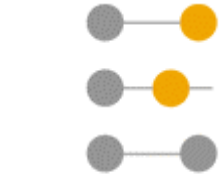

LeCANDO

Master's Thesis in Computer Science

An Event Service Implemented with J2EE for Integration of Enterprise Systems

by

Markus Wurz

Department of Microelectronics and Information Technology,
Royal Institute of Technology (KTH)

April 2001

Examiner:

Prof. Seif Haridi
Department of Microelectronics and
Information Technology,
Royal Institute of Technology

Supervisors:

Ass. Prof. Vladimir Vlassov
Department of Microelectronics and
Information Technology,
Royal Institute of Technology

M.Sc. Johan Lins
Lecando AB

Abstract

The major topic of this Master thesis is event monitoring in distributed systems and use of an event mechanism for integration of enterprise systems. In particular, this thesis studies an approach of event propagation using the server-side technology – Java 2 Platform, Enterprise Edition.

The thesis presents design, implementation and performance evaluation of an event service that can be used to integrate and to connect enterprise systems. The service allows systems to interact and communicate synchronously and asynchronously via an event signalling mechanism where events are propagated as messages from a source system to destination systems. Asynchronous communication between the systems is the key feature of the service.

An e-Learning system based on the Lecando Enterprise Server is used in this thesis as a case to study the needs and requirements that an enterprise application might have for an event service to be linked to other systems.

The main objectives for the event service to be used by integrated enterprise systems are to avoid degradation in system performance, and to provide high reliability, so that events propagated via the service are never lost.

The results of the performance evaluation show that the event service is extremely efficient in receiving and propagating events from one enterprise system to others.

Contents

List of Figures

List of Tables and Diagrams

1 Introduction

1.1 Background and motivation for the project

The Internet opens the market for new industries in different areas. Such as e-Commerce, e-Learning and e-Business. Enterprises are usually specialized in one area, but not more than that. It becomes important that software products of different vendors can be integrated to each other in order to meet the market requests. It is not an easy task to integrate such enterprise systems. The problem is that changes in one system might effect other systems. Many questions arise immediately: How should information be exchanged across systems and platforms? Should for every state change a notification be sent to all connected systems? If the software of one of the integrated systems is being changed, does this effect all other systems? Some of the questions will be answered in this project.

This thesis will be concentrated in event monitoring and propagation of event notifications. Changes in one system should effect other systems. This is done by modelling these changes as events. Event monitoring is the technique of capturing events and propagating them to other systems. In Figure 1.1, enterprise systems will be connected to a middleware component that will be called Event Service. This component is a central point and used to distribute events. The enterprise systems that need information on certain business data, needs to subscribe to the event representation of that data in order to receive it.

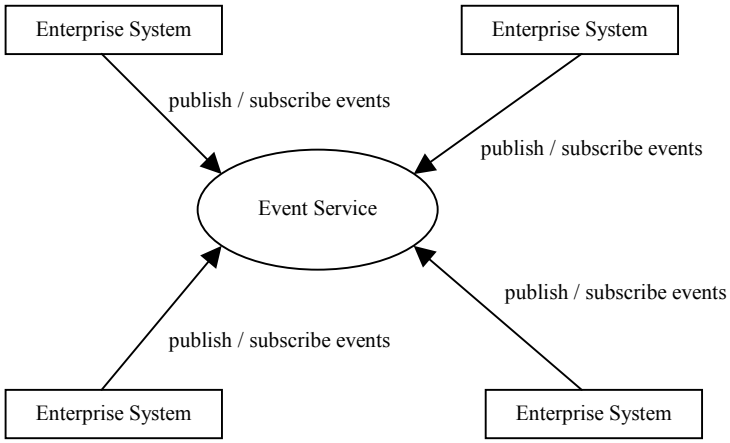

Figure 1.1 Symmetric view of systems communicating with the Event Service.

E-Learning is an alternative to conventional training and education, based on the Internet. This means that students all over the world can participate together in the same course, groups or projects. Books are replaced by multimedial content, that are accessible through the user interface. Tests are done online. Communication between students are done either via email, chat or news groups.

An e-Learning system from Lecando is used in this thesis as a case study in development of the Event Service. Lecando is a company, that developed an e-Learning platform with the need of having the Event Service module for business-critical information exchange with other enterprise systems. The platform is called Lecando Enterprise Server. It is a server-side

multitiered enterprise system build upon the specifications of Java™ 2 Platform, Enterprise Edition [3].

Lecando wants to use the Event Service to extend its platform to provide their customers with a more advanced and functional product for e-Learning. Interesting systems to integrate with the Lecando Enterprise Server are customer resource management system, human resource management systems or other enterprise information systems.

1.2 Aim of the project

The main objective of the project is to analyse, design, implement and evaluate an approach for asynchronous event monitoring in distributed enterprise applications. The Event Service will be able to define and add events to the system. An easy approach for processing and distributing events to integrated systems is also provided.

Integration is not brand new and therefore parts of this thesis concentrate on studying existing event monitoring applications and other research material covering that subject. With the newly gained information on event monitoring, the Event Service will be designed and implemented. The technology used for implementation of the Event Service will be Java™ 2 Enterprise Edition (J2EE) [3] and in particular Enterprise JavaBeans™ (EJB) [4] and Java™ Message Service (JMS) [6]. Finally the system's performance will be evaluated in order to measure the efficiency of the Event Service.

1.3 Organization of the report

The remainder of this report is organized as follows.

Chapter two presents a case study of the Lecando Enterprise Server. Furthermore are possible use cases for the Event Service framework presented, its architecture and functions.

Chapter three considers related work and the requirements for the Event Service are specified.

Chapter four presents technologies that are used for the implementation of the Event Service. These are the Java™ 2 Enterprise Edition (J2EE) [3], the Enterprise JavaBeans™ (EJB) [4] and the Java™ Message Service (JMS) [6].

Chapter five presents design and implementation of the Event Service. After that the design and implementation is described. The chapter finishes with an example of one of the use cases described in chapter 2.2.

Chapter six presents the performance evaluation of the Event Service and a discussion, whether the requirements are fulfilled.

Chapter seven summarizes the work. It also gives conclusions and suggestions what can be done to improve the system in the future.

2 Case study: Lecando Enterprise Server

This chapter gives a short overview of the Lecando Enterprise Server used as a case study in this thesis. It shows examples (use cases) of how Lecando Enterprise Server can be integrated

with other systems. Next chapter considers related work and together with the case study of the Lecando Enterprise Server the requirements of the Event Service are specified.

2.1 Overview of the Lecando Enterprise Server

First the different business modules are listed and explained. Afterwards the architecture of the Lecando Enterprise Server is described. The Lecando Enterprise Server can be divided into four modules: *Management*, *Content Creation*, *Project Environment* and *Training*.

The *Management module* is used to administrate all the system's users and the course content. New users are added here to the Lecando Enterprise Server. The roles for the users in the system are defined here, e.g. students, teachers, principals, etc. Courses are also organized in this module. Students and teachers are registered to courses.

In the *Content Creation module*, the content is defined as course material, tests, evaluations and assignments. The content of a course consists of several chapters. Each chapter has its own structure with titles, pages and with multimedial content.

The *Project Environment module* is for users that want to be able to communicate, share files and working together, outside of the courses that they are studying. Projects can be started by any user of the system.

The *Training module* includes chat-rooms, whiteboards and news. Students who are taking the same course can do lectures together with the whiteboard application, which can be private for a group of users. Messages can be posted and read in different news groups. Students can see their results on exams, test and assignments.

The business functionalities explained above are all realized by the same server-side architecture depicted in Figure 2.1. That means that every business model is build upon the three-tier structure: Presentation layer, Business Logic layer and the Persistence layer.

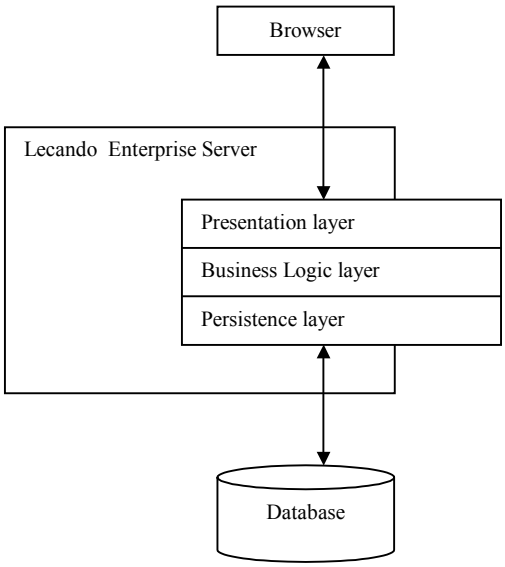

Figure 2.1 The Lecando Enterprise Server architecture. The user of the system interact with a browser. The browser communicates with the Lecando Enterprise Server. The modules of the business functions all use the three layers: Presentation, Business Logic and Persistence. The persistence layer is connected to a DBMS (Database Management System).

The Lecando Enterprise Server is implemented in Java™ using Java™ 2 Enterprise Edition [3]. J2EE is an open server application standard supported by leading vendors of distributed systems. See chapter 4.1 for more information about the J2EE technology.

The end user interacts with the e-Learning platform using a standard web browser. The Lecando Enterprise Server is a multi-layered application, implemented according to well established patterns for distributed applications. The layers in the server consist of the Presentation Layer, the Business Logic Layer and the Persistence Layer as seen in Figure 2.1. For more information about multi-layered distributed applications see [1].

The Presentation Layer communicates with the end user through the web browser. Communication is achieved using HTTP and HTML standards [9]. The Presentation Layer can also be equipped with Secure Socket Layer encryption (SSL) [9], upon which HTTPS [9] is used. The Presentation Layer is implemented with Java™ Servlets and Java™ Server Pages (JSP) technologies [8]. The Business Logic Layer contains all the logic in the systems. It is implemented with components called session beans, which in turn is part of the Enterprise JavaBeans™ technology [4].

The Business Logic Layer communicates with the Presentation Layer, providing the information and services it needs.

The Persistence Layer provides the system with the information stored in the Database Management System (DBMS). The Persistence Layer is implemented with entity beans, also part of the Enterprise JavaBeans™ technology (see chapter 4.2 and [4]).

2.2 Use cases for information exchange within an e-Learning system

A few examples are given here to demonstrate how an e-Learning platform can use the Event Service to distribute data to other systems. Data changes that is important to other systems are modelled by events. When such data has to be exchanged with some other systems, a specific event is fired in order to publish it to the Event Service.

Examples for such systems are e-Commerce, knowledge management system, customer resource management systems, human resource management systems, external result-handling systems, or existing user information databases.

With the Event Service the following use cases are easy to implement.

- Updating user information in an external database whenever user data is changed at the Lecando Enterprise Server. This is necessary, e.g. if a customer of the e-Learning platform has a separate business application that depends on the correct user data. It is necessary that changes on user data in the e-Learning platform, triggers an event with the new data, captured by the Event Service and sent to the subscribing business application to update a database management system (DMBS).
- Sending student results to a result-handling system and notifying the students is another interesting use case. Here the functionality of the Lecando Enterprise Server is extended with functionality that every graded exam, test and assignment, will cause an event to be fired with the information of the respective result of the grading. These events will be received by a system for handling student results and also by an application that sends a standardized email to inform the student about the result.

- Users of the e-Learning platform will be billed by an external e-Commerce system that sends invoices to the users. This could be done by firing an event when a user gets registered at a course or a seminar. The event will be send via the Event Service to the e-Commerce system where the registration will be handled.

The list could be longer. Mentioned here are just a few examples listed that makes the task for the thesis easier to comprehend.

3 Related work and requirements for the Event Service

This chapter considers related work and with the case study of the previous chapter, the requirements are specified.

Messaging middleware and Event Notification Systems are studied to get information about existing approaches in event monitoring. The Event Notification Systems is a framework for a dynamical approach to publish and subscribe events. There are a few products that have implemented parts of that framework, most of them are only used for academic purposes. On the other hand, there are commercial products most often called Messaging-middleware. They are quite similar to the Event Notification Systems, but less complex. Some of the most important terms and concepts will be explained later in this chapter to make it easier to compare those products with each other.

3.1 Concepts, Semantics and Architecture

Event Notification Systems are described in the papers: [10], [11] and [14].

In distributed systems literature there is no straightforward definition for an **event**. Some describe events as happenings of interest that occur in a system. Elsewhere events represent state changes of interests that may occur in a system. Every event is causing an action, most likely in form of an event notification message that is send to interested parties.

Event Publisher is a component that will generate events and have to register them at the event service before they can be published. After that, whenever an event occurs the publisher can send a notification about the occurrence of the event to the event service. It is also possible for the event publisher to cancel a registration of an particular event.

A component that is interested in getting notifications about events is called **Event Subscriber**. They have to subscribe to the events before they can receive them. Only events that are registered at the event service can be subscribed upon. Subscription can be done on a single event or a group of events. It is the component that specifies which notifications are sent to it. It is possible that components can publish one type of event and subscribe on others at the same time.

The **Event Service** could exist of many servers that altogether provide the same service. How many servers are included in the event service depends on the architecture of the system. The event service decouples the event publishers from the event subscribers. This is because components can operate in the system without being aware of other components. Components need to know about the event service and how to communicate with it. The event publisher can send notifications to the event service after it has registered the event. It is the responsibility of the event service to forward the notification to all subscribers. Sometimes it is necessary that the service is persistent. This means that the event service guarantees

delivery to all subscribers. Most likely there are repositories at one or more servers to fulfil that requirement. This solves the problems when subscribers are not able to receive a notification. The event service also perform some sort of lookup to find all the subscribers to an particular event.

Publishers can register events that they will generate and publish. This process is called **Event Registration**. The events get registered at the event service. In some services it is also possible to cancel a registration. It is not clear if it is the responsibility of the event service to cancel subscriptions on those events.

Event notifications are defined as data structures that are used by the publisher to represent the event. These data structures could include parameters such as type of subject, time of occurrence and information about the event that occurred.

A **event subscription** is generated by components that are interested in receiving a notification about the occurrence of an event. Subscriptions are handled by the event service. Components can also unsubscribe events when they are not interested in them any longer.

Selection mechanisms of the Event Notification Service

Publishers send notifications of events to the event service. They will be forwarded to subscribers. Subscribers want to receive notifications that they have subscribed to, not to every notification that is send to the event service. The event service has to select the notifications and send them to the subscribers that want to be notified about the events. A filter selects one event notification at a time, a pattern can select several notifications that together match an algebraic combination of filters. Two mechanism are usually used: Filters and patterns.

An **event filter** defines a class of event notifications by specifying a set of attribute names and types and some constraints on their values. Filtering is done on channels, subjects and content. This is then called **channel-based**, **subject-based** respective **content-based** filtering. More about that follows below. A pattern of events is defined by combining a set of event filters. An event filter is itself a pattern and two or more patterns can be combined to build a new one. Patterns usually operate in content-based filtering event services.

Subscription mechanisms

There are different types of subscription modes that are offered by the event service. Normally not all of them are supported at the same time. These are: **Channel-based, subject-based** and **content-based** subscription. Interested parties can subscribe or listen to a channel. The channels are identified by an unique channel id. The Event publishers send notifications to one or more channels. The Event service identifies the events and delivers them to all subscribers of the event.

In subject-based event notification services, the notifications contain a well-known attribute named subject. Producers register the subject at the service and publish those notifications under the specified subject. Notifications want to be received by subscribers. The service just has to identify the subject in the notification, search for the subscribers to these subjects and send those to them. With subject-based subscription it is possible to express interest in many subjects by specifying some form of expression. It will be used when the event service has to determine the subscribers for a notification.

In content-based subscription, notifications are handled by the event service based on the whole content which is being sent in the messages and not by specified subject fields. Producers will generate messages without a particular destination. Message delivery depends on the subscribers that are interested in them. They have to specify predicates that are used by the Event Service for filtering the notifications and forwarding them to the subscriber. The delivery of messages to subscribers is completely independent of the publisher, it is the content that matters for message delivery.

Server Topologies

Servers in an event-based notification service could be organized in many different ways. A description of the most common configurations follows below. Hybrid architectural configuration are also possible but less usual. The most common topologies are: **Centralized**, **hierarchical**, and **peer-to-peer**.

The network topology is configured with one central server with all its clients around it. All notifications are sent through this single unit. An advantage of this composition is simplicity, but one of the major disadvantages of this configuration is that the server could be a bottleneck, as all communication is supposed to passed through it.

In hierarchical server topology, the event service is built as a set of distributed components. This means that the event servers are organized in a hierarchical fashion. Each event server manages the communication among the components that are local to the server. These components are objects of interest for interested parties. The servers have to communicate with each other to build a transparent event service to the user. Subscription are stored and forwarded upward in the hierarchy until they reach the root server. Notifications are sent to all the local servers, all low-level servers that have forwarded a corresponding subscription. The hierarchical topology is an extension to the centralized one. A propagation mechanism between the event server has to be added to the centralized version, to make it hierarchical. One of the problems with this architecture is the overloading of high-level servers. All subscriptions and notifications are forwarded upwards and downwards through the hierarchy. Failures of a single server can disconnect the whole subnet reachable from this server.

There are two different forms of the peer-to-peer architecture: **acyclic** and **general**. In acyclic topology, servers communicate with each other through a bi-directional protocol for the flow of subscriptions and notifications. The clients communicate as described above through the publish/subscribe protocol with the servers. A problem with this architecture is that a failure of one link separates the event service and the servers can not communicate any longer with each other. The general peer-to-peer architecture is an extension to the one which is acyclic. Servers are allowed to have connections to more than two servers. One of the advantages is that it is more flexible in the connection between the servers. It is also possible to have redundant paths to the servers. In case of a server failure, it is still possible that the other servers reach each other through a different path.

3.1.1 Survey of Event Notification Systems and Messaging middleware

This chapter describes some event notification services and messaging systems using the terms and concepts described in the previous chapter. Event notification services and messaging systems are quite similar, the only difference is that event notification services are using events to send data and in messaging systems data is sent by using messages instead.

iBus is a commercial product from SoftWired [19]. It supports the peer-to-peer topologies described above. Subscription is done through channels. There are two communication models implemented: Publish/Subscribe, which is asynchronous and request/reply, which is synchronous. Information gets pushed from the producer to the consumer transmitted through the channel. The transport mechanism is similar to IP multicast. iBus was built to fit the Java™ Messaging Service specifications [6].

The **CORBA Event Service** [17] defines a set of interfaces that allows objects to communicate with each other through channels. The communication is synchronous and both push and pull techniques for the clients are supported. All subscribers connected to a channel receive the notifications published on that channel. Both centralized and hierarchical server topologies are supported by many vendors who have implemented the specifications. Unfortunately are some aspects of event notification and event observation not supported which makes it quite limited for other solutions.

TIB/Rendezvous [15] from Tibco utilizes a distributed architecture with a hierarchical server topology. It supports both publish/subscribe and request/reply messaging. This means both synchronous and asynchronous communication. Subscription is subject-based and the notifications are self-describing and platform independent. The notifications supports a user-extensible type system like XML. The subject is a list of strings over which it is possible to specify filters based on a limited form of regular expressions. TIB/Rendezvous APIs are available in Java and a few other programming languages.

The **Elvin** Event Notification Service [14] is a research project of the Distributed Systems Technology Centre at the University of Queensland, Australia. The event service is implemented with a centralized server topology. Notifications are sets of named and typed data elements. A subscription is a declarative boolean expression over the components of the event notification. A component can declare its interest on a number of notifications characterized by some common properties. The producers pushes notifications to the service, which in turn pushes them to the consumer. The event service which uses content-based notification selection, compares the received notifications with the registered subscription expressions and forwards those whose expressions are satisfied. The communication between the components is based on publish/subscribe protocol with quenching. Quenching allows producers to receive information about what consumers that are expecting the information. Events are only generated when they are demanded.

Java Event-based Distributed Infrastructure (**JEDI**) [18] is a research project at Politecnico di Milano. The event service organizes the server hierarchically. Subscription is subject-based and it is possible to unsubscribe events at runtime. The notifications are structured in an object-oriented fashion and furthermore are they defined by name and a number of parameters. This system is similar to Elvin.

SmartSockets [12] is a publish/subscribe middleware solution from Talarian. The architecture of servers is structured in a peer-to-peer fashion. It allows synchronous and asynchronous data transfers. Subscription is done by specifying a subject. The event service can accept subscriptions of a number of different subjects. A notification consists of a subject part and a data part. Components receive all notification belonging to the subject they have subscribed on.

Gryphon [16] is a research project of IBM. The Event Service is structured in a distributed peer-to-peer topology. The flow of events are described via an information flow graph. This

flow graph is responsible for filtering the events, which is content-based. When two different sources produces event streams, they get transformed in a new single stream. Subscriptions are modelled as conjunctions of basis predicates defined over specific attributes.

The **Keryx** Notification System (KNS) [13] was developed by Hewlett Packard. KNS supports communication between loosely coupled components. These do not have to know of the existence of each other. All routing is done by the Event Distributor (ED) based on the content of the messages. The events are encoded in SDR (Self-Describing Data Representation), a textual syntax for structured data. There are three possible ways to build these structured events: map values, list and atomic values. Furthermore is content-based event filtering supported. Clients send requests for delivery of events to the Event Descriptor, which are called event subscriptions. A subscription defines the events of interest to a client by including a filter that includes the necessary predicates over events. Just the events that pass through these filters are delivered to the clients. The communication protocols used by Keryx are TCP and UDP. An implementation of the Keryx Notification System is done in Java with a hierarchical server topology.

Siena [10] is one of the most advanced event notification services. It is a research project of the University of Colorado. The servers that build the event service could be structured both hierarchical and peer-to-peer. The event notification service exports functions which are usually referred to as the publish/subscribe protocol. The notifications sent by clients have sets of attributes with a type, name and a value. Efficient routing based on the content of the notifications is one advantage of the defined set of those. The selection of the notifications is done with filters and patterns. There exists a prototype implementation in Java.

Sun Microsystems has defined the **Java™ Message Service** (JMS) specification [6]. Various companies have implemented the JMS specifications. SoftWired with its iBus is one of them. Some key features of JMS are: Support of asynchronous messaging, subscription is channel-based and the architecture of the service is centralized. Message filtering is done with message selectors. Another important fact of JMS is that it is included in the specifications of Java™ 2 Enterprise Edition, which is used in many Internet business solutions. In chapter 4.3, more information about JMS is provided.

3.1.2 Discussion

Table 3.1 presents a comparison of Event Notification Systems and Messaging middleware described in the previous chapter. iBus is a product that is implemented under the JMS specification. Recently many more new products have been introduced onto the market for messaging middleware. They differ in the server architecture and the way they subscribe to events (messages).

Most of the products above are not convenient to be used in the project. They have a quite complicated server structure that will be too complex to use for the purposes of this project. Siena, Keryx, Gryphon, JEDI, and Elvin are research projects without a stable implementation. No system fulfils our needs since non can provide a mechanism for delaying events. An implementation of JMS will be used inside the Event Service. JMS is used with a centralized server and subject-based subscription. Asynchronous and synchronous communication can be used with JMS. Another advantage that comes with JMS is that it is included into the J2EE specification, which many enterprise application systems are based on. An example is the Lecando Enterprise Server, which is using the Orion Application Server [20], an implementation of the J2EE specification.

	Architecture			Subscription		
	Centralized	Hierarchical	Peer-to-peer	Channel-based	Subject-based	Content-based
JMS	X		X	X	X	
iBus	X		X	X	X	
CORBA Event Notification Service	X	X		X	X	
TIB/Rendezvous		X			X	
Elvin	X					X
JEDI		X			X	
SmartSockets			X		X	
Gryphon			X			X
Keryx		X				X
Siena		X	X			X

Table 3.1 Overview of the products described in chapter 2.3.2. Architecture and subscription types are compared.

3.2 Requirements for the Event Service

As shown in chapter 3, there are many different solutions for distributing events from one system to another. There are three important requirements on the Event Service that have to be fulfilled in order to reach a satisfying result for the project. These are:

- **Ease of usage**: The usage of the Event Service should be easy for the application developers. Only an overview of the Event Service should be enough information to use it. External consultants should be able to manage implement customized handlers without deeper knowledge of the technology used in the Event Service module.
- **High Performance**: The Event Service should have short response time. This means that the Lecando Enterprise Server (monitored system), that is responsible for the information sending, should not be occupied to long while communicating with the Event Service.
- **High Availability**: Events that are fired should never get lost. The Event Service should be fault tolerant. Failures in the Event Service are handled with the Java Exception mechanism. Most critical data will be implemented by entity java beans and must be persistent. The issue of high availability is not considered in this thesis.

In the next chapter programming technologies are represented which will be used to implement the Event Service.

4 Overview of Enterprise Java technologies

This chapter presents some of the technologies used in the implementation of the Event Service. First a brief introduction is given to server-side programming: Java™ 2 Enterprise Edition (J2EE) [3]. After that the Enterprise™ JavaBeans (EJB) [5] and the Java™ Message Service (JMS) [6] technologies are introduced and briefly explained. How EJB and JMS technologies are used to obtain asynchronous communication between event publishers and the JMS server is also discussed in this chapter.

4.1 Java™ 2 Enterprise Edition

The J2EE platform provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. The J2EE specifications provide support for a large number of Java APIs; such as

- Java Database Connectivity (JDBC),
- Remote Method Invocation (RMI),

- Java IDL (CORBA Distributed Objects),
- Java Naming and Directory Services (JNDI),
- Enterprise JavaBeans (EJB),
- Java Servlets,
- Java Messaging Service (JMS),
- Java Transactions API (JTA),
- Java Server Pages (JSP),
- JavaMail,
- JavaBeans Activation Framework.,
- Java API for XML (JAXP),
- J2EE Connector,
- Java Authentication and Authorization Service (JAAS).

Some of the Java technologies listed above will be briefly explained in this chapter.

A multitiered distributed application model is used within the J2EE platform. This means that application logic is divided by their functions into components that together form the J2EE application. Normally, these components are placed on different machines, depending on the tier in the J2EE environment they belong to. An application could have the following tiers:

- Client tier components run on the client machine,
- Web tier components run on the J2EE server,
- Business tier components run on the J2EE server,
- Enterprise information system (EIS) tier, run on some database server.

This is considered as a three-tiered model since the tiers are placed in three locations: client machine, J2EE server machine and the database machine. The client components could be either web-based or non web-based. An ordinary web-browser downloading content from the web tier, is such an example. A web component consists either of Java™ Servlets or JSP pages [8]. Servlets are classes that process requests and create responses. JSP pages are run as Servlets, but are used to create text-based content inside HTML pages. The business tier is the logic that solves the needs for the business application. Inside this tier you find enterprise beans, which come in three flavours: session bean, message-driven bean and entity bean. More about that later in chapter 4.2. The EIS tier handles all from database systems to Enterprise Resource Planning (ERP) systems.

The J2EE server, also called Application Server, provides underlying service to each component types in form of containers. This is convenient since the application developer can concentrate on the business logic and will not need to solve complex low-level details like multithreading, resource pooling etc.

The components have their running environment inside its containers. We have web, enterprise and application containers (see Figure 4.1 for an overview). These containers function as an interface between the components and the low-level platform functionality that supports the component. They have configurable and non-configurable services. To the configurable services belong security, transaction management, directory and naming lookups, and remote connectivity. The servlet and bean life-cycle, database connection resource pooling, data persistence, and access to the other J2EE platforms APIs are non-configurable services. This means that you canot change that behaviour.

We will concentrate on **Enterprise™ JavaBeans** and the **Java™ Message Service**, since the implementation of the Event Service uses these two technologies. They are described in the next two chapters. Below you will find a short description about some of the APIs that are included inside the J2EE specifications.

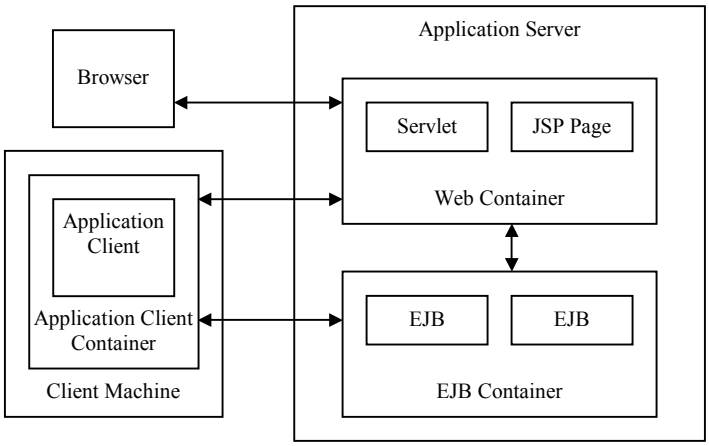

Figure 4.1. The J2EE Application Server and its containers. The web and EJB container are provided by the Application Server. Browsers communicate with the web container. Other client program such as JavaBeans can communicate directly with the enterprise beans within the EJB container. Business logic is processed inside the EJB container and is sent to the web container or the application container.

The **Java™ Database Connectivity (JDBC)** is used to invoke SQL commands from inside Java methods. Usually you will not use this service if you are not using the bean-managed persistence mode for an entity bean. More about that in the next chapter. The JDBC API are divided into two parts: the application-level interface and the service provider interface. The first is used by application components to access a database and the later to connect an JDBC driver to the J2EE platform.

Java™ Servlets are used to extend the functionality of Web servers. Servlet is a server-side application that executes requests and creates responses. Servlets run inside the Application Server within the servlet engine.

With **JavaServer Pages (JSP)**, Java code can be included inside text-based documents like HTML pages. With this technology it is easy to produce dynamic web-content. When such a page is requested by the client, the server returns a dynamically generated HTML page.

Java™ Transaction Architecture (JTA) and **Java™ Transaction Service (JTS)** are used for transaction handling. Applications can use JTA to access the transaction monitor. JTS is the implementation of such an monitor. This monitor provides service to the application server, resource managers, stand-alone applications, etc.

The **Java™ Naming and Directory Interface (JNDI)** is used to access naming and directory services. Such resources are DNS, LDAP, local file system, or objects inside the application server. Objects can be connected to simple names and located from application that need some service of them.

4.2 Enterprise JavaBeans™ (EJB)

Enterprise beans are components that represent a discrete, well-defined piece of functionality. In a distributed multi-tiered application the Enterprise JavaBeans components form the business logic for the application. See Figure 4.1 for an overview of the EJB architecture. In Figure 4.2, the EJB server is hosting the EJB container. Usually the Application server provider will support the EJB server tasks too.

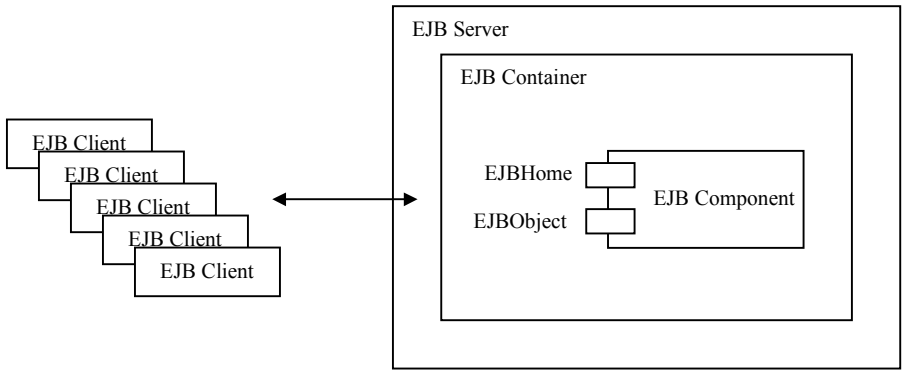

Figure 4.2 Enterprise JavaBeans architecture. Clients that want to use an enterprise bean will use the EJBHome and EJBObject interfaces for using the bean methods.

The EJB architecture consist of four different components:

- The EJB client
- The EJB server
- The EJB container
- The EJB component

Clients to the EJBs are processes that want to use the service of EJBs. Such a client could be an application running on a different machine than the server, e.g. a web browser or a Java servlet located on the same server and providing dynamically created web content. Clients use the remote objects of the EJB to access them. Remote Method Invocation (RMI) is used if it is a Java client and otherwise Corba. Enterprise JavaBeans are located through the JNDI technology. All the client needs is a unique name to reference the EJB with the naming and directory service. When found a reference to an EJBHome object is returned to the client. This object is used to get the reference to the EJBObject object, which is the remote object interface. Now the client can call methods on the EJBObject to access the service that the EJB provides.

The **EJB server**, most likely included into the application server, provides all low-level services required by the EJBs. The naming services provide the service to locate the EJBs, as mentioned above. Who is allowed to access an EJB is controlled by the security services. The persistent data storage is accessed via the database access services. The transactions services are responsible to handle commit and rollbacks. The transaction monitor is also provided if it was implemented into the EJB server. The EJBs life cycle management is another service that is offered to the EJBs. Multithreading is also another service.

The **EJB container** gives the EJBs the environment in which they run. The applications functionality is provided to the EJBs and some other functions are provided by this container. EJBs get registered by the container when these are loaded. The container is responsible for the life-cycle of the EJB, i.e. to create and remove them. The persistence for the EJBs is handled by the container. They can get serialized and stored in the persistent storage when they are not in use. Access control to the EJBs service is also handled by the container. Some methods are made accessible to the clients and others are not.

The **EJB component** is the Java class that represents the business logic of one module of the distributed application. With the EJB2.0 specifications [5], there are three types of enterprise beans: Session beans, Entity beans, and Message-driven beans.

The **EJBHome** and the **EJBObject** are two interfaces that are associated with the EJB component. When the EJB is loaded into the container, their classes are created. The EJBHome interface has for both session and the entity bean create methods that will return an instance of the EJBObject class. The entity bean will have methods, that take a primary key as a parameter and return a reference to a collection of EJBObjects matching the key. The methods that are provided by the EJB component are accessed by the clients through an instance of the EJBObject class.

Session beans represent a set of processes or tasks, which are performed on behalf of the client application. Since the client request a service from a bean, it will get its own instance of the session bean, that cannot be shared with other clients. There are the stateful and stateless session beans. The stateless session bean provides a single-service to the client and can not store any information related to a client between calls. Stateful session beans can retain state between multiple client requests or transactions. Session beans are not persistent, they do not survive if a server is restarted. In this case the clients need to get a new session object.

Entity bean represent specific data. They map a Java class to a data source. This could be a single row or an entire table in the database. Each entity bean has a primary key associated with it that identifies the data inside it. There is only one instance of an entity bean for any primary key because multiple copies of the data would be very difficult to manage. There are two different types of entity beans: bean-managed and container- managed. With container-managed entity beans all database access calls are performed by the container. In a way this is automatic persistence. The other type is the bean-managed entity bean, which itself is responsible to provide all database access calls.

The **message-driven bean** (MDB) is a stateless enterprise bean component that is designed to consume asynchronous JMS messages. This is further explained in the next chapter. The difference between the MDB and the other two bean types, is that it does not have a remote and a home interface. This bean does not have any business methods that are invoked from other clients. The MDB listens only to the virtual channel and consumes the messages posted to them by other JMS clients. Messages that are send to such a channel are forwarded to one message bean instance from the message bean pool of the EJB container. These messages are received by the bean instance when its onMessage method is called.

4.3 Java™ Message Service (JMS)

The Java™ Message Service (JMS) is an interface for communication among clients in distributed applications. The messaging functionality of JMS is described with interfaces that vendors of messaging systems implement. With JMS you can send messages from one client

to another client through the messaging service it provides. There are two different subscription paradigms in use: Point-to-point or publish/subscribe messaging.

In **point-to-point** a message has at most one consumer. Many producers can send messages to the queue that is maintained by a consumer. When a message arrives in the queue it is removed only by the consumer of this queue.

When using the **publish/subscribe** mechanism, messages are published to topics which consumer can subscribe on. These messages are routed to all subscribers of that message. They can be many publishers and subscribers on the same topic.

The **message** consists of two parts: header and body. The header contains routing and identification information. The message body carries the application data or payload. There are several message types which depends on the payload. They are types that carry simple text (TextMessage), serialized objects (ObjectMessage), etc.

The main advantage using JMS is the asynchronous sending methods, which means that a JMS client can send a message without having to wait for a reply. The reason for this is that clients send messages to a topic and do not want to wait for the reply of the subscribers.

The JMS architecture is much larger than described above, for more information see the JMS specifications provided by SUN Microsystems [6].

5 Development of the Event Service

This chapter describes design and implementation of the Event Service. The service is implemented with a central JMS server that receives events in a topic from the system that wants to notify about the occurrence of events. These events will be distributed by the Event Service to all subscribers. In this chapter, first, the global view of the service is specified, second, the components in use are described and, third, the implementation of these components is presented and explained.

5.1 System specification

5.1.1 Three separate units

The units involved in the integration of an enterprise platform to other systems can be identified as follows: the monitored system, various clients and the Event Service itself (see Figure 5.1). The monitored system in our case is the Lecando Enterprise Server. Clients can play different roles. They can act as proxies between the monitored system and a third party or they could simply extent the functionality of the monitored system. Figure 5.1 depicts these three components and their connection to each other.

5.1.2 Events and messages

Events are happenings that are created by the monitored system and sent as messages in order to notify other applications of their occurrence.

The monitored system is the source of all event production. It only makes sense to fire events here, since it is the monitored system that wants to notify third parties of the occurrence of events. Only predefined events can be used by the system. These events have to be defined in

a certain manner that will be described later in this chapter. Since the monitored system has no knowledge who might receive notifications of the event occurrence, it can not possibly know if there are event handlers that want to receive such notifications at all. If there are no event handlers for a fired event, this would lead to bad utilization of an efficient monitored system. The monitored system fires just those events that are defined. The frequency of how often and when events occur is not predictable. The reason is the unpredictable usage of the monitored system. In case of Lecando's e-Learning platform, most events are triggered by users while using the system different times of the day, with different frequency and intensity.

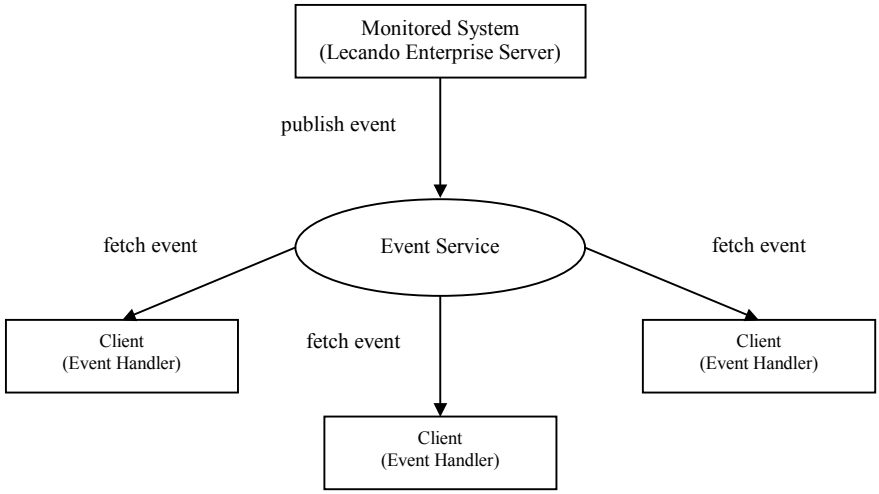

Figure 5.1 Overview of the Event Service. The monitored system will use the Event Service to send notifications about the occurrence of events. Event Handlers are subscribers of those events.

Example of events are: adding a new user to the monitored system, grading exams by examiners or receiving messages from other users. The list of different events is long and there is not really a restriction in the variety of events.

If system performance is getting poor because too many events are fired, one might change the approach in how to use the Event Service.

5.1.3 Interaction between Lecando Enterprise Server and the Event Service

The Lecando Server uses the Event Service to communicate with clients, (see Figure 5.1 for an overview). The Event Service provides an event passing mechanism. It accepts events packed into messages and published by the monitored system. This system has no knowledge how events are handled and who will be notified. There is no feedback from the Event Service to the monitored system. All events sent from the monitored system to the Event Service are kept stored until the clients succeed in processing them. The Event Service guarantees event persistence in case of failure.

5.1.4 Interaction between the Event Service and clients

The Event Service guarantees sending events to all clients that made a subscription to receiving them. In Figure 5.1 the clients are the event handlers. Clients that want to process events not instantly can delay them. The Event Service provides facilities to store them into a database. All delayed events need to have a new execution time. Clients can also use the Event Service to save their own properties. Examples of such properties could be last

execution time, number of tries an event processing are allowed to fail or simply the time when the next execution should take place.

5.2 Design of the Event Service

The monitored system, event handlers and the Event Service are divided into various components that are shown in Figure 5.2 and described in the rest of this section.

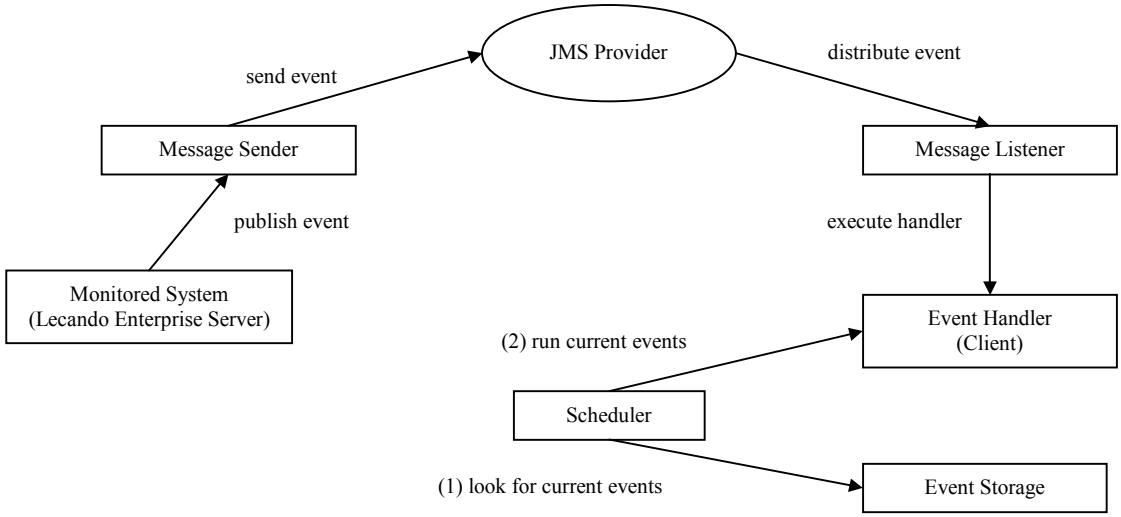

Figure 5.2 The Event Service and its components. The monitored system uses the MessageSender class to send an event to the JMS server (provider). The JMS server distribute the event to the MessageListener. The Message Listener looks for the EventHandlers of the event and executes them. The EventHandlers can delay events and use the EventStorage for this purpose. The Scheduler is responsible for finding and executing delayed events.

5.2.1 Events

Moments arise when the monitored system wants to inform other entities of the occurrence of a specific situation. This situation will be called event. An event is a description of what has happened at the monitored system. The Event Service will get a notification of the occurrence of every event. All events carry information that are useful for other parties. Events that are similar will be built in a special structure. This structure is a tree. On top of the tree is the root event. Every new event will be defined in a sub branch of the tree. The number of branches is not restricted. The reason for this is to make the tree as flexible as possible. Every event in a branch will inherit all information from the event that is higher up in the tree structure and add some more information to it as well. New events are added to a branch of the tree where it fits into the branch. If no branch seems to carry information that a new event could use, a new branch is added to the root of the tree. The reason for structuring events as a tree is to simplify the subscription mechanism. This mechanism will be explained in chapter 5.2.5, where we talk about the event handlers. Events can hold several different objects and attributes that are used to pass information from the monitored system to the clients.

An example of an event in the event tree is the `NewUserEvent` (see Figure 5.3). This event inherits everything held by the parent node. This is the date of creation that is included into the root event and the user information, such as name and address and so on. Additionally the

`NewUserEvent` carries a password. This event is supposed to be fired when a new user is added to the monitored system.

The next event to be added to this branch might be a `UpdateUserPasswordEvent`. It will carry the new password and some new information on the user.

5.2.2 Event Producer

The Lecando Enterprise Server includes many business logic modules that together form an e-Learning platform. Since the functionality is divided into different modules, every module should be able to produce events. Generally there are no restrictions on what kind of events every module is allowed to fire. Since these events are highly flexible, they should be used with precaution.

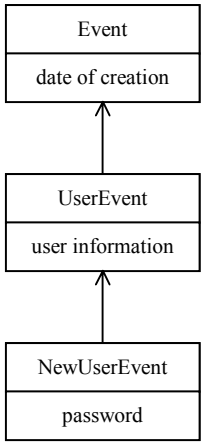

Figure 5.3 Example of one branch in the event tree. The Event is the root of the tree. The UserEvent inherits the Event class and adds user information. A password is added by the NewUserEvent class to the existing object of the UserEvent class.

One of the main tasks in designing the Event Service is to make the process of firing an event as lightweight as possible for the monitored system. The firing event mechanism will be divided into three parts: first an event is produced, second an event is packed into a message and third the message is sent to the Event Service. The reason for packing events into messages is that we want to use the Java Message Service. We will send events inside a message and to this message we will attach information about the JMS destination. This additional information has semantically nothing to do with the event itself, therefore we use messages as the transportation object instead of the events. In the next section you will discover how the JMS technology is used inside the Event Service.

5.2.3 Java Message Service

Java Message Service is included in the Event Service because it is possible to communicate asynchronously between two parties. This is why the monitored system will not be occupied too long while firing and sending an event. As mentioned in chapter 4.3, there are two ways to send messages with JMS. Here we choose the topic method, since it seems to be more flexible when adding more functionality to the Event Service. Topics are not more complicated than queues which is an advantage and that is why it is used here. We will choose to have one topic with a single subject. Every message received by the topic is sent further to a consumer that is subscribing to the topic. Again, normally there are many consumers of a topic, but in

this system only one will be used. Next you will see how the consumer distribute the messages to the event handlers that have subscribed to particular events.

5.2.4 Message Listener

The Message Listener's duty is to receive messages from the JMS, to filter events from the messages, to look up the event handler(s) that subscribe to the event, and to send the event to the event handler(s) for processing.

The Message Listener is a subscriber of a JMS topic and will be activated on arrival of a message to it. Every message will be processed by a Message Listener. If a Message Listener is occupied with processing another message, while a new message arrives from the JMS, a new instance of the Message Listener will be invoked, receive the message and processe it. Details about this mechanism will be explained in chapter 5.3.6.

When an event is uncovered from the message, its event handlers are looked up in a file. Every event must have at least one event handler that is subscribing to it. There is no restriction on how many different event handlers could receive the same event. The subscription file includes event and event handler pairs. Every event will be listed with all event handlers that subscribe to it.

Every found handler will be instantiated and executed by the Message Listener. This is further explained in chapter 5.3.6, were we describe the implementation details.

5.2.5 Event Handler

The power of the Event Service lies in its event handlers. Event handlers could have many different tasks. The main duty are to react on the reception of events and process them in the way it is defined in the handlers. There are some limitations about what is allowed to do and what is not. These limitations are the same as discussed in chapter 4.1, where we talked about the application server provider. Event handlers are added easily to the system. They can receive one or more different events. There is also no restriction on the number of different events they might subscribe to. As seen in Figure 5.2, the event handler is called from the Message Listener or the Scheduler. The Scheduler and its tasks will be described in the next chapter. Every event handler needs to subscribe to events. This is done, as described in the last chapter, with the event properties file.

Every event handler can use several services when processing events. They could execute events instantly or delay them for later execution. If events are sent to an event handler that processes events, e.g. once a week, then all events arriving within this time period have to be stored for later processing. This means that event handlers will process them later. The event handlers can be stateful, which means they can save their processing state for later execution. For this purpose, the event handlers may save their last execution time at the Event Storage and fetch it from there later when necessary. In the next section we will describe the duty of the Event Storage. See also Figure 5.2 for an overview. Properties can be saved at the Event Storage as well. They are always saved as key value pairs that could be of any type. Delayed events are sent with a timestamp to the Event Storage where they are collected by event handler name and processing time. It is the Scheduler that will instantiate the event handlers when they should process events for those who are stored at the Event Storage.

With the possibility of delaying events and saving its properties, the event handlers have all they need for error handling. Events that fail to process can be delayed. How many times an event handler should try to reprocess failed events can be saved and retrieved in the event storage. This is useful for the implementation of error strategies.

One of the main tasks while designing the framework for the event handler was to give the developers of the handlers as much flexibility as possible. In chapter 5.3.7, you will see how developers can use the event handler framework when customizing new event handlers. The variety of such handlers is huge. Sending an email with user information, login, and password to every new user when added to the system, could be one possible task. For this purpose we would add the NewUserEvent, described in chapter 5.2.1, to the system. Next step would be to create a new handler that produces an email holding the user information. The Event Storage is only used if an email could not be send to its receiver. This means that every incoming event is processed on arrival without delay. Later you will find more examples for event handlers.

5.2.6 Event Storage

The Event Storage is used by the event handlers and the Scheduler (see Figure 5.2). As mentioned above, the event handler is responsible for delaying events and the Scheduler for checking the processing times of all delayed events and initiating the event handlers with the events waiting to be processed.

The Event Storage collects all events in different jobs. A job is used to collect events that should be processed by the same event handler at the same time. Every job holds also only events for the same event handler with all events belonging to this handler that have a processing time within one minute. This means that every handler could have many jobs that are saved at the Event Storage. Every job could include many events, because an event handler could process events once a day and every event that reaches this handler before that time, will get delayed until than. A job is deleted whenever the Scheduler activates the event handler for this job and send all information that the job is holding to the handler. The Event Storage is persistent and recovers from system failure. This means no events are lost whenever they were saved. The service of the Event Storage will be described in detail in chapter 5.3.8.

5.2.7 Scheduler

The Scheduler is responsible for checking the Event Storage if jobs are ready to be run by its event handlers. These jobs will be passed from the storage to the Scheduler. Next the event handler, as mentioned above, will be instantiated with a collection of the events that were found inside the job. After that is the job deleted from the Event Storage. Every job found by the Scheduler will be processed by the handlers. If no job is found, the Scheduler will be terminated. This process is repeated every minute. The Scheduler is activated every minute by an external application.

5.3 Implementation of the Event Service

This chapter describes how the Event Service is implemented. An overview of the Event Service is illustrated below in Figure 5.2. The following subchapters will explain how the components interact with each other.

5.3.1 Packages of the Event Service

The Event Service contains five packages listed below.

```
com.lektor.events
com.lektor.events.beans
com.lektor.events.handlers
com.lektor.events.servlets
com.lektor.events.events
```

Events added to the Event Service is handled by the events package. Handlers in the handler package. The Message Listener is as described below an enterprise bean and therefore put into the beans package. The Event Storage and the Event Wrapper is added to the bean package also.

In the following sections you will find out which classes are used to build the components from Figure 5.2, where they are located and how they are used by the event handler developers. An example of how to use the Event Service is added at the end of this chapter.

5.3.2 Events

Events are used to pass information from the monitored system to external systems. Every event is a class with its own objects and methods. All `Event` classes are included into the `com.lektor.events.events` package. The top class in this package is the `Event` class. Figure 5.4 presents the event structure. Every new class added to the Event Service should inherit from this class or its sub classes. This will create a tree structure of all event classes. The event objects have to be serializable, since they are passed around between enterprise beans inside the EJB container of the monitored system. The `Event` class just holds a `creationDate` object. Every sub event class holds additional objects.

Every event class has, excepts its constructor, at least two more methods. These are the `equals` and the `hashCode` method. They are used to store and retrieve events by the Event Storage. Furthermore these two methods must be overwritten by each sub class to make the events distinguishable. In the next paragraph you will see how a new event class is added to the event tree.

The `Event` class is inherited by the `UserEvent` class. This new class holds an user object that includes all data of the user such as first name, last name, etc. The `UserEvent` class could be used every time the user information is updated in the monitored system. When an user saves his/her new data by using the save button, a `UserEvent` will be fired. An example of how this is done is described in the next section. Now the `NewUserEvent` class that inherits the `UserEvent` class is introduced. The `NewUserEvent` could be used every time a new user is added to the monitored system. This new event has, compared to its superclass, an additional password parameter, see Figure 5.4 for an illustration.

5.3.3 Monitored system

In this section you will find out how events are fired and published. A monitored system, in our case the Lecando Enterprise Server, is built upon many modules that together comprise a platform. Such a module consists of one or more enterprise beans that are responsible of

computing one specific task. When the application developer wants to transfer information from the monitored system to external systems, the usage of the Event Service will solve this task. The application developer of the monitored system must remember that publishing events without any subscription on it will give the data transfer unnecessary overhead.

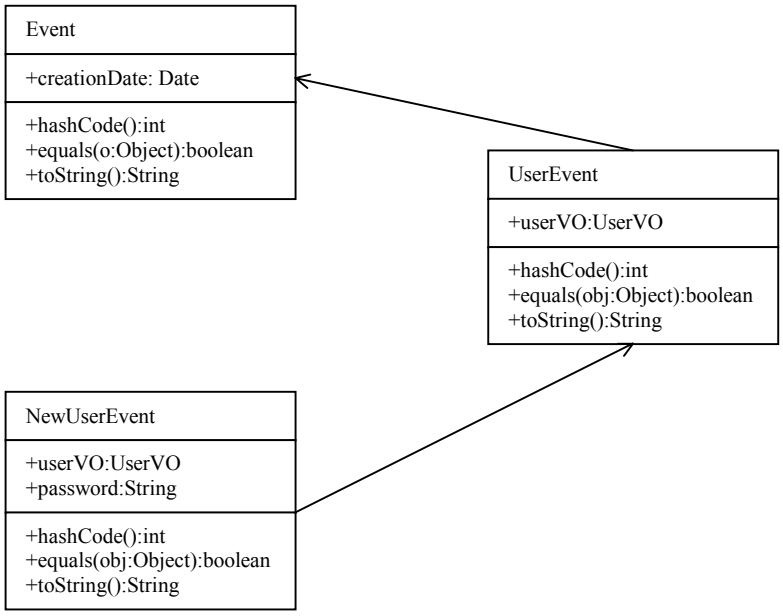

Figure 5.4 One branch of the event tree. The root holds the creationDate object. The UserEvent additionally the UserVO object and the NewUserEvent class adds a password to the UserVO of the UserEvent class. Note: + means public method.

Events are objects that carry information. Events are created by instantiating one of the event classes, e.g. a new `UserEvent` will be created by calling the constructor of the `UserEvent` class with the parameter `UserVO`. An `UserVO` is an object containing user information such as name and address. The construction of an event can be performed in the following way:

```
UserEvent userEvent = new UserEvent(UserVO userVO);
```

The `userEvent` object is sent with the `publish` method of the `MessageSender` class.

As soon as the Message Sender delivers the message to the JMS topic, the application will not be blocked anymore and continues execution. The publishing of an event can be performed as follows:

```
publish(userEvent);
```

The monitored system does not know who will receive the event. It is transparent to the monitored system. Events are fired when the user of the monitored system triggers them. This could be done when a button in the user interface is pressed or some sort of interaction between the user and the interface takes place. Next you can see how the Message Sender is publishing messages to the JMS topic.

5.3.4 MessageSender

The `MessageSender` class (`com.lektor.events package`) is used by the monitored system to send messages to a JMS topic. `LecandoTopic` is the name of the topic used together with the monitored system.

Furthermore this class is used to make the JMS communication transparent to the application developer since the JMS setup is hidden in the `publish` method. The object of this method needs to be serializable because of the need for transfer of it over a network.

```
public static void publish(Serializable object);
```

The following steps are executed by the `publish` method; in chronological order:

- A `TopicConnection` to the JMS provider is created.
- A `TopicSession` is started.
- An `ObjectMessage` is created.
- The `LecandoTopic` is looked up.
- A `TopicPublisher` for the session is created.
- The event object is added to the `ObjectMessage`.
- The message is send by the `TopicPublisher`.
- The `TopicPublisher`, the session and the connection is closed.

5.3.5 JMS server

The JMS server, normally referred to as the JMS provider, receives messages at its topics. The subscribers of every topic will receive every message that arrives to the specified topic. In our case we only have one topic, named `LecandoTopic`. Our subscriber is the Message Listener that will receive every message. We want that the monitored system and the Message Listener to communicate asynchronously. The only way to do this with applications that are based on J2EE is to use JMS.

The JMS server is fault tolerant and persistent. This means that when a message can not be delivered to the `MessageListener` right away, the JMS server will try again until it succeeds. Every message that reaches the JMS server will be kept in the storage of the JMS until it is delivered to all its subscribers.

The setup of the JMS server is different for every JMS provider. Since Lecando Enterprise Server uses the Orion Application Server [20], we use the JMS implementation of Orion. The Orion Application Server follows the J2EE specifications. If the monitored system uses another application server, the setup could be different but since everything in the Event Service compatible to the specifications, it should not be a problem. In the Orion application server, the JMS server is set up with a file named “jms.xml”, that is used to specify the names of the topics and its subscribers. More details about how to setup the server can be found in the Orion documentation [20].

5.3.6 MessageListener

The `MessageListener` class in the `com.lektor.events.beans` package is responsible for identifying the incoming messages, to locate the event handlers of the identified events, to instantiate these event handlers, and to call the `process` method of every found event handler.

The `MessageListener` is a message-driven bean that receives all messages from the JMS server. The key method `onMessage` is called by the JMS server. The method code looks like this:

```
public void onMessage(Message message) {…}
```

As a bean developer you must only provide the bean class with this method in accordance to the EJB2.0 specifications [5]. How and when this method is called is transparent to the developer. Below you find a detailed description about what this method will do in our case.

As mentioned above, the message received from the JMS server, must be identified as an event. First the object of the JMS message has to be retrieved. As you know from chapter 5.3.2, events were added to the `ObjectMessage`. The object is retrieved from the message with the following line:

```
Object object = ((ObjectMessage) message).getObject();
```

If the object is an instance of the `Event` class, the event handlers for this event are fetched. For this purpose the `getEventHandlers` method of the `EventConfiguration` class will look for the event handlers that are listed as subscribers of the event in the `event.properties` file. An entry in this file can look like this:

```
com.lektor.events.NewUserEvent = com.lektor.events.handlers.NewUserMailer
```

On the left-hand side the complete path of the event is used as a key and on the right-hand side the complete path of the event handler is used as the value. This method returns a collection of all event handlers found for the specified event to the Message Listener. The event handlers are already instantiated and ready for execution. Every event handler will be executed sequentially until every handler is terminated. Eventually the `process` method of every event handler is called with the event itself as a parameter. See the next section for details.

5.3.7 Event Handler

The `EventHandler` class located in the `com.lektor.events.handlers` package is an interface that every customized handler must implement. The event handler developer could use the methods of the abstract `EventHandler` class as shown in Figure 5.5. A description of how these methods can be of use follows below.

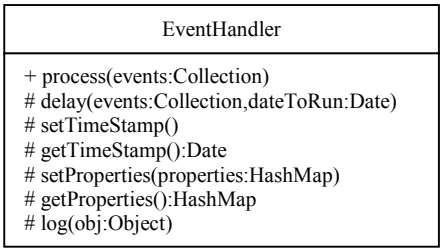

Figure 5.5 The EventHandler class and its methods that can be used by all handlers added to the Event Service. Note: + means public and # protected method.

The abstract `process` method must be implemented by every handler. This method is called by the Message Listener and takes as parameter a `Collection` of one or more events. This method is used to define the action that should be taken when an event is received.

The `delay` method sends a `Collection` of one or more events to the Event Storage. The parameter `dateToRun` is used by the Scheduler to get all events that are ready to be processed by the event handlers.

The method `setTimeStamp` is used to save the current time of execution of the event handler at the Event Storage. The `getTimeStamp` is used to retrieve the time that the `setTimeStamp` method saved during the last executed.

Every event handler implementation can save data of its own choice at the Event Storage. How often a particular event should be reprocessed when failures occur, could be one example of such a property. The method to do this is `setProperties, getProperties` `re`trieves the property from the storage.

The `log` method is used for logging the action an event handler is handling. Every time this method is called, the object parameter is appended to a file. Logging should be used with great care to avoid performance degradation and failure due to loss of free space on the storage media.

Above we described the methods that the `EventHandler` class provides to the developers of the event handlers. Next you will find out what happens if the event handler delays an event.

5.3.8 Event Storage

The Event Storage is used by the event handlers and the scheduler to store and retrieve events. We will describe how the Event Storage collects events into jobs and how they are stored by it. Additionally the event storage is also used to save information about event handlers.

The `EventStorage` class has two more classes that are used to give the event handlers and scheduler the services they need. For an overview see Figure 5.6. The Event Storage is an entity bean with its persistence managed by the bean container. This means that for the developer, the maintenance of the database is transparent. It is the enterprise bean container that creates, updates and searches tables for data. Furthermore is the Event Storage a singleton bean, this means that the entity bean only exists in one instance. This was chosen to eliminate the risk for critical sections and other similar problems. In the next few paragraphs we will explain the methods and data types of the `EventStorage` class.

The `getTimeStamp` and `setTimeStamp` methods are called from the event handlers as described above. The data type `timeStampsByHandlerName` is a `HashMap` and used by the set and get methods. This `HashMap` uses the event handler name as key and the timestamp as value.

The `getProperties` and `setProperties` methods are also called from the event handlers. They are operating on the `propertiesByHandlerName` of type `HashMap` that is similar to the `timeStampsByHandlerName` also a `HashMap`. Every event handler has its own defined properties stored in the Event Storage.

A job in this context is used to keep events that are delayed by the same event handler together if the `dateToRun` parameter of them are within one minute. This means that every event is kept by one job. Event handlers could have many jobs with events. As you can see from Figure 5.6, a job keeps an `ArrayList` with all event ids, the event handler name and the date when to execute this job, which is the same as the first event added to it. The methods used by the job, except for the constructor, are the `equals`, `compareTo`, `createEventWrapper`, `getEventIds`, `getEvents`, and the `destroyEvents` methods. The `equals` and the `compareTo` methods are used when the Event Storage looks for the right job to save a new event that should be delayed. When an event is passed to a job, it will be put into an entity bean called `EventWrapper` by the job. These entity beans manage the persistence for the events automatically. The job will get a handle with the event ids after calling the `createEventWrapper` method. This method is called as soon a new event arrives at the job. The `getEventIds` method just returns a `Collection` of all event ids that are saved by the job. The `getEvents` method also return a `Collection` of all event instances, which are fetched from the Event Wrappers. The methods that are used by the job for this purpose are the `getId` and `getEvent` methods. The `destroy` method of the job, removes all entity beans that are found inside a job.

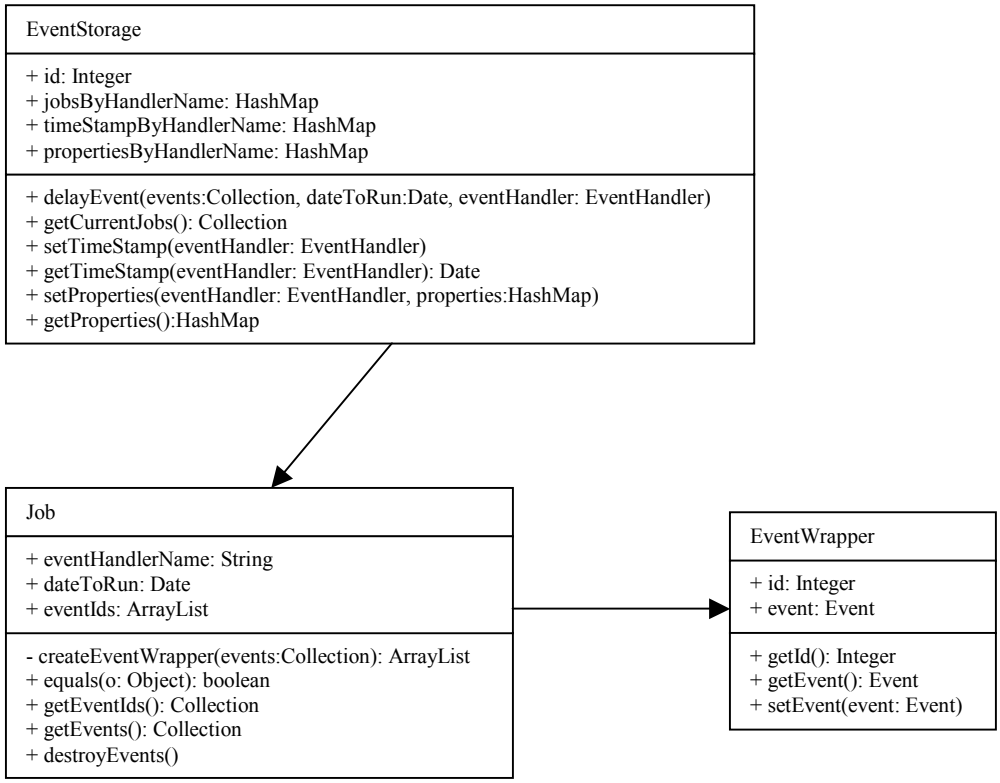

Figure 5.6 The Event Storage collects events that are stored in enterprise beans and into jobs. A job has a date to run and a list with all events that are included inside the job. Every event is stored with its id and event object inside the EventWrapper. The EventWrapper is an entity bean that stores the event into the DBMS. Note: + means public and – means private method.

The `delayEvent` method of the Event Storage looks for an existing job and if one is not found, creates a new. After that the event will be added to the job. As mentioned above, an EventWrapper is created for every new event added to a job and the `HashMap jobsByHandlerName` is updated. Remember that events get delayed only by event handlers.

The Scheduler is responsible to check the Event Storage for jobs that have to be executed. This is done by calling the `getCurrentJobs` method of the Event Storage.

The `getCurrentJobs` method is called by the scheduler. This method looks through the `HashMap` called `jobsByHandlerName` and returns all jobs that are ready to be processed to the scheduler. After that the job is removed from the Event Storage.

5.3.9 Scheduler

The Scheduler's responsibility is to look for jobs that are ready to be executed. The jobs that are found are then send to its event handlers where they get processed. Every minute is the Scheduler executed by a process of the operating system called `clock`.

First the scheduler calls the `getCurrentJobs` method of the Event Storage. As mentioned a `Collection` with the found jobs are returned. For every found job, the event handler is located and instantiated. After that the events are fetched from the job and passed to the event handler where they get processed.

The Scheduler is implemented as a servlet, because it can be called from outside of the application server.

5.4 Example of how to use the Event Service

The Event Service designed above can be used for propagating events that happened in one system to others. This chapter presents an example of typical usage of the Event Service. In this example the monitored system will fire predefined events and send them to the JMS server. A client (event handler) is added to the Event Service that will subscribe to this type of events. Every incoming event will be delayed by the event handler until midnight every day when an email will be sent to the administrator with all event's data included. Figure 5.7 shows all components involved in this example.

The `NewUserEvent` is used to produce and fire an event. The `NewUserMailer` is the handler that will delay every incoming event until midnight every day. This means that every event sent before midnight will be added to the same job at the Event Storage. The Scheduler will receive this job when executed at midnight or shortly thereafter. In our setup the Scheduler will be executed every minute. This is of course changeable if necessary by shortening the wait circle of the operating systems process that is responsible to call the Scheduler every minute. After the event handler receives the job from the Scheduler, it will send an email to the administrator of the Lecando Enterprise Server with all new user data.

Figure 5.7 shows only the most important methods that are used in the example. All other methods have been discussed in chapter 5.3.

At the appropriate place inside the monitored system we fire the new event by first calling the constructor of the `NewUserEvent` class with the `UserVO` object and a `password`.

```
NewUserEvent newUser = new NewUserEvent(userVO, passwd);
```

Next we call the `publish` method of the Message Sender.

```
publish(newUser);
```

The Message Sender's `publish` method communicate with the JMS server. What it does is basically packing the event object into a message and sending it to the LecandoTopic at the JMS server. See chapter 5.3.4 for more detailed description.

Since the Event Service only has one subscriber for all messages from the JMS server, our `NewUser` message will be sent to the Message Listener. This is done by the EJB container which invokes the `onMessage` method of the Message Listener every time a new message arrives there. If the Message Listener is occupied with processing a message while a new message arrives at the JMS server, a new instance of the Message Listener is invoked. How many different instances at the same time that can reside at the same time depends on the application server provider. In the next chapter we will test the Event Service and will see with what frequency messages can be processed.

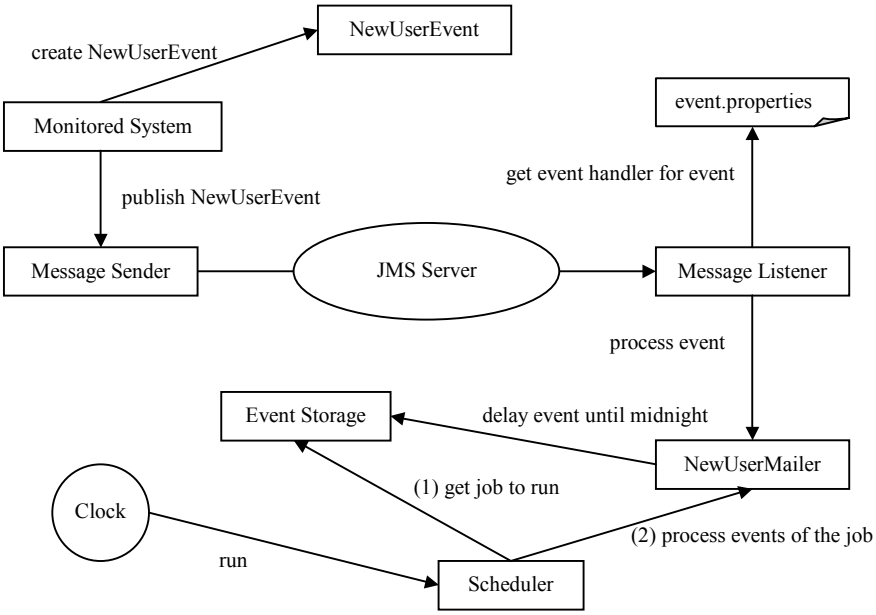

Figure 5.7 Component overview for the NewUserMailer with the NewUserEvent. The monitored system uses the MessageSender class to send the NewUserEvent to the JMS server. The JMS server distributes the event to the MessageListener. The Message Listener looks for the EventHandlers of the NewUserEvent. The NewUserMailer is registered as a subscriber to the NewUserEvent in the event.properties file. The Message Listener finds the NewUserMailer, executes it and delays all events until midnight with help of the EventStorage class. An external clock will trigger execution of the Scheduler once a minute. When the Scheduler finds out that midnight has passed, the NewUserMailer will get all jobs with the events that are ready to be processed.

After the Message Listener receives the message, it will search for the event handlers that have subscribed to the event and include them in the message. We will name our handler `NewUserMailer`. Since this handler should receive the `NewUserEvent`, we have to add the following lines to the `event.properties` file:

```
com.lektor.events.events. NewUserEvent  =
com.lektor.events.handler.NewUserMailer
```

When found, the `process` method of the `NewUserMailer` will be called with the `NewUserEvent`. The `NewUserMailer` will only implement the abstract `process` method of the `EventHandler` class. This method just calls the `delay` method with the `NewUserEvent` and the time representation of midnight.

```
public void process(Collection events) {
                    . . .
           delay(events, midnight.getTime());
                    . . .
}
```

The `delay` method locates the Event Storage and calls its `delayEvent` method. See chapter 5.3.8 for details. Since all events will have the same `dateToRun` parameter, they all will be kept in the same job. When the first event is received, a new job is created and every new event is added to the existing job.

The `NewUserMailer` will have one additional method that will be called when the Scheduler calls the `process` method of the `NewUserMailer`. We name this method `sendMail`. This method receives all events of one job from the Scheduler. We omit the details of this method since it is of particular interest precisely how the email is created and the events are appended to it.

The Scheduler is in our case executed once a minute. At midnight the Scheduler will get all events from the job when calling the `getCurrentJobs` method of the Event Storage. Next, as mentioned above, the `process` method of the `NewUserMailer` is called. And from there the `sendMail` method.

As you have seen above, there are four things to do when extending the Event Service with additional functionality.

- First, the event that is causing the action has to be added to the event package if it does not exist already.
- Second, at the monitored system the event has to be created and published with the method of the Message Sender.
- Third, a new handler is added.
- Finally, the event handler has to subscribe the new event which is done in the `event.properties` file.

6 Performance Evaluation of the Event Service

The performance of the Event Service is measured as the event processing time, defined in chapter 6.1. The experiments are performed for different values of input parameters, which are: The intensity of the fired events and the number of subscribing handlers to these events. The output parameters are the different latencies for the event processing time. It is not easy to draw conclusions from the results of the experiments, since there are no reference times to compare with. The possible throughput of events in the Event Service is important and from the results conclusions can be drawn regarding performance of the Event Service.

6.1 Evaluation Technique

First the event processing time is defined and second the realization of the experiment are shown in the following two sections. It is important to find out how long time it takes to process events. Events are fired from the monitoring system. The Lecando Enterprise Server produces events and sends them to the JMS server with help of the `publish` method of the Message Listener at a time indicated by **t1** in the timing diagram depicted in Figure 6.1.

The `publish` method, as described in chapter 5.3.4, establishes a connection and a session to the JMS server. After that, the publisher sends the event to the JMS server and when the event is received there, the connection and the session are closed. The `publish` method returns when these steps are finished.

The second measurement, **t2**, is the time interval the Message Listener needs to locate the handlers for the received event and executes them. After all handlers are processed, the events and the Message Listener return. It can be noticed that the time periods **t1** and **t2** overlap. The reason for this is that the `publish` method of the Message Sender takes more time to close the session and connection, than the internal process of the JMS server for sending the event from the topic to the MessageListener.

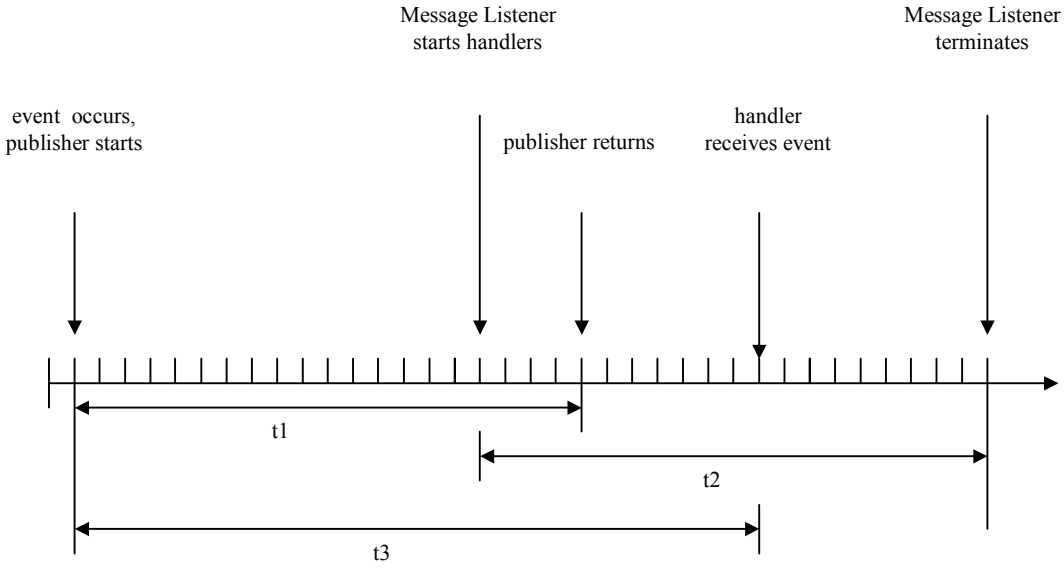

Figure 6.1. Definition of the processing times in the Event Service.

The third measurement, **t3**, is the time an event needs from being created until it is delivered to the handler that subscribes the event. The three latencies are the output parameters of the experiments.

As mentioned in the beginning of this chapter, the event intensity, the number of monitored systems and the number of event handlers that subscribe to the events are the input parameters. The experiments are done by creating new event classes and adding them to the event package. Instead of using the web interface of the Lecando Enterprise Server, a new test class will be added to the server. The `TesterServlet` class fires events every time it is called, with an intensity we define in it. Next, event handlers are created and added to the Event Service.

The three experiments are depicted in the following figures. The first illustrates the first set of experiments with one source system that fires one type of event and one event handler that fetches these events.

Figure 6.2 The experiment with one source system, one event type and one event handler .

The second figure depicts the experiment with one source system and three different event types that are received by three different event handlers.

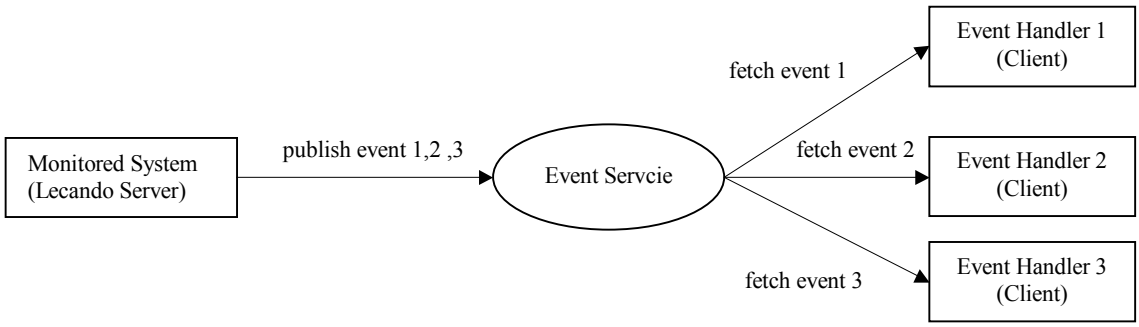

Figure 6.3 The experiment with one source system, three different event types and three different handlers.

The third experiment uses three source systems. Each of them produce one different event. Every event is received by a different event handler.

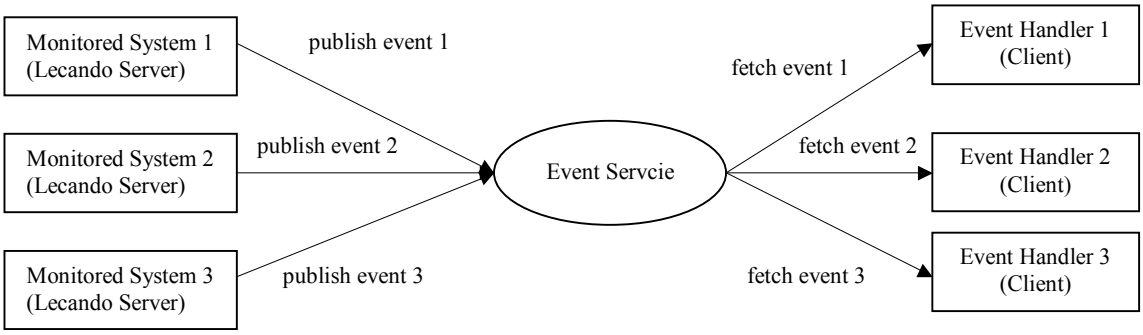

Figure 6.4 The experiment with three source systems, three different event types and three different handlers.

Next a detailed description about how the experiments are realised by modifying the Lecando Enterprise Server and the Event Service is explained.

The `TesterEvent` is an event that is used for the experiments. It inherits the `creationDate` from the `Event` class and has an id for the identification of the events. Two more event classes are added that are used in the second test case. Below, parts of the `TesterServlet` code is

shown. In the `for` loop the intensity for producing and publishing events is defined. The time is captured before and after the event firing process. The latency **t1** is written to a log file.

```
for(int i=1; i<=15; i++) {
    Date startTime = new Date();
    try {
        TesterEvent testerEvent =
               new TesterEvent(i, newDate(startTime.getTime()));
        MessageSender.publish(testerEvent);
    } catch (JMSException e) {
               . . .
    } catch (NamingException e) {
               . . .
    }
    Date endTime = new Date();
    Log.getInstance(this).info((endTime.getTime()-
               startTime.getTime())+ " T1");
}
```

The `MessageListener` class is modified to capture the latency **t2**. This is the time the event handlers need to process the received event. See below for details.

```
public void onMessage(Message message) {
    Date receivedTime = new Date();
    try {
         . . .
         eventHandler.process(eventCollection);
         . . .
    } catch (JMSException e) {
        throw new RuntimeException("Nested exception is: " + e);
    }
    Date finishedTime = new Date();
    Log.getInstance(this).info((finishedTime.getTime()-
               receivedTime.getTime())+ " T2");
}
```

The time **t2** of the `onMessage` method is sent to a log file as well. The `TesterHandler` class used for the experiments calculates the duration from creating the event until it arrives at the handler and writes an entry in the log file. This is **t3** in Figure 6.1. To capture the time **t3**, the Message Listener is modified as follows:

```
public void process(Collection events) {
    Date receivedTime = new Date();
    Iterator eventsIterator = events.iterator();
    while (eventsIterator.hasNext()) {
        Event event = (Event) eventsIterator.next();
        if (event instanceof TesterEvent) {
            TesterEvent1 e = (TesterEvent) event;
            Log.getInstance(this).info((receivedTime.getTime()-
                e.created.getTime())+" T3");
         } else {
            throw new RuntimeException("TesterHandler only accepts
                 TesterEvent's");
         }
    }
}
```

6.1.1 Test bed and Performance Experiments

All experiments are done on a workstation with an Intel Pentium 3 with a 700 Mhz processor and 128 Mb memory. The operating system is Linux RedHat 7.0. The modified Lecando Enterprise Server is used with the Orion Application Server 1.38 [20]. In the test environment, apart from the handling of events the Lecando Enterprise Server is idle, i.e. no external workload can interfere with the test measurements

Three different sets of experiments are performed, as listed below.

- The experiment with one source system, one event type and one event handler (see Figure 6.2).
- The experiment with one source system, three different event types and three different handlers (see Figure 6.3).
- The experiment with three source systems, three different event types and three different handlers (see Figure 6.4).

First the `TesterServlet` fires the `TesterEvent`. An external process calls this servlet once a second. The `TesterServlet` will produce 1, 5, 10, 15, 25, 50, 100 or 500 events every time it is called. This means that eight experiments are needed. One `TesterHandler` that subscribes the `TesterEvent` will receive the produced events.

The second setup uses three events: `TesterEvent1`, `TesterEvent2`, `TesterEvent3`. The `TesterServlet` fires every event once inside the loop. This means that if the loop is executed 10 times, 30 events are produced. Since the `TesterServlet` is called every second, the experiment produces 30 events per second. The events are the same except their names. Each of the three events are subscribed by a different handler. The handlers are called `Testerhandler1`, `Testerhandler2`, `Testerhandler3`. The `event.properties` file, with the subscription table, looks like the following:

```
com.lektor.events.TesterEvent1 =
com.lektor.events.handlers.TesterHanlder1
com.lektor.events.TesterEvent2 =
com.lektor.events.handlers.TesterHanlder2
com.lektor.events.TesterEvent3 =
com.lektor.events.handlers.TesterHanlder3
```

The handler will not delay events, since is does not effect the system where the events are produced.

In the third setup, three different monitoring systems are producing events. Every system fires one unique event every time the TesterServlet is called. There are three different handlers that receive one of the events each. The same event intensities as in the second experiments are used.

6.2 Performance Results

Three sets, with one experiment for every frequency, are tested. A few entries from the log file while running the first setup with a frequency of 10 events per seconds are listed below. Notice that the entries are very small. Many events take less than 1 ms to be produced and published, and since the system clock returns number of integers some of the latency measured are zero.

```
Date:      Time:         Latency (ms):
2001-03-16 15:43:12,467 T1 = 121
2001-03-16 15:43:12,497 T3 = 425
2001-03-16 15:43:12,498 T2 = 70
2001-03-16 15:43:12,514 T3 = 2
2001-03-16 15:43:12,514 T2 = 1
2001-03-16 15:43:12,515 T1 = 3
2001-03-16 15:43:12,517 T3 = 2
2001-03-16 15:43:12,517 T2 = 1
2001-03-16 15:43:12,518 T1 = 0
2001-03-16 15:43:12,519 T3 = 1
2001-03-16 15:43:12,520 T2 = 2
2001-03-16 15:43:12,520 T1 = 1
```

In Table 6.1, the results of the first setup are shown. The figures in the rows for the latencies **t1**, **t2** and **t3** are average values for each experiment. Every column represents a different test run with different event firing frequencies. The last row shows the number of events that are fired by the `TesterServlet` and received by the `TesterHandler` during each run.

	Event intensity							
	1	5	10	15	25	50	100	500
T1 (ms)	1,78	1,80	2,00	1,81	1,88	1,85	1,88	2,10
T2 (ms)	1,58	1,50	1,06	1,21	1,14	1,18	1,17	1,41
T3 (ms)	4,32	3,10	1,84	2,33	1,56	1,55	1,30	1,53
Number of events	300	1765	2890	4335	7275	13350	37100	69500

Table 6.1 The results of the experiments with the first setup. One event and one event handler. The results are also depicted Figure 6.5.

In the column representing 15 fired events per second, a total of 4335 events are produced during the experiment. In average, every event took 1.81 ms to be produced and published by the `TesterServlet` (**t1**). The `MessageListener` (**t2**) needed 1.21 ms in average to locate the handler, execute it and wait until it is finished. The `TesterHandler` (**t3**) calculates how much time in average it takes for an event to be produced by the `TesterServlet` and received by it. This is 2.33 ms in the test case when 15 events per second were fired. Below the diagram for the first experiment is shown.

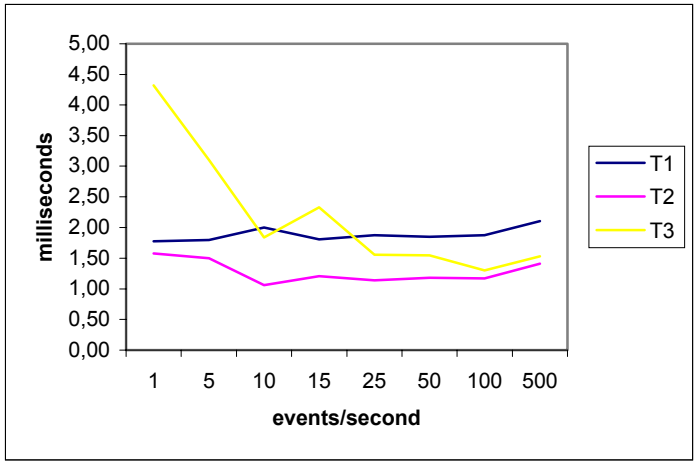

Figure 6.5 Result of the test run with one event type and one handler. Exact values are tabulated in Table 6.1

The second experiment is similar to the first. The results are shown in Table 6.2. The difference is that three different events are fired and published by the `TesterServlet` in every `for` loop.

	Event intensity						
	3	15	30	45	75	150	300
T1 (ms)	2,25	1,87	1,94	1,92	1,91	2,05	2,00
T2 (ms)	1,60	1,34	1,32	1,32	1,35	1,44	1,39
T3 (ms)	2,43	1,50	1,79	1,60	2,02	1,43	1,66
Number of events	939	4305	8370	12555	21525	43050	86190

Table 6.2 The test setup with 3 different events and three event handlers. The results are also depicted Figure 6.6

The time for one loop is measured. Since every event received by the JMS topic is send to a separate instance of the MessageListener, the processing time of every handler is measured. The TesterHandler measures this time (**t3**). The plot for the second experiment is depicted in Figure 6.6

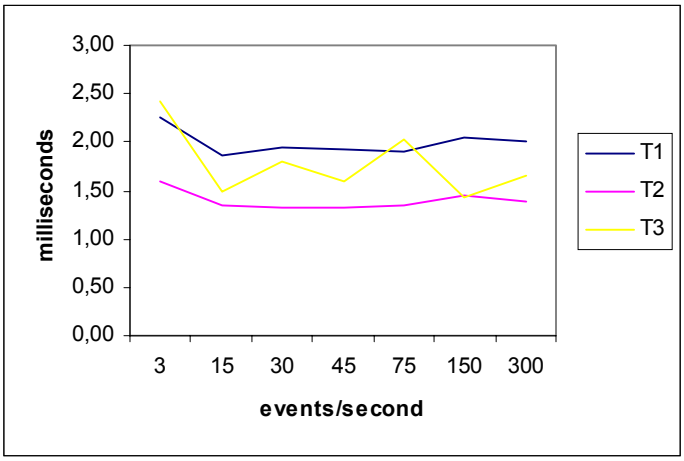

Figure 6.6 Result of the experiment with three event types and one handler for each event. Exact values are tabulated in Table 6.2

The third experiment shows similar results compared to the first two experiments. All three monitoring systems are firing events. The event intensity is the same as in the second experiment except that each of the three different events is fired by another system.

Table 6.3 shows the results of this experiment.

	Event intensity						
	3	15	30	45	75	150	300
T1 (ms)	2,73	2,52	2,07	1,99	2,00	1,87	2,12
T2 (ms)	1,59	1,38	1,21	1,06	1,17	1,11	1,37
T3 (ms)	2,09	1,78	1,64	1,54	1,52	1,46	1,63
Number of events	912	4560	9120	13680	22800	45600	91200

Table 6.3 The experiment with 3 different monitored systems, 3 different events and three event handlers. The results are also depicted Figure 6.7

The diagram of test results is shown below.

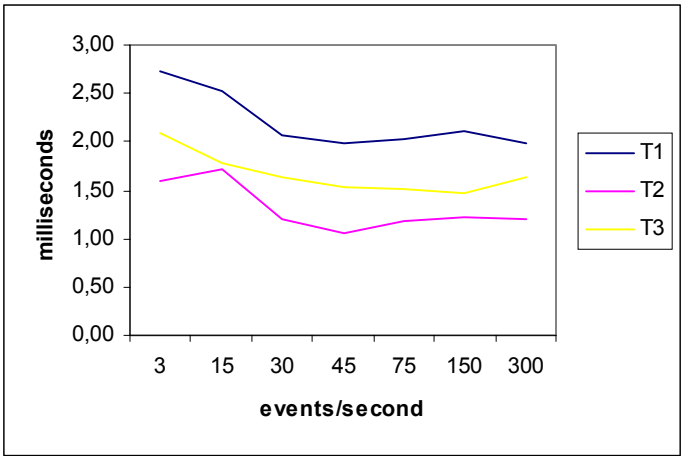

Figure 6.7 Result of the experiment with three monitored systems, three event types and one handler for each event. Exact values are tabulated in Table 6.3.

6.3 Discussion

All execution times in the three tables are approximations since it is not possible to capture time values more accurate. The log file shows that the TesterServlet sometimes executes faster than 1 ms, which leads to 0 ms in the log file. All latencies in the Table 6.1, Table 6.2 and Table 6.3 are approximations. In every experiment each of the latencies are added up to a total and divided by the number of events that are sent. Note that the first time the classes are called the execution time is higher. The first time, all classes are loaded into memory of the Orion Application Server. After that they are cached until they get garbage collected by the JVM (Java Virtual Machine).

The first set of experiments shows that the TesterServlet (**t1**) need about 2 ms to create a new event and publish it. The experiments are done with a frequency from 1 up to 500 events per second. This is satisfying since all events in the monitored system are supposed to be triggered by user interaction. The MessageListener (**t2**) will take between 1 and 2 ms to locate

the event handler and wait until it is terminated. The time is depending on the amount of work an event handler will perform. The amount of time the events are in the system shows much variation, but the standard deviation appears to be stable when the experiment last longer (see Figure 6.5). The reason for this is the start up, since the first instances of the classes takes time to be loaded and instantiated. Nonetheless the value is here also between 1 and 2 ms. One curiosity is that the value for the event firing process in the TesterServlet (**t1**) takes more time compared to the time the event needs from creation time until it is received by the TesterHandler (**t3**). The reason for this is that once a message arrives at the JMS server it is sent straight away to the MessageListener, but the `publish` method of the MessageSender still closes first the session and connection and this takes more time than sending the event to the handler.

In the second series of experiments, three different events are produced and published. Every event type is subscribed by a different handler. There might be situations when the monitored system needs to fire several events at once. This is tested in this experiment. In average it takes between 1 and 2 ms to create and publish events (see Figure 6.6). The MessageListener (**t2**) measures around the same execution times for the handlers as in the first setup.

The test results of the third experiment are similar compared to the second setup, since the same amount of events are fired, but from three sources instead of one.

The amount of time the events need to reach the handlers are almost the same between all series (see Figure 6.7). All experiments indicate that the Event Service satisfies the requirements stated in chapter 3.2. In this environment the user of the monitored system can not notice whether events are fired or not, since the sending latency (**t1**) is very short even with a high number of produced events.

7 Conclusions and future work

7.1 Summary

We have analysed several different approaches to design and develop an Event Service for enterprise applications. Therefore we have studied similar systems that can be categorized into two areas: Event Notification systems and Messaging Middleware systems. Commercial products were found mostly as Messaging Middleware and Event Notification Systems are developed basically only in research groups. We described the terms used in both areas and used them for comparison of such systems. After that, the Java™ 2 Enterprise Edition architecture was briefly explained and some Java APIs that are included in that standard were listed. Java™ Message Service and Enterprise JavaBeans™ were described in chapter 4.2 and chapter 4.3, since the Event Service was based upon those technologies. Next, the Event Service was designed and developed. Finally three different setup of experiments were used to evaluate the Event Service.

7.2 Conclusions

Our task was to develop an event sending framework that can be used by an enterprise application based on the J2EE standard architecture. The requirement for fast processing times for the monitored system limited the choice of usable technologies. From the comparison of the Event Notification systems and Messaging Middleware systems, we decided to implement the Event Service with an JMS implementation. Since the Lecando

Enterprise Server was build with the Orion Application Server, we used the JMS implementation of the Orion Application Server to implement the Event Service. Asynchronous communication between the monitored system was achieved by using JMS together with message-driven beans. The experiments done when the Event Service was complete, satisfied all requirements. The event processing time was short which is extremely important.

With help of the Event Service it is possible to monitor enterprise applications and propagate the occurrence of certain events to other systems. Those events can be delayed and collected for later usage. It is easy to add clients (event handlers) to the Event Service in order to create the link between the monitored systems and the system to be notified.

7.3 Suggestion for future work

The JMS implementation of the Orion Application Server 1.38 is not satisfying, since it is not possible for more than one message-driven bean to subscribe on the same topic. Different JMS implementations should be tested to find a better one, which will improve the mechanism for message-driven beans to subscribe on topics.

High availability is of great importance, and to assure this different approaches have to be studied in more detail. It is not unlikely that the availability of the Event Service can be improved. To make it easier for developers, the delayed event mechanism should be more fine grained, i.e. to make it easier and faster for handler developers to build their handlers.

8 References

[1] N. Kassem, Designing Enterprise Applications with Java™ 2 Platform, Enterprise Edition. Sun Microsystems, Inc. June 2000.

[2] D. Flanagan, J. Farley, W. Crawford and K. Magnusson, Java™ Enterprise in a Nutshell, O'Reilly & Associates, Inc. 1999.

[3] Java™ 2 Platform, Enterprise Edition Specification, Version 1.2. Sun Microsystems, Inc, 1999. http://java.sun.com/j2ee/

[4] R. Monson-Haefel, Enterprise JavaBeans™, Second Edition. O'Reilly & Associates, Inc, 2000.

[5] Enterprise JavaBeans™ Specification, Version 2.0, Proposed Final Draft, Sun Microsystems, Inc. October 2000. http://java.sun.com/products/ejb/

[6] Java™ Message Service Specifications, Version 1.0.2, Sun Microsystems, Inc, 1999. http://java.sun.com/products/jms/

[7] J. Wetherill, Messaging Systems and the Java™ Message Service, Article, Java World, 2000.

[8] J. Hunter and W. Crawford, Java™ Servlet Programming, O'Reilly & Associates, Inc. 1998.

[9] W. R. Stevens. TCP/IP Illustrated, Volume 1, The Protocols. Addison-Wesley, 1999.

[10] A. Carzaniga, D. Rosenblum, and A. Wolf, Interfaces and Algorithms for a Wide-Area Event Notification Service, Technical report, University of Colorado, May 2000.

[11] G. Liu and A. Mok, An Event Service Framework for Distributed Real-Time Systems, Technical report, University of Texas at Austin, 1998.

[12] Talarian Corporation, SmartSockets Overview, White Paper, 2000. http://www.talarian.com/products/smartsockets/

[13] Keryx, In Internet Notification Service, Hewlett Packard Laboratories, Bristol, UK. 1998.

[14] B. Segall and D. Arnold, Elvin: A Publish/Subscribe Notification Service with Quenching, Technical report, University of Queensland, Australia. 1997.

[15] TIB/Rendezvous: White Paper. TIBCO Software, Inc. 1994.

[16] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An information Flow-Based Approach to Message Brokering, IBM TJ Watson Research Center. http://www.research.ibm.com/gryphon/

[17] Ma C. and Bacon J., "COBEA: A CORBA-Based Event Architecture",Proc USENIX COOTS'98, pp117-131, Santa Fe, New Mexico, USA, April 1998.

[18] G. Cugola, E. Di Nitto, and A. Fuggetta, The JEDI event-based infrastructure and its application to the development of the OPPSS WFMS, Politecnico di Milano, Italy. 1998.

[19] S. Maffeis, iBus: The Java Intranet Software Bus, Technical report, SoftWired. Switzerland, 1997. http://www.softwired.ch/

[20] Orion Application Server™, http://www.orionserver.com.