version 1.0, Hans Hrasna, Sun Microsystems, Inc., June 2002.

[5] *Java Message Service Specification*, Version 1.1, Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli and Kate Stout, Sun Microsystems, Inc., April 2002.

[6] *Enterprise JavaBeans Specification*, Version 2.1, Linda G. DeMichiel, Sun Microsystems, Inc., November 2003.

[7] *JBoss Administration and Development*, Scott Stark, Marc Fleury, The JBoss Group, Sams publication, 2002, ISBN: 0672323478.

[8] *Managing Java Applications with JMX*, J. Steven Perry, O'Reilly Publication, 2002, ISBN: 0596002459

[9] *WebLogic: The Definitive Guide*, Jon Mountjoy, Avinash Chugh, O'Reilly publication, 2004, ISBN: 059600432X

[10] *JMX in Action*, Benjamin G. Sullins and Mark B. Whipple, Manning publications, 2002, ISBN: 1930110561

[11] *Java Message Service Tutorial*, Kim Haase, Sun Microsystems, Inc., 2002, [http://java.sun.com/products/jms/tutorial/index.html]

[12] *Enabling Component Architectures with JMX*, Marc Fleury, Juha Lindfors, 2001, [http://www.onjava.com/pub/a/onjava/2001/02/01/jmx.html]

[13] *Using JMX to manage Web Application*, Tony G. Thomas, AdventNet Inc., March 2003, [http://www.theserverside.com/articles/article.tss?l=JMXWebApps]

[14] *JBoss Clustering Analysis*, Özalp Babaoğlu, Alberto Batoli, Vance Maverick, Alberto Montresor, Davide Rossi, Jakša Vučković, March 2003, Information Society Technology Programme of the 5th Framework (1998-2002).

[15] *Using WebLogic Server Clusters*, Version 8.1, BEA Systems Inc., 2004, [http://edocs.bea.com/wls/docs81/cluster/]

[16] *Clustering with JBoss 3.0*, Bill Burke, Sacha Labourey, October 2002, [http://www.onjava.com/lpt/a/1517]

[17] *JMX Studio product documentation*, Version 5, AdventNet Inc.

[18] *Introducing A New Vendor-Neutral J2EE Management API*, Andreas Schaefer, March 2002, [http://www.onjava.com/pub/a/onjava/2002/03/27/jsr77.html]

[19] *Introduction to Aspect-Oriented Programming*, Graham O'Regan, January 2004, [http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html]

[20] *Separate software concerns with aspect-oriented programming*, Ramnivas Laddad, April 2002, [http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html]

[21] *Aspectwerkz – Dynamic AOP for Java*, Jonas Bonér and Alexandre Vasseur, [http://aspectwerkz.codehaus.org/index.html]

[22] Performance Analysis of J2EE Applications Using AOP Techniques, Ramchandar Krishnamurthy, May 2004, [http://www.onjava.com/pub/a/onjava/2004/05/12/aop.html?page=1]

[23] *Implementing vendor-independent JMS solutions*, Nicholas Whitehead, IBM Software Group report, February 2002, [http://www-106.ibm.com/developerworks/java/library/j-jmsvendor/]

[24] *Java theory and practice: Coaxing J2EE out of the container*, Brian Goetz, IBM Software Group report, April 2004, [http://www-106.ibm.com/developerworks/java/library/j-jtp04204.html]

[25] *BEA WebLogicTM JMS Performance Guide*, BEA Systems Inc., July 2003.

[26] *Java Management Extensions White Paper - Dynamic Management for the Service Age*, June 2001, Sun Microsystems Inc.

[27] *JMX for Managing Java Applications*, Daniel F. Savarese, October 2003 issue of JavaPro magazine.

[28] *EJB 2 Clustering with Application Servers*, Tyler Jewell, December 2000, [http://www.onjava.com/pub/a/onjava/2000/12/15/ejb_clustering.html]

[29] *Draft paper - Achieving Instant Insight into the Real-time Electronic Enterprise*, David Luckham, Stanford University.

[30] *Complex Event Processing in Distributed Systems*, David C. Luckham and Brian Frasca, Program Analysis and Verification Group, Computer Systems Lab, Stanford University, August 1998.

[31] *Core J2EE Patterns*, Deepak Alur, John Crupi, Dan Malks, Sun Microsystems press, A Prentice Hall Title, 2001.

[32] [http://jboss.org/wiki/Wiki.jsp] - JBoss documentaion.

[33] [http://mx4j.sourceforge.net/] – Open source JMX for Enterprise computing.

[34] [http://www.jboss.org/index.html?module=bb] – JBoss forum.