An Operation and Maintenance Platform

Based on

Enterprise JavaBeans™

Hama Biglari

Information Technology
Computing Science Department
Uppsala University
Box 311
751 05 UPPSALA

This work
has been carried out at

Network Core Products
Ericsson Utvecklings AB
Element Management
Application Design

Supervisor: Måns Engstedt, Ericsson Utvecklings AB, Älvsjö
Examiner: Richard Carlsson
Passed:
Abstract

Enterprise JavaBeans™ (EJB) is an architecture specified by Sun Microsystems and published for the first time in March 1998. The EJB specification, which is based on the Java platform and part of Java 2 Enterprise Edition (J2EE), is a framework for server-side component-based development. EJB is used for building server-side systems that are scalable, reliable, secure, and portable. The EJB specification also defines a set of services that must be provided by EJB servers (and EJB containers). Some of the main services that are provided to the server-side EJB components are: transaction management, persistence, security, concurrency and instance pooling.

In the element management applications and tools that are in use at Ericsson, and also in other telecommunication areas, there is often great need for standard mechanisms and tools that can increase reusability and maintenance of software components, and that provide services such as transaction management, security and persistence, so that the application developers will be able to focus on business logic instead of underlying system properties. It is also important that middle-ware systems and client/servers systems (and the corresponding element management applications and tools) are portable, scalable, reliable and reasonably efficient.

A typical platform for operation and maintenance in the element management area would be based on asynchronous communication, callbacks and concurrent transactions. Therefore, the application that the EJB-based platform has been designed and implemented for (in this work) is an alarm handling application with notification and delivery of spontaneous alarms, concurrent transactions on alarms and storage of alarms in databases. Several design alternatives have been investigated and implemented, and different transaction modes have been tried out in various scenarios where portability, scalability and reliability of the EJB-based platform is discussed.
CONTENTS

About this paper ........................................................................................................... 6
Introduction ........................................................................................................... 7
Readers’ Guide .......................................................................................................... 13

1. Enterprise JavaBeans™ (EJB) ................................................................................. 15
   1.1 Why EJB ........................................................................................................ 23
   1.2 EJB Architecture ............................................................................................. 27
      1.2.1 EJB Client ................................................................................................ 27
      1.2.1.1 RMI Clients ....................................................................................... 30
      1.2.2.2 CORBA Clients ............................................................................... 33
      1.2.2.3 RMI over IIOP ................................................................................ 35
      1.2.2 EJB Container ....................................................................................... 35
      1.2.2.1 Support for Transactions .................................................................. 35
      1.2.2.2 Support for Management of Multiple Instances .............................. 42
      1.2.2.3 Support for Persistence ..................................................................... 45
      1.2.2.4 Support for Security ......................................................................... 47
      1.2.3 EJB Server ............................................................................................... 48
      1.2.4 Interfaces ................................................................................................. 48
      1.2.4.1 The EnterpriseBean Interface ........................................................... 48
      1.2.4.2 The Home Interface .......................................................................... 49
      1.2.4.3 The Remote Interface ....................................................................... 54
      1.2.4.4 The EJBContext Interface ................................................................ 57
      1.2.4.5 The SessionContext Interface ........................................................... 59
      1.2.4.6 The EntityContext Interface ............................................................. 61
      1.2.5 EJB Beans ................................................................................................. 62
      1.2.5.1 Session Beans ..................................................................................... 62
      1.2.5.2 Entity Beans ......................................................................................... 66
      1.2.5.3 Restrictions on EJB beans ................................................................. 72
      1.2.6 EJB Roles .................................................................................................. 73
      1.2.7 EJB Specification Versions ...................................................................... 75
2. APPLICATION

2.1 The Choice of Application

2.2 Application Description

3. DESIGN

3.1 Design 1

3.1.1 GateBean

3.1.1.1 Creation

3.1.1.2 Transactions

3.1.1.3 Improvements

3.1.1.4 Persistence

3.1.2 AlarmPlug2Bean

3.2 Design 2

3.2.1 ForwardBean

3.2.1.1 Creation

3.2.1.2 Transactions

3.2.1.3 Persistence

3.2.2 TraderBean

3.2.2.1 Creation

3.2.2.2 Transactions

3.2.2.3 Persistence

3.2.3 BackBean

3.2.4 Improvements

3.3 Design 3

3.3.1 ForwardBean

3.3.1.1 State of ForwardBean

3.3.1.2 Transactions

3.3.2 AlarmConsumer

3.3.3 BackBean
3.4 Design 4 ........................................................................................................ 129
3.5 Comparison of the Designs ........................................................................ 134
3.6 Design Afterthoughts ................................................................................... 137

4. IMPLEMENTATION ....................................................................................... 140

4.1 Implementation of Clients .......................................................................... 140
4.2 Implementation of Plug-in in Designs 1 & 2 ............................................. 141
4.3 Tests and Benchmarks ............................................................................... 141
4.4 The Java Code ............................................................................................ 148

5. DISCUSSIONS .............................................................................................. 150
5.1 EJB in Element Management Applications ........................................... 150
5.2 Commercial Products ................................................................................. 152
5.3 EJB vs. COM+ ............................................................................................ 153
5.4 General Discussions and Conclusions ...................................................... 154

Bibliography ....................................................................................................... 159
About this paper

This document is the formal report for a Master’s Thesis assignment that has been carried out at Ericsson Utvecklings AB in Älvsjö. The Master’s Thesis (MT) work is part of the mandatory assignments of the Computing Sciences program at Uppsala University. In April 1999, the assignment specification (provided by Ericsson) was approved by Sven-Olof Nyström, who is the MT supervisor at the Information Technology Department (sub-department of Computing Sciences) of Uppsala University. The MT work is worth 20 academic points at Uppsala University.

The supervisor at Ericsson (UAB/I/MT) has been Måns Engstedt, and the examiner at CSD is Richard Carlsson. Måns Engstedt had also an examiner’s role from Ericsson’s point of view, and he approved the actual work that had been done in January 2000. The formal approval by Måns Engstedt was given in late July 2000, when he received an earlier version of this report (that version differed from the current report mostly in details and formats).

I would like to thank Måns Engstedt for his aid and his constructive criticism.

Richard Carlsson received an outcast of this report in winter 2000. In August, a more complete version was delivered to Richard for examination together with the implementation code. The report was corrected and extended according to the remarks and suggestions of Richard, and was finally delivered in December 2000.

Hama Biglari
Introduction

Objectives
This document is about **Enterprise JavaBeans (EJB)**, that is Sun Microsystem’s architecture for server-side components. The first version of the EJB specification was introduced by Sun Microsystems in 1998. In 1999, **Ericsson Utvecklings AB (Core Network Products CNCP)** had announced a Master’s Thesis assignment in order to investigate this new component technology, and to see whether it can be used in element management applications. The assignment would be carried out at the CNCP unit **Element Management (Application Design)**, and therefore the focus should primarily be on the usability of EJB in management applications for telecommunication nodes and switches.

At the unit Element Management, the main task is to provide and develop systems, tools and applications for management of target systems that can be network nodes, elements, switches and platforms. By management we mean configuration, monitoring, operations and updates on the target systems. The tools are often used remotely, meaning that the machine on which the management tool is running is different from the target system machine. Thus the tools may communicate with the target machine either in a client/server system or via a middle-ware.

If the middle-ware approach is chosen, the following issues must be considered:

- **Portability** – can the tool be used on different processors and operating systems? If we implement the middle-ware classes in a portable language such as Java, does it mean that our objects/component are all together portable?

- **Code reuse** – do we write and develop different tools with code reuse in mind? If we have tools (or parts of tools) that solve similar problems, how can we share these parts among different tools?

- **Scalability** – what happens if the number of operators using our tools increase? What happens if the communication rate between clients and the target systems increases? Can our middle-ware respond to increased load if we add processor power?
• **Maintenance** – as with all software, the management tools must be developed and maintained continuously as the customer requirements change over the time. How can we make clear separation between the customer’s concern (business logic) and our underlying architecture and services (databases, transactions, security, fail-over, load balancing etc)?

Another issue that is related to middle-ware systems is the separation between presentation, model and data. Let’s say that the management tools present information about data that is stored somewhere in a database. In the middle-ware, we must have a model for the stored data that we have brought in (from the database) for processing, while the presentation part of the tool should only present the information (the model) in a convenient way. While a tool might use different presentation formats for different clients, there should be only one model (for the same information) in the middle-ware.

Thus we should have our models and our information processing (business logic) on the middle-ware. The customers use the client parts of our tools in order to take care of the business logic, and the client programs communicate with objects/components on the middle-ware. For the clients, the middle-ware is normally the only one to communicate with; in other words, the middle-ware is a server for the clients. This is where EJB comes in; we said that EJB was an architecture for *component-based development on the server-side*. Now the question is what parts of our tools would be located on the *server-side*. We shall answer this question later on, and we are also going to be discussing all the other issues related to middle-wares.

Let’s carry on with the CNCP objectives and see what typical tasks we may be performing in our element management tools:

• In the element management tools we want to present information about the state of the target systems. The client part of the tool must present information on the user’s demand, but the tool must also inform the user about sudden changes of the state of the target system.

• The user is usually an operator whose task is to respond to some specific events that occur on the target system. Such an event could be activation of
alarms on the target, and the user’s respond may be taking some actions, e.g. updating the state of the target (by making requests to our tools).

- The user might want to view information about the history (previous states) of the target system. We said that we usually store data in a database and that the data models reside on the middle-ware. Thus information about the history of the target system is stored in a database, and our tool’s client part presents the model information provided by the middle-ware.

- The user may be required to provide some Id and password in order to perform operations and access information

Enterprise JavaBeans is a technology that promises to provide support and help us achieve reliable, scalable, secure and portable middle-ware systems. Furthermore, EJB claims to provide services and support for transaction management, persistence and security. The purpose of this Master’s Thesis assignments, and the main issue in this report is to verify whether EJB can be used in our element management tools. We shall try to check whether it is possible to use EJB in order to build and develop reliable, portable and scalable systems for our tools and our applications. We shall also try to find and illustrate weaknesses of EJB, both in general and in particular regarding similar telecom management applications.

Since EJB was a relatively new technology, and especially since it was a new technology at CNCP, the assignment specification of this Master’s Thesis was written so that the employee working with the assignment would be able to choose some fields of interest under supervision of the supervisor Måns Engstedt. The specification also says that some tasks may be carried out if there is time: for instance, comparing EJB with Microsoft’s COM+, or comparing different commercial EJB products. Other fields and tools mentioned in the specification are UML and Rational Rose, Design Patterns and JavaBeans.

The assignment was started by a study phase where Java, RMI, JDBC, JNDI and CORBA, were the new topics to be learnt, and the studies in EJB were started in parallel. During the autumn 1999, Hama Biglari (the employee to carry out the
assignment) and Måns Engstedt (the supervisor) decided to concentrate on transactional issues and different component relations in the design alternatives. Therefore, the following issues have been omitted or touched very shortly:

- UML and Rational Rose
- Design Patterns
- EJB security
- Implementation of corresponding systems using COM+

Another issue that was not done was: test and verification of EJB portability by porting the components from Weblogic to other EJB products. Actually, some work was done to port the implemented EJB components to Inprise Application Server towards the end of the work in January 2000, but it soon turned out the large parts of the code must be modified for two reasons:

1) Inprise Application Server required that the EJB components’ code would be written according to EJB Specification version 1.1, whereas the implemented EJB components on Weblogic were written with version 1.0.

2) EJB components built on Weblogic have product-specific code, and the code has to be modified before it can be ported to other EJB servers.

**Preface**

Now that we have presented the objectives of the assignment, let’s see how this paper is structured. The document consists of three different parts: theory, application and discussions. The theory is covered by section 1. The second part, application, is itself consisted of three parts: application description (section 2), designs (section 3) and implementation (section 4). In the discussion part (section 5) we discuss problems and solutions that have been addressed in the previous sections, and we will also try to illustrate some strengths and weaknesses of EJB.

**In section 1,** we are going to start with an introduction to the world that Enterprise JavaBeans (EJB) is born into. We describe the problems and the needs that make EJB interesting, starting with the historical background of current middle-ware systems. We will discuss some weaknesses of old-fashioned client/server systems, and then we
will introduce some concepts and techniques in distributed computing. After that, we
will describe the EJB architecture by presenting and describing the basic and standard
components of an EJB-based system, as well as classes, interfaces, methods and some
new concepts that build up EJB. We are going to give examples and discuss correct
and faulty usage of some EJB features.

Section 2 presents and describes the application, and we can reveal here that it is an
alarm-handling application. First we discuss why we have chosen that application, and
then we give a detailed description of the application itself. Note that the application
will be in focus when we try to investigate weaknesses and benefits of the different
designs and scenarios in section 3.

In section 3 we present our four different designs. We start with some basic ideas
about design issues such as scalability and throughput, and then we present our first
design. Here it is important to inform the reader about the state of the first design
(design 1, section 3.1): in this design we break against some important rules of EJB.
We still have presented these ideas as well as the implementation, and the reason is to
share our thoughts with the reader, and to show how and why things work (or not
work). In this faulty design we also present some problems that are referred to in the
other design comments, and we describe some scenarios in detail. So it is up to the
reader to read or to skip the first design. In designs 3 and 4, we discuss several
alternative solutions and their usage (these alternatives differ mainly in transactional
behaviour). When all designs have been described and analysed, we present a
comparison of the four designs.

Section 4 is about the implementation of the designs described in section 3. We also
discuss benchmarks and test results, and we mention some problems that have
hindered us in creating ideal or desired testing conditions.

Section 5 is where we start with our discussions about the work that we have done,
design, implementation, EJB advantages and limitations as well as weaknesses. Here,
we try to address some questions and known problems regarding the usage of EJB in
element management applications. Then comes a short presentation of some
commercial EJB products, where we also provide web addresses where the current
products are listed. We also present some ideas for comparison of EJB with
Microsoft’s corresponding technique, namely COM+. The final part of the section contains a general discussion about EJB.

Before the theory part starts, we have written some guidelines in order to prepare the reader for the pedagogic purposes of colourful pictures as well as some notational conventions throughout chapters 1 and 3.

All implementation and all testing has been done on Microsoft’s **Windows NT**, and all Java code has been implemented with JDK 1.1.7B.
**Readers’ Guide**

In the following text, sometimes it is difficult to describe a method, an interface or a mechanism without mentioning other parts that have not been introduced and described yet. And sometimes the same information may be presented in more than one section.

By using colourful pictures I have tried to make it easier for the reader to understand the relationship between the different parts involved in the illustrations. I have also tried to use each colour for a special functionality and I have given each kind of component its own colour. Thereby the reader will hopefully be able to see both differences and similarities between the different examples (and designs) that are described below.

In *section 1* (1.2.1 & 1.2.4) different colours are used for the different classes and interfaces used in the Enterprise JavaBeans architecture, and some names and methods related to those classes and interfaces are also written in the same colours. The frequently used colours are:

- **GREEN** - naming services
- **DARK BLUE** - home interface
- **BROWN** - remote interface
- **PINK** - business logic methods

In *section 1* (1.2.4 & 1.2.5) we introduce two new colours that are used to indicate what kind of bean (EJB component) is used in the picture:

![Session Bean](image1) ![Entity Bean](image2)

In *section 2* where we present the application we introduce two objects that will have the same two colours throughout this document, but the colours shown above lose their meanings in section 2:

![AlarmConsumer](image3) ![AlarmHandler](image4)
In section 3 we present the different designs used in this assignment. We still use the colour orange for Entity beans (alarms), but we will also be using another Entity bean in all the designs, and that Entity bean will be painted light blue:

In section 3 we will also be using several different arrows for business logic method calls:

<table>
<thead>
<tr>
<th>Arrow</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Subscribe on target(s)" /></td>
<td>Subscribe on target(s)</td>
</tr>
<tr>
<td><img src="image" alt="Transaction call to update an alarm" /></td>
<td>Transaction call to update an alarm</td>
</tr>
<tr>
<td><img src="image" alt="Return value of transaction (changed alarm)" /></td>
<td>Return value of transaction (changed alarm)</td>
</tr>
<tr>
<td><img src="image" alt="New spontaneous alarm sent from the target" /></td>
<td>New spontaneous alarm sent from the target</td>
</tr>
<tr>
<td><img src="image" alt="In some designs all alarms are sent asynchronously (by a call – not by returning a value) from target and also from plug-in" /></td>
<td>In some designs all alarms are sent asynchronously (by a call – not by returning a value) from target and also from plug-in</td>
</tr>
</tbody>
</table>

Please note that the meanings of the arrow colours differ between sections 1, 2 and 3. All sections have one thing in common concerning arrows: solid arrows are normally used to illustrate calls (method invocations), while a returned values’ path is illustrated by an arrow similar to the pink arrow (changed alarm) shown above. Arrows that look like being hand-drawn show references, and relationships such as inheritance are explicitly written out.

If you print this document on paper, please use a colour printer – it makes the designs and illustrations much more comprehensive. References are written with marks such as [E4] or [C1], in which the letters denote the specific category in the bibliography list.
1. Enterprise JavaBeans (EJB)

In March 1998 Sun Microsystems introduced an architecture for distributed server-side components. It was called Enterprise JavaBeans (EJB), and it would provide the architecture for secure, transactional, scalable and portable middle-ware application servers. There were also plans for persistence support, and persistent components were included as optional additions in the first version, i.e. EJB Specification 1.0. We will take a quick look at EJB versions 1.1 and 2.0 at the end of this section.

The name Enterprise JavaBeans may remind you of the concept of Java Beans, but the similarity between these two concepts is only in their names and the fact that Java Beans are also components. And it is important to realise that EJB is not ‘a better Java Bean’ or an extension of it. Neither is EJB a product – it is an architecture. EJB is said to be the fruit of two decades of development in the object-oriented world. During the past twenty years there has been a rapid development in several areas:

- The client-side development: graphical interfaces, Java applets, caching, multithreading and transactional sessions, security checks etc. [E4]
- The server-side development: n-tier servers consisting of database servers, web servers, and the distinction between stored data (in the database tier) and data model (in the server’s business logic) [E1,E3]
- Remote Procedure Call (RPC) and Remote Method Invocation (RMI) [J2]
- Distributed programming with concurrent calls and multithreading
- Distributed transactions over different databases and different vendors’ database products [E3]
- Increasing knowledge about the advantages of object-oriented programming (OOP) such as code reuse
- Object Management Group (OMG) standards, and the development of Common Object Request Broker Architecture (CORBA) and its component model [C3]
We will talk about each of the above mentioned issues:

1. The client-side development had some major problems: the clients had got too heavy. They had to be involved in mechanisms other than business logic, such as thread synchronisation, security, transaction management, database connections etc. This was a contradiction to the idea of code-reuse in OOP, as the clients had to include code written for each of these mechanisms. And programmers had to write code for transactions, security and thread synchronisation all over again for each new application. In the best case, a company could have its own ‘services’ source code to be used in different applications. Clients knew too much about a lot of things; business logic was just a small part of the client [E4]. This was the typical client-server system where the server was often just a database server, as shown in Figure 1.1:

![Figure 1.1 Typical client-server systems](image)

2. The servers were successively developed into 2 tiers, and later even more. First we had 2-tier systems consisting of a database-server and a client as shown in the picture above. Then the server developed into 1.5 tiers consisting of stored data and its model (see [E1,E3]), as illustrated in Figure 1.2:

![Figure 1.2 Client-Server system with a 1.5-tier server](image)
Eventually the 1.5-tier servers were divided into two well-separated tiers, and that is how n-tier systems arose (see Figure 1.3):

![Figure 1.3](image)

Figure 1.3 The N-tier architecture

3. Remote Procedure Call (RPC) is the mechanism that enables a program or an object executing on a machine to communicate with another machine’s object/program and call its methods almost as if the accessed object would be on the local machine and in the same runtime environment. In the Java world this mechanism is provided by Remote Method Invocation (RMI). RMI almost provides network transparency. [J2]

If a Java class is declared to implement the interface `java.rmi.Remote`, and if we compile our code with RMI compiler so that we get the proxy classes (stub and skeleton, see [J2]), then we can let a Java object call an instance of this remote class. Assume that the caller is on machine A, and the remote object on machine B. Then such an invocation (the call to the remote object’s method) will be executed on machine B.

But if the class is not declared as remote (its instance is still on machine B), then the caller (object on machine A) will get a reference to a copy of the called object. That is, a copy of the object will be transferred from B to A, and the invocation will be executed in the runtime environment on machine A.

In the first case the arguments needed by the B object’s method must be transferred, whereas a copy of object B itself has to be transferred in the second case. An object must be **serializable** in order to let us transfer it. In other words, classes whose instances will be transferred in RMI (whether they are called objects or arguments of a method) must declare the interface `java.io.Serializable`. 
Primitive types are serializable in Java. The serialisation mechanism is not used only in RMI – it can also be used in order to move an object to storage (file or database), which is why the interface is in the package java.io.

We shall illustrate the RMI usage of remote and non-remote objects with an example in Figure 1.4, where we want to know the registration number and colour of our car and then paint the car green if its current colour is red.

```
Class CarPainter1 {
    // RMI registry code ...
    // ...
    CarA car = get(regNumber);
    String color = car.getColor();
    if (isRed(color))
        car.paint(“green”);
}
```

```
Class CarPainter2 {
    // RMI registry code ...
    // ...
    CarB car = get(regNumber);
    String color = car.getColor();
    if (isRed(color))
        car.paint(“green”);
}
```

**Figure 1.4** RMI invocations on remote and non-remote objects (cars)
In Figure 1.4 we are in Stockholm and our car is on the other side of the Moon. If our car’s class is declared as remote we just send a request to the moon and the value denoting the colour will be returned to us. That’s a light-weighted call (we send few or zero arguments) and we get a string transferred back to us. But if the car would not be remote, then the car object (the fact that it is a copy does not make a difference here) will be sent down to earth so that we can check its colour. And that is for sure not a light-weighted call. On the other hand, if the process of painting is cumbersome (for instance, if it is expensive to send all colour to the Moon and that the transport takes a long time), then it might be a good idea to bring the car down to earth, paint it green and then send it back to the Moon.

4. Distributed programming had been developed on the client-side in a client/server system, or among several co-operating nodes in a network. Often, code for management of multiple threads (concurrent calls) had to be written in the methods that executed the business logic of an object. Especially in the graphical interface provided by the client, distributed programming had been developed. Also the server’s code for management of several client calls had to be written from scratch.

5. Java has a service called **Java Transaction Service (JTS)**, that is defined by SUN based on CORBA’s **Object Transaction Service (OTS)**. JTS supports distributed transactions, which is the term used when multiple nodes in a network are involved in one unit of transaction (in other words: a distributed transaction spans over more than one network node). By using **Java Database Connectivity Interface (JDBC)**, a Java application can easily use drivers and bridges in order to connect to one or several database servers, and run SQL commands. [E3]

In JDBC 1.x there was no support for 2-phase commit, and therefore, distributed transactions could not be performed properly. By properly performed transactions we mean that the four defined rules for transactions not be broken. These four rules are referred to by the acronym **ACID (Atomicity, Consistency, Isolation and Durability)**. If you are not familiar with these rules and the concept of transaction, then it would be a good idea to get some basic knowledge in the topic so that you will be better prepared when we discuss transaction management in EJB.

Anyway, 2-phase commit means that database updates are done in two steps: first the update is written into a temporary storage (a log). If the caller confirms that the
temporary update should be made in the real database, then the update is written out to the real database. When several network nodes are involved in a transaction, the second call to confirm the update is made only if the temporary updates have succeeded for all nodes. In JDBC 1.x, you could have one transaction that made updates in several network nodes, but there was no logging, which means that even if the update to the second node failed, the first node’s update would still remain unchanged. JDBC 2.0 has 2-phase commit, and this means that distributed transactions could and will be supported in products and APIs from now on, including EJB. An EJB product’s support for distributed transactions will be dependent on whether the database product used supports JDBC 2.0 in its low-level API. [E9]

6. CORBA has been increasingly used in the past few years. In this OMG architecture there is something called Object Request Broker (ORB) that works as a translator between different applications or objects written in (probably) different languages. The development in distributed programming, networks and data communication brought up the wish and the need for several different components to be integrated in one and the same system, and that applications or components (read: objects) written in different languages would be able to communicate with each other. An object written in C++ should be able to call a Java object (without knowing what language the caller is implemented in), and get or send data. The caller needs only to know a public interface implemented by the called object. This public interface should be written in a language that can be mapped to C++, Java, Cobol, SmallTalk or any other language supported by OMG’s ORB mapping. [C1,C3]

The called Java object’s public interface (don’t confuse this interface with Java interfaces) must be written in the language called Interface Definition Language (IDL). Then we compile the IDL file by using a compiler that generates the Java mappings of all the declared IDL interfaces and structures. The IDL-to-Java compiler generates Java classes and Java interfaces that correspond to those declared in the IDL file (see Figure 1.5), and additional helper and holder classes are also generated for runtime delegation of calls and conversion of values. For each IDL interface several other classes are generated in addition to the Java interface and the holder/helper classes.
The caller can either find the Java/CORBA object via the naming service, or by getting a special string that can be converted to a CORBA Object reference and then downcast (narrowed) to the type of the Java interface corresponding to the IDL interface. The naming service of CORBA is part of the Common Object Services (COS), and is referred to as COS Naming Service.

**Figure 1.5** Compiling IDL files into Java/C++

**Figure 1.6** Obtaining reference to a CORBA object by using an IOR string
The second alternative involving a string uses the reference mechanism called **Inter Orb Reference (IOR)**. Each CORBA object has a unique address consisted upon, among others, the IP address and a port number of the machine it is living on. This information is part of the unique address that also shows what ORB and what Object Adapter is used for creation of the object. All this can be acquired as a string by the creator of the CORBA object. The string can later be fetched from a file or a name server, and that’s what the caller of our object can do. The string is converted to an object reference by the ORB, as seen in **Figure 1.6. [C1]**

Although the caller of a CORBA object uses a reference and calls it just as if it were a local Java object, all communication is done through the orb. When the caller calls the CORBA object’s method `foo` with the arguments `1` and “hello”, this call is actually made on the stub object (the caller’s reference points out the stub object initiated and provided by the orb). The stub then forwards the call to the orb. The protocol used here (3 and 4 in the above picture) is OMG’s Internet Inter-Orb Protocol (IIOP), that can transfer all IDL types, and the Java values `1` and “hello” are converted to IDL values in order to be transferred on IIOP. Before the call is made to the CORBA object itself, the IDL values are converted to values and types of the called object’s implementation language. [C1]

So now that we have talked about the needs and progresses of the past two decades, it is about time to see where EJB comes in, what problems it can solve, and where it could fit in the expanding world of computing.
1.1 Why EJB

Server Side Architecture  [E4, E5, E6]

We saw that the clients remained heavy although the server side of the client/server architecture was developed and divided into several tiers. Shouldn’t we do something about our clients instead of developing the servers all the more? The answer is yes; we do want to make our clients thinner. We must also point out that there has been a lot of development on the client side, such as caching session data and providing advanced and user-friendly graphical interfaces. But then the question is: how do we make a client’s life easier by developing the server side?

Let’s go back and see why clients remained heavy: clients contained a lot of code for management of underlying system resources, and they had to deal with a whole bunch of problems that were not part of the business logic of the application. So if we let the server take care of these issues (transaction management, data storage, multiple threads etc), then the only thing that would remain on the client side is the calls on business logic methods of server components and some initialisation code for connecting to the server. From the client’s point of view, all that exists is the business logic interface provided by the server. Those interfaces are part of the server side components.

Before we look at what a component is, let’s see what happens on the server side now that we have shovelled all these client problems to the server. On the server side, the idea is to have objects that are instances of classes that implement the business logic interfaces that are visible to the client. These classes will not contain the code that we removed from the clients. In other words, the classes normally avoid dealing with issues such as thread synchronisation, RMI programming, database connections and database updates, transaction management and security (authorisation and authentication). The reason is to give application developers and server-side programmers the possibility of concentrating on the business logic, and we also want to make clear the separation between the business logic implementation and the underlying mechanism and system resources of the server. But an instance of our business logic class must be given a lot of support for all the mentioned tasks and services. And this is exactly what the server must do: to provide services to the objects that live inside the server so that they can carry on executing business logic methods properly. The server must also provide means for clients’ connection, database connections and so on. These servers are called Application Servers as they
provide the same set of services to different objects that are used in different applications. The Application Server is the middle-ware part of modern N-tier architectures, as shown in Figure 1.7:

![Diagram of N-tier architecture with thin clients](image)

**Figure 1.7** The Application Server is the middle-ware in an N-tier architecture with thin clients

A **component** is an object plus much more. When we speak of component-based architectures and component-based development on the server side, then we have taken the step from the object-oriented phase into the new era in the server side evolution. Up until now, we have been trained in OOP; we are supposed to think about code-reuse, maintenance, abstraction, separation of concerns, encapsulation etc, and we want to keep on thinking in those terms, but now we want to build further on the knowledge and experience that we have gained in OOP. Objects have attributes and methods, state and behaviour. Components have states (not everybody agree on this though – we will discuss that later) and public business logic (BL) interfaces, but components also have transaction modes and transaction attributes, security roles and persistence properties (again, this view is not shared by all companies), as shown in Figure 1.8. Components are supposed to be coarse grained, so that we will be able to customise them for different applications. [E4]

![Diagram of objects vs. components](image)

**Figure 1.8** Objects vs. components
Having a component-based server should give us the possibility of having a scalable, secure and distributed system with persistence properties. EJB does give us such a server side, as it provides: [E1, E3, E5]

- Declarative transaction management, and distributed transactions
- Persistence support (persistent components)
- Declarative security for authorisation and authentication
- Management of multiple component (object) instances
- Management of multiple threads and concurrent calls
- State preserving components during client sessions (stateful components)
- Portability
- Mechanisms for increased scalability

In an EJB based server there are three main parts: the **EJB server** (the application server that provides EJB), the **EJB container** and the **EJB components** (referred to as EJB beans or just beans), as illustrated in **Figure 1.9**. The beans live inside a container that provides services and support to the beans by communicating with them during their lifetime. Earlier we said that the BL implementation class (that is really the bean class) would contain only BL methods, but there are actually several other standard methods that must also be written in the bean class. These standard methods will be called by the container, but they are small and we don’t write code transaction management or anything like that in these methods. [E1]

![EJB Diagram](image)

**Figure 1.9** The three main parts of an EJB-based server

Thus the container provides services to the beans, and it also ensures that state synchronisation, security and persistence requirements are fulfilled, and the server provides services to the container. We will discuss all these issues in section 1.2, but it is probably a good idea to try to get a picture of the whole architecture before each part is described in isolation. [E1]
The API between the EJB container and EJB server is not well specified, which gives vendors the possibility of letting the server be a container as well. Anyway, there is logical distinction between the server and the container. The container is supposed to interact between clients and the beans, while the server is supposed to do a lot of work behind the scenes and provide system resources, communication channels and database connections as well as gateways to other systems. [E5, E1]

The clients will not call the beans directly; instead they communicate with the beans’ proxy objects. The container’s interaction occurs between the proxy objects and the bean instances, and it gives the container the opportunity of providing its services to the beans. In this interaction, the container performs transaction management, security checks, state synchronisation, database updates, thread synchronisation and other tasks for increased concurrency and scalability.

![Diagram](image)

**Figure 1.10** Clients access EJB beans via proxy objects provided by the Container. The EJB Server provides services to the Container, such as database connections and handling of communication channels for clients.
1.2 EJB Architecture

An EJB system is an n-tier system that consists of one or several clients, the EJB server and one or several databases (see Figure 1.11). This is only the logical view of an EJB system as the mentioned three tiers, or two of them, could be on the same machine [E1]. We could have the EJB server and the database on the same machine. Even the client and the server could reside on the same machine, as is the case when a bean calls another bean in the same container and thereby acts as a client. And a combination of these two cases makes the three tiers reside on the same machine, as it is in our built prototype.

Furthermore, the database tier may itself consist of more than one layer, for example a database server, JDBC drivers and the actual physical storage layer. Also the EJB server may use services provided by other layers in the middle-ware, such as HTTP server, servlets, JNDI services, services for integration with CORBA or with Microsoft’s Component Object Model (COM).

The client calls remote objects, i.e. EJB beans, on the EJB server. So the client needs to know how to find the EJB server and how to use its beans. The beans live inside an EJB Container residing on the EJB server. When a client and a bean communicate, the server is never accessed directly by them. Instead all the calls are done via the container. The third tier, i.e. the database, can be explicitly accessed by the beans, e.g. using JDBC. Alternatively, the container can do data storage for the beans. [E5, E8]

1.2.1 EJB Client  [E1, E2]

The client never calls the EJB beans directly. Instead it communicates with two other objects created and provided by the container. Each EJB bean has a home interface and a remote interface, as was shown in Figure 1.10. A client talks to an EJB bean through these two interfaces; the objects created by the container are implementations of these interfaces. [E1]
The client has the following tasks: [E1]

a) Finding the bean
b) Getting access to a bean
c) Calling the bean's methods
d) Getting rid of the bean

a) Finding the Bean (see Figure 1.12 and State 1 in Figure 1.14): There are two ways specified for a client to find a bean. The first way is by using the Java Naming and Directory Interface (JNDI), which is a Java API that provides a uniform interface to naming and directory services. A Java client can use JNDI to lookup an EJB component. The JNDI call returns a reference to an object implementing the home interface of the EJB. This reference can be used to find or create an instance of the EJB bean. The other way to find an EJB component is to use the naming service in CORBA's Common Object Services (COS). COS naming can be used directly or via JNDI.

![Diagram](Figure 1.12)

**Figure 1.12** Finding a bean is started by obtaining a reference to the object that implements the bean's remote interface. This is done by a lookup at the naming service provided by the server.

b) Getting Access to a Bean (see Figure 1.13 and State 2 in Figure 1.14): The home interface is used to find an existing instance of an EJB bean, or to create a new instance. In either case, the client gets a reference to an object implementing the remote interface of the EJB bean. Then the client uses this (remote) interface to communicate with the EJB bean residing in the EJB container.
c) Calling the Bean's Methods (see State 3 in Figure 1.14): The methods declared in the remote interface are, and must be, implemented in the EJB bean. These methods will be invoked on the created/found instance of the bean when the client calls those methods using the remote reference. As we said before, the container has created a proxy object implementing the remote interface, and the client has a reference to this proxy object. The proxy object forwards the client's calls to the bean instance.

A4:  try {
    myBeanRemoteRef.foo();
} catch( RemoteException re ) {}
Clients communicate with EJB components using either CORBA or RMI, as is shown in transition b in Figure 1.14. We will now take a look at these two client types.

1.2.1.1 RMI Clients

When a client using RMI communicates with the proxy object of an EJB bean, the calls actually go through the stub and skeleton objects of the bean’s proxy object. These classes are automatically generated by the container, which means that we don't do any server-side RMI programming. On the other hand, if the client is declared as remote, then its stub and skeleton classes must be generated. RMI clients use JNDI to lookup a bean's home interface. [E1]

Suppose the remote interface X declares a method named foo() that returns an integer. The sequence diagram shows the communication steps when a client calls foo() on an X reference. EJBObjext is the proxy object of the bean, and it implements X. The division between the stub and skeleton classes is also the division between the client platform and the remote EJB platform (see Figure 1.15).
Now suppose that the client has given its own reference to the bean, so that the bean will call the client’s method \texttt{bar()}, i.e. the bean makes a call-back, as illustrated in Figure 1.16:

1.2.1.2 CORBA Clients [E1, E7]

In EJB Specification 1.0 there is an EJB-to-CORBA mapping specified so that CORBA clients will be able to use EJB beans. A CORBA uses COS naming in order to lookup EJB home interfaces, and therefore there is a mapping between JNDI names and COS names. Furthermore, a CORBA client may either be a “pure CORBA” object that uses IIOP to communicate with the beans (see Figure 1.17), or it may use RMI-over-IIOP so that the client API code will be the same as for an RMI client (see Figure 1.18). A CORBA client that manages its own transaction (calls a bean within an open transaction) uses CORBA Object Transaction Service (OTS).
A CORBA compliant EJB implementation must provide IDL interfaces for each bean, and it should preferably provide tools for generating IDL. The environment of the implementation should support COS naming and provide a JNDI-to-COS mapping for names. The environment should also support OTS transactions.

The relationship between an Enterprise JavaBeans component and a CORBA object, and the mapping of the Java RMI remote interfaces to OMG IDL is defined under Mapping of Distribution in EJB Specification 1.0. The process by which COS can be used to lookup the home interface of a bean, i.e. EJBHome objects, is specified under Mapping of Naming. In Mapping of Transactions, the propagation of client-demarcated transactions (the client could also be a bean) is ensured by making the IDL interface (corresponding to the remote interface of a bean) inherit from the CosTransactions::TransactionalObject IDL interface. This enables the ORB and OTS to propagate the client’s transaction context to the bean. In Mapping of Security, the server ORB is required to determine the client’s identity that is specified in the Access Control List (ACL). This mapping is dependent on the actual security/communication protocol used by the ORB: plain IIOP, Common Secure IIOP, and IIOP over SSL. [E8, See Chapter 4.4 in EJB Spec 1.0]

When we define and write our EJB components, we should think one step ahead. Even though EJB is totally Java based, having a CORBA compliant implementation means that we may have C++ clients that know only our beans’ IDL interfaces, and
use IIOP to call our beans. This implies that the business logic methods of our beans must have signatures that fit into the IDL language. That is, arguments and return types of methods must be primitive types, arrays or simple objects that have only data fields (no methods other than getters/setters). In other words, the types used in the visible interfaces must be limited to the types that can be converted to IDL types in order to be transferred over IIOP. [C1]

1.2.1.3 RMI over IIOP  [E9]
The infrastructures of RMI and CORBA were two totally different things. For example RMI had its own messaging protocol (JRMP), a registry mechanism of its own and a set of services similar to existing CORBA services, but non-compatible with the latter were under development. There was a demand for one infrastructure that would provide Java programmers the possibility of using both the CORBA and RMI programming models. This brought up the idea of “RMI over IIOP”.

Figure 1.18  CORBA client using RMI-over-IIOP and RMI-IDL interfaces to communicate with EJB components

Using RMI implies some limitations on method arguments and return values sent between the stub and the skeleton (this communication occurs normally across a network). The types (remember that we’re talking Java) that can be used are:
If an object is both remote and serialisable, then it is not a valid type for RMI communication, and an attempt to pass such an object as argument or returning the object (that process is called marshalling) will make `java.rmi.RemoteException` be thrown. The same thing happens if we try to marshal a remote object that contains a method that does not throw `java.rmi.RemoteException` or a subclass of it.

IIOP 1.2 is the version that is included in CORBA 2.3 supports the semantics required by the messaging protocol used by RMI. Earlier versions of IIOP could not be used for RMI communication, since the requirements of JRMP could not be fulfilled. The difficulty lied in Java value types passed in JRMP: at that time IIOP could only pass primitive types (int, long, boolean etc) and also arrays and structs. These types could be passed as method arguments or return types of the IDL interfaces, and the easy thing about arrays and structs was that they all are state-only types, which means that they didn’t have any behaviour; they didn’t have any methods. After all, it does not make sense to send a Java object to a C++ client and let the methods of the Java client execute in the C++ environment. So we want to be able to send Java value types such as objects that have methods, over IIOP. In other words, we want to have RMI semantics over the IIOP protocol. \[E9, \text{See also } \text{J3 (JDK 1.3)}\]

On way to do this would have been to provide a mapping consisting upon the marshalling rules, from RMI to the new IIOP. OMG did not do this, and instead they provided a mapping from RMI to IDL. The advantage of OMG’s solution is that RMI interfaces can be converted into equivalent IDL interfaces. The important consequence of this is that in Java applications using RMI, we can add new C++ clients (or client written in other languages, as long as they are covered by IDL) and let the new clients use the corresponding IDL interfaces of the former Java-RMI interfaces, without changing the interfaces. Of course this enables interoperability between RMI and CORBA, but the most important benefit of this mapping is really gained by EJB: you let your beans declare home and remote interfaces in IDL with RMI semantics (read: RMI interfaces), and you can choose to have non-Java clients that use those IDL interfaces. EJB is part of the J2EE (Java 2 Enterprise Edition), and
RMI-over-IIOP is said to be the thing that really makes it possible for EJB to fit into J2EE, with the emphasis on the word Enterprise. [E9]

1.2.2  EJB Container
An EJB container is an environment in which EJB beans execute. It serves as a wall between the beans and the outside world. It creates proxy objects that forward clients' calls to the bean instances. This makes the container much more flexible in servicing clients' requests: if possible (as is the case with stateless Session beans, and sometimes also with Entity beans), the container creates a pool of beans in order to reuse the same instance for different clients. The container provides several services to the beans so that the beans won't have to worry about the underlying mechanisms of those services' implementations. These services support the beans in transactions, management of multiple instances, persistence and security. [E1,E2,E4,E5,E8]

1.2.2.1 Support for Transactions
The EJB container's task is to provide beans with the ACID (atomicity, consistency, isolation and durability) properties of transactions. In addition to the common approach using "begin", "commit" and "rollback" instructions of transaction management, EJB also provides declarative transaction management. In the declarative approach, the type of transaction support is specified by the bean's author, so that at deployment time the container will read that information and support the bean with the specified type of support.

An Entity bean can not manage its own transactions, and is therefore not allowed to use the attribute TX_BEAN_MANAGED. Session beans can choose between bean managed (BMT) and container managed transactions (CMT), but they can not have both, i.e. a Session bean has either BMT or CMT. Six modes of transaction management are specified in EJB: [E1,E2,E4,E5,E8]

- TX_NOT_SUPPORTED
- TX_BEAN_MANAGED
- TX_REQUIRED
- TX_SUPPORTS
- TX_REQUIRES_NEW
- TX_MANDATORY
Below, the left picture in each mode explanation shows the case when the client has an open transaction and the right picture shows a client with no transaction in effect. The blue/green squares denote transaction contexts.

**TX_NOT_SUPPORTED** means that the bean does not support any transactions and cannot be used from within a transaction. When a client calls a bean method from within a transaction, the container suspends the client's transaction and invokes the method. When the method's execution is done the client's transaction is resumed. This mode should not be used on Entity beans.

![TX_NOT_SUPPORTED Diagram](image)

**TX_REQUIRED** means that the bean's methods must always be executed in an open transaction. So if the client calls a bean method within a transaction, the bean runs in that transaction context. Otherwise, the container starts a new transaction before invoking the bean's method, and terminates the transaction before returning to the client.

![TX_REQUIRED Diagram](image)

This transaction mode could be useful if the bean’s method is going to be called both within transactions and with no open transaction, as is illustrated in the following example. We have two accounts named A and B in a bank. The accounts are beans of type AccountBean, on which we can perform ‘deposit’ and ‘withdraw’. In order to provide the atomic operation ‘transfer’ to the external clients (those used by customers), we have also a bean of type ClientBean that is called by the clients, and it calls the account beans in turn. Now Bob uses a client that has access to an instance of ClientBean, and he wants to transfer money from A to B, as shown in **Figure 1.19:**
ClientBean starts a new transaction when the client calls its method `transfer`. That is, the execution of `transfer` will be done in the new transaction of ClientBean. Having the mode `TX_REQUIRED` on AccountBean enables the beans A and B to play in the same transaction as ClientBean. This way, we have ensured the atomicity of the operation `transfer`, which consists of `withdraw + deposit`. This bank also wants to be able to perform single operations such as `withdraw` or `deposit` without interfering with the ClientBean. The reason for this might be security (that customer operations should not take the same path as operations made by internal bank clients), or the quality of service (that even if external clients can not operate on the accounts due to system failure, internal business will be able to proceed). Anyway, since these single operations also are transactional, the conditions for the isolation and concurrency properties with regards to other operations must be fulfilled. Therefore, the bank’s internal BonusClient may access the accounts and perform transactional operations, and the point is that this client does not have to deal with any transactions.

**TX_SUPPORTED** means that if the client has an open transaction when it calls the bean, then the bean's method will run in that transaction's context. If the client has no transaction in effect, the bean method will be executed without a transaction context.

---

**Figure 1.19** An example to show the usage of transaction mode `TX_REQUIRED`
Let’s say that AccountBean has also some non-transactional methods (and forget about the conditions in the previous example) that provide information about the account’s owner, his/her address, the creation date of the account etc (see Figure 1.20). These operations are not coupled with the current amount of money in the account and the bank would save a lot of system resources if they could call these methods without having open transactions – and it can. An internal client in the bank can call these methods without running inside a transaction context.

**Figure 1.20** An example to show the usage of transaction mode **TX_SUPPORTS**

**TX_REQUIRES_NEW** means that the container always starts a new transaction before invoking the bean's method. If the client calls the bean within a transaction, the container suspends the client's transaction and resumes it only when the bean's transaction is finished. Since EJB uses only flat transactions, the bean's new transaction will not be dependent upon the client's suspended transaction. And even if the client's suspended transaction later is rolled back after the bean’s new transaction has committed, then the bean’s transaction remains committed.

The usage of this mode is best shown in the *ClientBean* of the bank example. Assume that Bob’s client does not deal with transactions (just as before), while Janet’s client manages its own transactions (see Figure 1.21).
The transaction started for BeanClient due to BobClient’s calls will be committed or rolled-back before the method called returns from BeanClient. If we don’t use this mode on ClientBean, then the following situation might arise: JanetClient has opened a transaction and calls ClientBean within that transaction in order to transfer money. The transfer operation might or might not have succeeded, but the commit or rollback operations will be executed only when JanetClient explicitly calls commit/rollback. The problem is that JanetClient locks ClientBean (and thereby the affected account beans) until it calls for commit/rollback, and this lock time could become very long since JanetBean might do lots of other things in the same transaction after the transfer operation.

Thus ClientBean must perform the transfer operation in a new transaction context and while ClientBean’s new transaction is not committed/rolled-back, JanetClient’s transaction will be suspended. A consequence of this choice of mode is that any updates in the accounts affected by the transfer operation will persist after the transaction has been committed (which occurs before the method returns to JanetClient) – even if JanetClient’s transaction is rolled-back later on.

TX_MANDATORY means that the client must call the bean within a transaction, and the bean’s method will run in that transaction’s context. If the client does not have an open transaction, the container throws a TransactionRequiredException to the client.
**TX_MANDATORY** means that the bean explicitly uses transaction management commands to directly manipulate its own transactions. This is the only mode in which a bean is permitted to use Java's `UserTransaction` interface. If the client calls the bean within a transaction, the container suspends that transaction until the bean's method is done. The container never starts a new transaction for the bean, and if the bean had an open transaction before the client's call, then the invoked method will run in that context. If the bean had no transaction in effect before the client's call, then its method runs without a transaction context. This mode can not be used on Entity beans.

**TX_BEAN_MANAGED**

At time $T$ the client calls the bean within a transaction, but the bean has an open transaction of its own. The client's transaction is suspended until `foo()` is done.

---

Again, the bean has an open transaction when the client calls `foo()` at time $T$. The method `foo()` runs in the previously opened transaction.
**TX_BEAN_MANAGED**

At call-time T the bean has no open transaction, so foo() runs without a transaction context. The client's transaction is suspended until foo() is done.

---

**TX_BEAN_MANAGED**

The client has no open transaction at call-time T. The bean has no open transaction either, so foo() runs without any transaction context.

---

Transactions in TX_BEAN_MANAGED mode differ between stateful and stateless Session beans:

- **Stateful Session beans** can maintain an open transaction across method invocations, so that the bean can open a transaction in one method, let several methods be invoked by clients and run in the same transaction context, and finally close the transaction in another method. What happens is that the container suspends the bean's transaction context after each completed method invocation in order to resume it when the next method is invoked.

- If a **stateless Session bean** uses transactions in this mode, the transaction must either commit or rollback before the method returns. Thus, stateless Session beans can't maintain open transactions across method invocations, and the reason for this is that the same instance of the bean can be used by different clients.
In addition to the six transaction attributes mentioned above, EJB has also four different levels of concurrency/isolation for concurrent transactions. **Transaction Isolation Levels** have an inverse relationship with the level of concurrency for different transactions on the same bean, and they specify how much of other transactions your application will see. When two threads access the same data in a database simultaneously, three kinds of problems can occur:

1. **Dirty read**: Transaction A modifies a row; Transaction B reads it; Transaction A does a rollback and the modification is made undone.

2. **Non-repeatable read**: Transaction A reads a row; Transaction B modifies the row; Transaction A reads the row again and sees a different value.

3. **Phantom read**: Transaction A selects a set of rows that satisfy a given search condition. Transaction B inserts rows that satisfy the same condition. If Transaction A executes the same query again it will receive additional data.

There are four levels of transaction isolation:

- **TRANSACTION_READ_UNCOMMITTED** permits dirty reads, non-repeatable reads and phantom reads.
- **TRANSACTION_READ_COMMITTED** permits non-repeatable reads and phantom reads.
- **TRANSACTION_REPEATABLE_READ** permits only phantom reads.
- **TRANSACTION_SERIALIZABLE** does not permit any of cases 1-3 to occur.

All beans must have a transaction isolation level declared.

**1.2.2.2 Support for Management of Multiple Instances** [E1, E5]
A stateful Session bean can be used by only one client, whereas a stateless Session bean or an Entity bean can be used concurrently by several clients. The EJB container
is responsible for handling these concurrent requests, and therefore several tasks might be performed:

- Instance pooling
- Database connection pooling
- Instance passivation
- Pre-cached instances
- Optimised method invocations

It should be pointed out that EJB Specification 1.0 requires only instance passivation among the mentioned tasks (the container provider has been given a detailed approach for this mechanism). Vendors could still easily provide the other tasks.

**Instance pooling** means that several clients can access the same instance of a stateless Session bean or an Entity bean, as shown in **Figure 1.22**. Thereby the container can get away with fewer instances of each bean class, whereas for stateful Session beans there must be a bean instance per client.

**Figure 1.22** Instance Pooling increases scalability

While instance pooling is mandatory for stateless Session beans, the mechanism is optional for Entity beans. But stateful Session beans can never be shared among several clients, and therefore instance pooling is never used there. Note that EJB based systems are expected to be portable, meaning that beans running on one EJB server should be easy to move to other vendors’ servers. The designer of the EJB components should have this in mind when using the optional vendor-specific mechanisms supported by the current EJB server. For example if we let our Entity
beans be active and make calls on other system components, then we must have instance pooling for Entity beans used on our EJB server, which implies that the system will not work on servers that do not use this mechanism.

**Database connection pooling** is about reusing pooled database connections for different beans (see Figure 1.23). This mechanism increases efficiency since the overhead of creating new database connections is avoided. Also the database server’s work gets easier as it will not have to maintain an open connection the whole time between frequent access requests.

![Figure 1.23](image)

**Figure 1.23** Database Connection Pooling reduces overhead and increases capacity when using databases

**Instance passivation** occurs when a bean is swapped out to storage. When the container needs space, a bean may be swapped out. Session beans are passivated only to leave space for the container, whereas passivation of Entity beans is used for synchronisation of the bean's state with the underlying database.

**Precached instances** can be used at bean creation time in order to speed up the initial loading of database information. The loaded information can be placed either in memory or in a highly indexed database table.

**Optimised method invocation** can be used to avoid the overhead of the RMI protocol in the communication between beans in the same container. The container can let a bean call another bean's method locally. Thereby the container avoids the
costly work of serialising the arguments and using stub and skeleton objects. But, the call still goes via the proxy object of the called bean, as shown in Figure 1.24:

![Figure 1.24 Optimised method invocation: RMI communication between two beans in the same container is short-circuited](image)

### 1.2.2.3 Support for Persistence [E1, E2, E4, E5, E8]

Entity beans are objects that have states, i.e. they have fields that can be given different values during a conversation. These values are stored in a database or a file. Session beans can always read and save persistent data explicitly, for example by using SQL statements with JDBC. But later, when the Session bean has been removed or the server has been restarted, the stored data must explicitly be read into some object again. The purpose of Entity beans is to have components that correspond to and reflect the stored data. So even after a server crash, you can create an Entity bean that automatically has a state that reflects the stored values of its fields.

EJB provides two ways for storing and updating the persistent values of Entity beans. In **container-managed** persistence, the bean class does not make any explicit storage operations using SQL or similar. All the bean author does is specify the field that should be persistent, and the container does the rest.

In **bean-managed** persistence, the bean class is written using common JDBC commands and the persistent fields are stored/loaded explicitly.
Entity beans can use only **container-managed transactions**.

**Container Managed Entity Bean**

```java
public interface myContMan ... {
    public String getName() throws ...;
    public void setAge(int age) throws ...;
}

// Bean class
public class myContManBean ... {
    public String name;
    public int age;

    public String getName() ... {
        return name;
    }

    public void setAge(int a) ... {
        age = a;
    }

    ... // create, remove, finders
}
```

**Bean Managed Entity Bean**

```java
public interface myBeanMan ... {
    public String getName() throws ...;
    public void setAge(int age) throws ...;
}

// Bean class
public class myBeanManBean ... {
    public String name;
    public int age;

    public String getName() ... {
        refreshFromDB(); //SQL-JDBC
        return name;
    }

    public void setAge(int a) ... {
        age = a;
        insertIntoDB(); //SQL-JDBC
    }

    void refreshFromDB() {
        ... "select NAME from Table1 ..."
        name = ...getString();
        age = ...getInteger();
    }

    void insertIntoDB() {
        ... "update Table1 set AGE ..."
        ... setInteger( age );
        ... setString( name );
    }
}
```
1.2.2.4 Support for Security

To support security, EJB uses an **access control list (ACL)**, which is a list of groups or persons that are permitted to call all or a subset of a bean's methods. A caller may have to provide name and password when it wants to access a bean. Software and system designers should introduce security into the EJB-based system in an early stage of the design phase. Normally, when there are multiple users in a system, the users are grouped with respect to the different tasks and problem domains that they are involved with. These classes and groupings of users are typically defined depending on what kind of information each group is permitted to access, and also depending on what actions the group members are allowed to take. [E1,E5,E6]

The process of specifying the groups and list of users in ACL is vendor-specific and differs between available EJB servers. For instance, in Weblogic you may write something like the following in the in the **weblogic.properties** file:

```
weblogic.password.operator=XXX
```

The ACL would then look like:

```
weblogic.allow.execute.weblogic.myPackage=operator,admin,mans,hama,richard.
```

As an example, let’s say that we have two groups of users: operators and administrators (and also three other people who are not part of these groups). The operator is a basic user that has typically the least (default) level of permission when accessing the system components. This means that an operator may not be able to access any of the methods of bean A and only have permission to invoke method `foo()` of bean B. An administrator on the other hand, would typically be able to access both beans and execute all their methods. In the **Deployment Descriptor** of bean, we can specify which user group is allowed to access what methods on the corresponding bean, as shown below: [E1,E5,E6]

```
(accessControlEntries
  Default [operator admin hama mans richard]
  "acknowledge" [admin hama]
  "subscribe(ConfidentialAlarmInfoType)" [admin mans richard hama]
); end accessControlEntries
```

When a bean calls another bean it can either use its own identity (the group it belongs to) or its own client’s (caller’s) identity. This can and should be specified when the bean is designed, and it helps us control and strength security in deeper part of the system.

47
1.2.3 EJB Server [E1,E2]
The EJB server contains the EJB container, and it provides low-level services, e.g. network connectivity, to the container. In version 1.0 of the specification, the interface between the EJB server and the EJB container is not specified. Even if this interface is left as a choice to the vendor, at the logical level, the server does not interfere in the calls between clients and beans. This means that the vendors must let beans and clients access the container - never access the server, whatever interface there is between the container and the server.

1.2.4 Interfaces [E1,E5,E8]
There are several other interfaces used in EJB, but since they are used in the bean classes, we will introduce them when we discuss the EJB beans. The author of an EJB bean must provide two interfaces and a class. The home interface declares methods for creating, removing and/or finding EJB beans. The remote interface contains the business logic methods of the EJB bean, and the bean class must contain the implementation of these methods, but the bean class is not declared to implement the remote interface. The bean author does not write the implementation classes for the home and remote interfaces; the implementation classes are generated at building time, and the container will instantiate them.

1.2.4.1 The EnterpriseBean Interface
The interface EnterpriseBean is the top-level type in the EJB class hierarchy. It works as a common super type for session and Entity bean interfaces, and it makes all beans serializable, as shown in Figure 1.25:

```java
public interface javax.ejb.EnterpriseBean
extends java.io.Serializable {}
```

**Figure 1.25**
Top level hierarchy of EJB interfaces
Since the transaction attributes are the same for both entity and Session beans, logically they can be seen as part of the EnterpriseBean interface, although they are not directly associated with it.

1.2.4.2 The Home Interface

The home interface of an EJB bean must extend the `EJBHome` interface:

```java
public interface javax.ejb.EJBHome
    extends java.rmi.Remote
{
    public abstract EJBMetaData getEJBMetaData();
    public abstract void remove(Handle handle);
    public abstract void remove(Object PrimaryKey);
}
```

The methods of the `EJBHome` interface are implemented in the object that has been instantiated by the container, since that object implements the home interface of the EJB bean which in turn extends the `EJBHome` interface. When a client uses a naming service to lookup an EJB bean, the naming service returns a reference to the object that implements the home interface, as illustrated in Figure 1.25:

```
A1: Context context = create a context for JNDI/COS
A2: MyBeanHome home = (MyBeanHome) context.lookup("beanPackage.MyBean");
```

**Figure 1.25** Getting a reference to the home interface
The class implementing **EJBMetaData** is used at building time in order to generate wrapper classes for method invocations on the beans.

The implementation class of the home interface implements both the create/find methods known by the bean’s client, and the methods of EJBHome interface.

**getEJBMetaData()**
returns a reference to an object that implements the **EJBMetaData** interface:

```java
public interface javax.ejb.EJBMetaData {
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
}
```

```java
MyBeanHome
extends MyBeanHomeImpl
implements MyPrimaryKey
```

```java
//Interface written by bean author
MyBean create(); // Only sessionBean
MyBean create( … ); // Only stateful sessionBean
// and entityBean
void/MyBean findByPrimaryKey( … ); //Only for entityBean
void findByPhoneNum(...); //containerManaged entityBean
```

```java
//Class generated by builder tool
EjbMetaData getEJBMetaData();
void remove( Handle myBeanRef );
void remove( Object primKey ); // Only for entityBeans
```

Figure 1.26 The implementation class of the home interface implements both the create/find methods known by the bean’s client, and the methods of EJBHome interface.

Figure 1.27 The implementation class of the home interface is used at build time to provide the proxy implementation classes (home and remote) and the primary key class used by entity beans.
public abstract Class getPrimaryKeyClass();
public abstract Class getRemoteInterfaceClass();
public abstract boolean isSession();
}

**remove (Handle handle)** removes an EJB instance using a **Handle**, i.e. a serializable reference to the bean. Having a serializable reference to a bean enables the container and other users to send the bean to temporary storage in order to restore it later on. This facility also gives us the possibility of letting several clients have access to a single Entity bean (only Entity beans may be accessed by more than one client). The component that is responsible for removal of a bean can use the handle to remove the bean, as shown in **Figure 1.28** (see also **The Remote Interface** in 1.2.4.3):

![Diagram of remove (Handle handle)](image)

**Figure 1.28** Using a handle to remove a bean
**remove (Object primaryKey)**

is used only for Entity beans, since they are the only beans that have primary keys. This method removes the Entity bean, which means that the row/object in the database table will be deleted, as shown in Figure 1.29:

```
class MyClient {
    MyEntityHome home = lookup(...);

    PrimKey key = new PrimKey();
    key.value = 43;

    MyEntityBean bean;
    bean = home.finByPrimaryKey( key );

    bean.getSalary(); // Use bean

    home.remove( key ); //removes bean
}
```

**Figure 1.29**
Calling the remove( primaryKey ) method on the home interface

As we saw, there were no methods in the EJHome interface for creation of new beans, or for finding existing beans. It is the responsibility of the bean author to write those methods in the home interface of the bean.
Create-methods in the home interfaces must be declared to throw javax.ejb.CreateException, and the finder methods for Entity beans must be declared to throw javax.ejb.FinderException. All methods in the home interface must throw java.rmi.RemoteException.

The home interface of a stateless Session bean must have only one method for creation of the bean. This method can not take any arguments, since stateless Session beans should not have any object variables that can be assigned values given by the clients.

```java
public MyStatelessSessionBean create()
    throws CreateException, RemoteException;
```

A stateful Session bean can have several different creation methods declared in its home interface. This is practical, since the bean can be given an initial state when it is created.

```java
public MyStatefulSessionBean create()
    throws CreateException, RemoteException;
```

```java
public MyStatelessSessionBean create( int x_init )
    throws CreateException, RemoteException, MyBeanException;
```

```java
public MyStatelessSessionBean create( int x_init, int y_init )
    throws CreateException, RemoteException;
```

The home interface of an Entity bean must contain methods for both finding and creating beans. The creating method should take all the persistent fields as argument, while the finder method takes the primary key of the bean as argument.

```java
public MyEntityBean findByPrimaryKey ( MyPrimaryKey primaryKey )
    throws FinderException, RemoteException;
```

```java
public MyEntityBean create(ColumnType1 value1, ... , ColumnTypeN valueN)
    throws CreateException, RemoteException;
```
1.2.4.3 The Remote Interface

Each EJB bean has a remote interface that is provided by the author of the EJB bean. The remote interface contains all the methods that are accessible to the clients of the bean, and all methods are declared to throw java.rmi.RemoteException. The bean class must implement all the methods of its remote interface. These methods are the business logic methods of the component. The remote interface of an EJB bean must extend the EJBOBJECT interface. The relationships are shown in Figure 1.30:

```java
public interface javax.ejb.EJBOBJECT
    extends java.rmi.Remote
{
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey ();
    public abstract boolean isIdentical();
    public abstract void remove();
}
```

- `getEjbHome()` returns a reference to the object that implements the bean’s home interface.
- `getHandle()` returns a handle, i.e. a serializable reference to the bean. The type returned is the interface Handle, which provides a method for recreation of a remote
reference to the bean. This mechanism can be useful if you want to save the reference to bean somewhere, for example in a database, in order to convert it to a reference later on.

*getByPrimaryKey()*

is to be used only on Entity beans, and it returns the bean's primary key. Only an Entity bean has a primary key.

*isIdentical()*

is to be used only on Entity beans. If two Entity beans are from the same home interface and have the same primary key, then they are considered to be identical and the method returns true. Since Session beans don't provide their identities, the method can not be used on them. Containers are free to choose between two instantiation policies for Entity beans. The first approach is to have multiple instances with the same primary key (these instances are considered to be identical), and let each client access its own instance of the bean. The second approach is to have a single instance of the Entity bean, so that two instances can not have the same identity, i.e. the same primary key, and let the clients access the instance serially.

*remove()*

removes the bean instance.

The bean class should not be declared to implement its remote interface. The reason for that is to ensure that the client's calls always go through the bean's proxy object. It is possible for the bean instance to implement its remote interface, but this should be avoided, since such a bean could give its own reference to a client. Then what might happen is that the client can call the bean's methods (the same methods as in the remote interface) without going through the bean's proxy object, and thereby make it impossible for the container to interpose in the client's conversation with the bean. The server would not be able to manage the client's calls with respect to the transaction policy that the bean author specified, nor manage the states and accessibility of multiple instances of the bean. **Figure 1.31** illustrates an example for this:
On the other hand, a class implementing an interface gains the benefits of compile-time type checking for the methods of the interface. There is a way to keep the advantages of type checking without violating the rules of object access in EJB. Instead of declaring the methods directly in the remote interface, we could have a non-remote interface that includes all methods. Then we let the remote interface be implemented by the bean class, and extended by the remote interface. The remote interface should extend the interface `javax.ejb.EJBObject` as before, as shown in Figure 1.32:

![Figure 1.31](image-url)

**Figure 1.31** StupidBean implements its own remote interface, and thereby makes itself available for calls that avoid its proxy, which means that the container becomes unaware about those calls.
1.2.4.4 The EJBContext Interface

The interface EJBContext declares methods that enable a bean to interact with services provided by the container:

```java
public interface javax.ejb.EJBContext
{
    public abstract Identity getCallerIdentity();
    public abstract EJBHome getEJBHome();
    public abstract Properties getEnvironment();
    public abstract boolean getRollbackOnly();
    public abstract UserTransaction getUserTransaction();
    public abstract boolean isCallerInRole(Identity role);
    public abstract void setRollbackOnly();
}
```

The EJBContext interface is not used directly by the bean implementation classes. Instead one of its two derived interfaces SessionContext and EntityContext are used in the bean instance depending on whether it is a session or Entity bean. All bean implementation classes must implement the method `setSessionContext/setEntityContext`
in order to be called by the container. The container calls this method on the bean instance after it has been created or activated but before the bean is ready to be accessed by the client(s). The $ctx$ reference makes it possible for the bean instance to get information about the state of the environment dynamically during its lifetime.

Let’s get on with the methods of **EJBContext**:

**getCallerIdentity()**

Returns a `java.security.Identity` object that represents the client's identity. Combining this method with `isCallerInRole` makes it possible for the bean to do authorisation checks before it performs an operation.

- This class `java.security.Identity` is no longer used in Java 2. Its functionality has been replaced by `java.security.KeyStore`, the `java.security.cert` package, and `java.security.Principal`. This change has be reflected in EJB Specification 1.1, where the following method has replaced `getCallerIdentity()`:

  ```java
  java.security.Principal getCallerPrincipal();
  ```

- The mechanism by which the client's identity is given to the container is left to the vendors; it has not been specified in the EJB Specification 1.0.

- An alternative approach for authorisation is given by using ACLs on the container in order to control access to beans and methods.

**getEJBHome()**

A bean can obtain a reference to its own home interface, in order to create or find other instances of itself, or to provide the reference to another object.
**getEnvironment ()**
This method has been deprecated in EJB Specification 1.1. Use the JNDI naming context java:comp/env to access enterprise bean's environment. A list of the properties that have been set at deployment time is returned.

**getRollbackOnly ()**
Checks if the caller's transaction has been marked for rollback. Thus the bean can avoid to take part in a transaction that is going to be rolled back anyway. This feature is helpful in distributed transactions.

**getUserTransaction ()**
Returns an object of type javax.transaction.UserTransaction (in EJB Specification 1.0 the return type's package was javax.jts). This method can only be invoked by a bean that manages its own transactions, i.e. a bean that has the transaction attribute TX_BEAN_MANAGED. Actually, going via UserTransaction is the only way for a bean to access the underlying transaction manager, and since Entity beans can have only container-managed transactions, only Session beans can use this method.

**isCallerInRole ()**
This method is deprecated in EJB Specification 1.1. It has been replaced by the following method:

```
boolean isCallerInRole (String roleName)
```

The name of the security role (the argument of the method) must be one of the security roles that are defined in the deployment descriptor.

The UserTransaction interface includes the well-known operations "begin", "commit" and "rollback". The user of the interface can also see the status of the transaction, and set a timeout value for the transaction.

1.2.4.5 The SessionContext Interface
The interface SessionContext is implemented by the object that is passed by the container to a Session bean when the bean is created. The object is used by the bean for interaction with the container-provided services that are available in the EJBContext interface:
**getEJBObject()**

Returns a reference to the proxy object of the bean, i.e. an object that implements the EJBObject interface (the bean's remote interface, which is implemented by the proxy object, extends the EJBObject interface). Thus the bean can use this method to give another component a reference to its own proxy object. At the component level, this looks as if the EJB component gives its own reference to another object. For Session beans this method can be used only if the bean is stateless. The caller of getEJBObject() must be prepared to catch an exception when he uses the received EJBContext reference for calling the bean methods, since the bean may be used by others, e. g. the creator (see Figure 1.33):

```
public interface javax.ejb.SessionContext
    extends javax.ejb.EJBContext {
        public abstract EJBObject getEJBObject();
    }
```

```java
public class MySessionBean {
    SessionContext ctx;
    ABean bean= ...create();
    MySession proxy = (MySession) ctx.getEJBObject();
    bean.takeRef(proxy);
}
```

```java
public class ABean {
    public void takeRef (MySession s) {
        s.foo();
    }
}
```

```java
public class MySessionRemoteImpl
    class Client {
        MySession mine=...create();
        mine.foo();
    }
```

![Figure 1.33](image)
Figure 1.33 A stateful session bean cannot provide its remote proxy reference to a any object other that its own creator.
1.2.4.6 The EntityContext Interface

The interface EJBContext has the same role for Entity beans that the SessionContext interface has in Session beans. In addition to the method getEJBObject() that is analogous to the one in the SessionContext interface, there is one other method.

```java
public interface javax.ejb.SessionContext
    extends javax.ejb.EJBContext {
    public abstract EJBObject getEJBObject();
    public abstract Object getPrimaryKey();
}
```

**getPrimaryKey()**

Returns the primary key of the Entity bean. The primary key is supposed to be an object that contains data that can work as the primary key for the database table in which the bean keeps its persistent fields. Formally, any serializable type can be used as primary key.

The method getEJBObject() is in fact very useful for Entity beans. If an Entity bean EB wants to be called by an object X, it can use this method to provide a reference of its own proxy object to X. This way X will not have to know how to find, create or remove EB (only the client that created/found EB needs to have that information), as shown in **Figure 1.34**:

![Figure 1.34](image-url)

**Figure 1.34** An entity beans can provide its own proxy object in order to be called
1.2.5 EJB Beans [E1,E2,E8]

1.2.5.1 Session Beans

Session beans are meant to live during a client session. They are given a limited maximum lifetime at building time, and if their timeout expires during a session, they are automatically removed. This should not cause problems because the timeout can be given very large values. Session beans do not survive server crashes, and the information kept by a Session bean can not be restored when the server is restarted. For all Session beans the following statements hold:

- **Timeout** – there are two timeout values: one is the time a bean will remain active since its last usage by the client. The other timeout value denotes the maximum time a passivated bean remains in the pool.

- **Tied to a client session** – Session beans can not be accessed concurrently by several clients, and they are supposed to be removed either by the client that created them or by the container. Although stateless Session beans normally are used by several clients during their lifetime, they are not accessed by a second client if the first client is using the bean.

- **Don't survive server crash** – a client will not find its Session bean after a server crash. Not even passivated stateful beans are restored after a server crash or restart.

- **Not reentrant** – a Session bean being in an open transaction at the moment can not be called by other Java threads. We already said that Session beans will not be accessed by any clients other than their creator, but being non-reentrant means that not even the creator of the bean may let another thread call the bean while the bean is in transaction.

Stateful Session beans belong to the client that created them, and can not be used by other clients. They bear the name 'stateful' because they are able to keep the obtained information during the conversation with the client. They should die, i.e. be removed by the client when they are no longer needed, or they die when their lifetime expires. Stateless Session beans are meant to be used by several clients during their lifetime, although the same bean instance can not be accessed by more than one client at a time. These beans do not obtain any other state than the default state that they had at
Clients should not attempt to remove stateless Session beans, because this would only cause the bean to be moved to a pool, which is unnecessary, since other clients might need to access a bean right after the client who performed the 'remove' is done. Figure 1.35 shows the main difference between stateful and stateless Session beans:

**Figure 1.35** A stateful Session bean can never be accessed other than by its own creator. A stateless Session bean can be accessed by another client when its creator is not using the bean. But no EJB bean can be accessed by a client when the bean is being used by another client.

All Session beans implement the **SessionBean** interface. All the methods of this interface are callback methods that are invoked by the container to notify the bean when some event has occurred:

```java
public interface javax.ejb.SessionBean
    extends javax.ejb.EnterpriseBean
{
    public abstract void ejbPassivate();
    public abstract void ejbRemove();
    public abstract void ejbActivate();
    public abstract void setSessionContext (SessionContext ctx);
}
```
**ejbPassivate()**

Is called by the container right before instance passivation is done. Thus the method is called before the bean is swapped out to "semi-persistent" storage. The called bean has a chance to safely interrupt what it is doing, do necessary cleanup, and save necessary data in its object variables in order to be able to continue when it is activated later on. Passivation is done via standard Java serialisation, or by some vendor-implemented mechanism, for example storage in a database instead of a file. **Figure 1.36** illustrates passivation:

**Figure 1.36** Passivation is used on stateful Session beans in order to free up resources. Passivation increases performance and scalability.

**ejbRemove()**

Is invoked by the container when a client has called the Remove() method of the bean's home or remote interface. The called bean instance is going to be destroyed (or sent to the pool) by the container, but by this call, the bean has a chance to close any open resources and perform its final processing.

**setSessionContext (SessionContext ctx)**

Is called by the container only once when the bean is created. The argument passed by the container is an object that implements the SessionContext interface. The bean stores this object in an object variable, in order to use it
dynamically during its lifetime for interaction with the container-provided services such as security and transaction management.

**ejbActivate ()**
Is called by the container when a previously passivated Session bean's state has been restored from semi-persistent storage, but before the bean becomes available to the outside world. In this method's body, the bean performs the necessary actions to restore any resources that it closed before being passivated, e.g. it can reopen a file. **Figure 1.37** illustrates activation:

*Client has a ref to the bean, and does not see whether bean is active/passive.*

*Client calls the bean, so the container brings the bean back out of DB. Then container calls ejbActivate() so that the bean will be able to get the resources that it freed up just before it got passivated. When the bean is activated fully, it receives the client's call.*

**Figure 1.37** Activation is used on stateful Session beans in order to free up resources.
1.2.5.2 Entity beans [E1,E2,E5,E8]

In EJB Specification 1.0, the support for Entity beans is not required. This support was planned to become mandatory in Specification 2.0. But due to the fact that almost all EJB vendors provided support for Entity beans in their EJB products, and since Entity beans make life much easier, the support became mandatory in Specification 1.1.

Entity beans are simply used for data storage (well, there are actually cases where some Entity beans can be used without being persistent, and we shall talk about that later). They are meant to contain and mirror data saved in a database. An Entity bean usually corresponds to a single row in a database table, but it could also contain data from more than one table or from several databases. They can perform transactions, and they get transaction support as well as support for persistence from the container. Entity beans are therefore more than the data stored in database; they contain business logic methods.

An Entity bean can be accessed by more than one client, but the vendor may avoid this by letting each client have its own instance of the bean. To put it more precisely, the logical access model for Entity beans allows the bean to be accessed by multiple clients see (Figure 1.38). Of course, that means concurrent access by multiple clients – not simultaneous access by two or more clients. [E1, E5, E6, E8, E9]

But the actual implementation of client access mechanism on Entity beans is left to the vendor, and there are two options. In the first case the same bean instance is accessed by multiple clients, so the concurrent client calls are queued by the container
to ensure that only one request at a time is allowed to access the bean, as shown in

**Figure 1.39:**

If the implementation mechanism allows clients to access the same Entity bean instance, the clients must be queued.

The latter case creates a new instance of the Entity bean for each client that wants to access the bean, but the bean instances have the same instance id – i.e. the instances are logically identical, and they are indeed considered to be identical by container (see **Figure 1.40**). The container leaves the synchronisation issues regarding the bean instances’ states to the underlying database. Since several instances of an Entity bean that have the same home interface and the same primary key value are considered to be identical, the clients will not see any difference between the two approaches, and logically, an Entity bean is accessed by several clients in both cases.

Leaving the implementation mechanism of multiple clients’ Entity bean access to vendors has consequences not only for the scalability and performance of the system (parallelism vs. serial access, and number of instantiated beans at runtime), but also for the portability of EJB. Let’s remember that EJB claims to be portable – in fact,
EJB is said to be “the core of J2EE” ([E5] see Sun’s FAQ page on EJB), and since the point with J2EE is portability and “write once, run anywhere”, this does not look too good. Let’s see what the problem is:

We have a singleton component (e.g. a CORBA object) that communicates with a target, and we have several clients (c1,c2,...) that communicate with the singleton through the same Entity bean (EB with instanceId=13). Our design is going to be dependent on the EJB product that we intend to use, since the question whether each client (c1,c2,...) will have its own Entity bean instance is very critical for our system behaviour.

In Figure 1.41 the implementation mechanism of multiple clients’ access on EB results in having TargetConnector deal with three different invocations, and each invocation in done in a separate transaction context. Indeed, TargetConnector must be prepared to deal with concurrent calls from the EB component (neither the clients C1-C3 nor TargetConnector can normally detect whether there is one or several EB instances), and this is the only way to make our code portable. But wasn’t one of the benefits of EJB its promise about taking care of concurrency issues? I really think that having this vendor-specific option on client access for Entity beans can become one of the pitfalls of EJB. Let’s see what the same system would look like on another EJB product that uses serial access on a single Entity bean instance for multiple clients (Figure 1.42):
Entity beans are long-lived. They are meant to exist beyond the lifetime of a single client, and they also survive server crashes. Entity beans must use the container-provided transaction management service, so they can not manage their own transactions. That is, an Entity bean can not be declared to use the transaction attribute TX_BEAN_MANAGED. Since Entity beans have database connections, another limitation (the limitation is also valid for Session beans) should be pointed out too: only beans with the transaction attribute TX_NOT_SUPPORTED are allowed to use the JDBC primitives "commit" and "rollback". An Entity bean is either bean-managed or container-managed.

**Container Managed Entity Beans (CMEB)**

The storage of the persistent fields in a container-managed Entity bean (CMEB) is handled by the container (see 1.2.2.3). This means that the bean author does not write code for JDBC connections and SQL statements. All he does is declare the variable names of the persistent fields of the bean class, and map each name to the name of the corresponding column in the database table. This is done in a descriptor file for the bean at deployment time. CMEB should be the kind of Entity bean that is used normally, since the code volume will be reduced by around 40%, and thereby the stability, debugging and maintenance of components will be much easier.

The bean class of a CMEB does not contain any implementation of the finder method findByPrimaryKey(...) that is declared in the home interface. Instead, the container handles the task of finding an Entity bean with the supplied primary key. Thus, to set
or update the persistent values of a CMEB, you just assign values to the bean variables - that's all.

In EJB 2.0 the architecture of CMEB has been changed. The bean developer writes an abstract bean class that implements the interface `javax.ejb.EntityBean`. Another class generated at build time will extend the abstract bean class, and this subclass will be the actual bean instance’s class. Another new feature in EJB 2.0 is the standardised syntax for CMEB finder methods. The syntax is written in `EJB QL`.

Bean Managed Entity Beans (BMEB)

In a bean managed Entity bean (BMEB), the bean must perform common JDBC actions to create a new row in a database table, or to update or delete an existing row. If the bean's data represent data from one database table, and if the representation is straightforward, then you should use container-managed beans. Bean-managed persistence is better suited for beans that represent aggregations of data in several underlying tables, or if there are complex processes in the conversion between the data stored in the database and the data represented in the bean. Figure 1.43 shows an example for using BMEB: we have the basic colours red, blue and green stored in our database and we want to used the colour purple. So we read red and blue from the DB and process the data (mix the two colours) to get purple, which can be represented as persistent data. To make the persistence of the complex data more comprehensive, we could think of money: we have only Swedish Crone and Euro in our database tables, and we want our Japanese customers to see values in Yens.

![Figure 1.43](image-url) Usage of bean managed entity bean
I have implemented and tested the performance of both CMEB and BMEB, and I found no difference in the execution time.

But the storage-modelling relation of complex data is not the only case where BMEB is useful. When we started talking about Entity beans, we said that they are normally persistent, but added that not all Entity beans need to be persistent. First of all, CMEB are always persistent since they expect a database table to be specified in the deployment descriptor. That is, they are always associated with stored data. But BMEB are not necessarily persistent, since all database related code, including table names, are written by the bean author in the implementation class using JDBC and SQL. This means that we can write a BMEB without making any database connections, and what we get then is a non-persistent bean that can logically be accessed by multiple clients. In other words, we get almost a Session bean with container-managed transaction (all Entity beans have that) that can be accessed by several clients, as shown in the example of Figure 1.44:

Thus we have a component that can be accessed (found) by several clients, and in the more special case, it can gives its own proxy reference to a server object (subscribe on data from a target) that will call the component (make call-back). This usage of BMEB seems to be very rare, and I have not seen anything about it in the EJB literature, and I found out about this non-persistent component months after I had finished my implementations for this Thesis. Therefore, I have not used it in my code.
1.2.5.3 Restrictions on EJB Beans  [E1, E6, E7]

Some of the most important programming restrictions for EJB beans of specification 1.0 are mentioned in section 16.4 of the specification. The restrictions of EJB 1.1 are specified in section 18.1.2 of the specification. Here are some of the restrictions on EJB 1.x:

- A bean can not start, stop, suspend or resume threads, neither can it manage threads (using synchronisation) or thread-groups. A bean can not change a thread’s name or priority.

- A bean cannot make direct use of the underlying transaction manager. There is one exception to this rule: a bean with the transaction attribute TX_BEAN_MANAGED uses javax.jts.UserTransaction (in EJB 1.1 the interface is moved to javax.transactions.UserTransaction) to access the underlying transaction manager and thereby manage its own transactions. Since Entity beans can not manage their own transactions, the exception includes only Session beans.

- Transaction-enabled beans (beans with a transaction attribute other than TX_NOT_SUPPORTED) cannot call JDBC commit() or rollback() methods

- Beans can not use java.io facilities such as file streams.

There are several other things to be careful with when you write EJB beans. For instance, sometimes there are restrictions on inheritance and polymorphism. Another case is passivation of a bean that holds a reference to another bean. These and other problems are well addressed in FAQ pages of some EJB-server vendors, such as BEA Systems’ Weblogic (the server used in this Thesis work), Inprise Application Server and IBM’s Websphere.
1.2.6 EJB Roles  [E1, E2, E8]

Earlier we mentioned that one of the strength points of EJB was that it collected different kinds of support into one place and provided them to the users/clients. And as we said, these different areas of support were transaction management, security, persistence etc. The hope is, and the reality has been, that expert vendors in each area will use their knowledge and experience to provide all the stronger and better EJB servers and containers. For example, Oracle’s knowledge in databases and database servers is certainly used when an EJB server vendor co-operates with Oracle in order to provide better persistence support and improved database access performance. IBM’s well-known transaction server and the company’s experience in large-scale transaction management should hopefully add advantage to their EJB server on Websphere. Inprise’s strength in CORBA is hoped to increase EJB-CORBA integration on their Application Server and so on. We will present some currently existing EJB servers towards the end of this paper. In addition to the informal division into knowledge and experience areas, there is also another kind of role division that is there to increase development and shorten time-to-market in EJB based applications and processes.

In the EJB specification specific roles are defined for the application development and deployment process:

- Application Developers
- Application Assemblers
- Bean Deployers
- Server Providers
- Container Providers
- Administrators

**Application Developer**

Defines the home and remote interfaces and the implementation classes (bean classes), the primary key class for Entity bean(s) and the resource requirements in the Deployment Descriptor. This role can also be referred to as Enterprise Bean Provider, and the person or group of people having this role need to be concerned only with the business logic of the application.
**Application Assembler**
Builds application using EJB classes by producing JAR files that contain EJB assemblies or complete applications. This role might also include the task of providing a user-interface client program.

**Bean Deployer**
The task of this role is to prepare the EJB assemblies for execution by taking an EJB-JAR file and install it into an EJB container. The deployer configures various bean properties, binds resource references to resources, provides database connections and connection pools, JNDI naming context and services, as well as JMS destinations and services. Some server vendors provide EJB deployment tools, e.g. BEA’s Deployment Wizard for Weblogic.

**Server Providers**
The idea is that an EJB server provider will be an expert in concurrent and distributed programming, transaction management and network communications. The server provider provides a server for hosting EJB containers, and implements the underlying services that the container needs in order to do its job. The server provider is also responsible for enabling efficient execution of beans, as well as administration tools.

**Container Provider**
The container provider must provide support for transactions, security and persistence to the beans, and it is also responsible for running the beans and taking care of beans’ contact with the outside world. Some other responsibilities and tasks of this role are: generating classes that comprise the container, installing executable artefacts into the server registry, participating in deployment of beans and implementing the EJB container contracts. The interface between EJB container and EJB server is still not standardised, and therefore the role of container provider is often played by the server provider.

**Administrator**
The administrator is responsible for monitoring the performance of the EJB server and the resource usage, and for keeping security and resource information up-to-date. Other tasks of this role are: managing the underlying database engine and initialising the name and directory contexts.
1.2.7 EJB Specification Versions [E5, E8]

EJB Specification 1.0 was introduced in 1998. Sun Microsystems had previously released an outcast to be studied and commented on by expert companies and ‘gurus’. In EJB 1.0 the usage of Entity beans was optional and the plan was to wait until specification 2.0 before these persistent beans would become mandatory. But since all or most vendors implemented support for Entity beans in their EJB containers, these beans became mandatory already in specification 1.1 released in 1999. In versions 1.0 and 1.1, finder methods for container managed Entity beans were on the list of things that would be standardised in the future. In Weblogic there was a vendor provided syntax for writing such finder methods already in the implementation based on EJB 1.0.

Some of the improvements done in EJB 1.1 were:

- The file format of the Deployment Descriptor (DD) was changed to XML.

- Also the content of DD was modified and restructured so that it would better help the separation of the expert roles Bean Provider (BP), Application Assembler (AA) and Bean Deployer. The DD would not contain any information provided by BP.

- Security: BPs are now able to build their beans without knowledge of the target security domain. A BP may now write components with little or no access control logic; the AA would then add method-level access control using application security roles, and Bean Deployers could later map AA-defined security roles to users and security groups.

- The specification was tightened up to improve cross-server portability of EJB. The specification was also made clearer in some areas so that different server vendors’ EJB implementations would be consistent.

The first public drafts of Specification 2.0 were released some time during early 2000. The main changes in this new specification are:

- The inheritance and implementation structure used for Entity beans has been changed.

- Finder methods of container managed Entity beans are standardised.
• The programmatic support for security has been removed, which means that only declarative security support is provided.

• Introduction of a new bean: MessageDrivenBean. This new bean enables asynchronous communication, and it is implemented in integration with Java Message Service (JMS). One of the main benefits of this new component is reduction of clients’ wait-time as well as elimination of bottlenecks, as clients can send their requests (messages) and disconnect immediately.

In the yearly held conference JavaOne, EJB has been one of the main headlines, and it has won more focus for each year. At JavaOne 1998 EJB 1.0 was known to have few shippings, lots of promises and only Session beans as mandatory. One year later, EJB 1.1 was part of the Java Enterprise edition J2EE, and now there were around 30 EJB products (with several free evaluation licenses), all of them implementing Entity beans. JavaOne 2000 was held in early June this year, and this time EJB 2.0 was one of the main topics.
2. APPLICATION

2.1 The Choice of Application

The aim of this Thesis is mainly to study and describe EJB, and to investigate it in practice by using it in an appropriate application that requires the kind of services provided by EJB. Earlier we talked about several features and properties of the EJB specification, such as scalability, portability and its being component-based. These issues are of course important and must be fulfilled by the EJB server that a company will buy for its applications. Still, EJB servers are supposed to provide other services that are known to give EJB its strength: transaction management, persistence and security. The application described below has been chosen with these services in mind.

The application is an alarm handling application that is basically a client-server system. Clients register themselves in as subscribers of some target systems that send alarms to the server. The alarms sent from a target system to the server must be delivered to those clients that are registered as subscribers of that specific target. So far the application sounds like a typical case for notification services, such as the one provided by CORBA products. The alarms received by the server will be stored in a database so that the clients will be able to look at older alarms. This means that the application also calls for persistence support, and since we don’t want to force each client to get a database of its own, the database handling must also be done by the server. But support for persistence is also provided by CORBA products. If we want to convince our boss to buy this hot EJB stuff we have to do better than this. Figure 2.1 illustrates the usage of notification services. The black arrows from client to server denote requests for
registration. Client 1 has registered itself on targets A and B, while client 2 is registered on targets B and C. Both clients also see old alarms stored in the database.

Well, in our well-chosen application, the operators using the client programs will be informed about arrival of new alarms. Normally an operator will want to take care of some alarms, and before taking any other action he or she must tell all the other operators about that decision. In order to do this, the operators will update the state of the specific alarm A by writing their own unique identity into a field of the alarm. In other words, the operator will have the client make a request for acknowledgement of the alarm A. Successful acknowledgement of alarm A should result in notification of the updated alarm to all clients that are subscribed on the target system that A came from. This means that all the clients that already had received A will get a new alarm A’ which contains the acknowledging operator’s identity. Note that the state of the alarm will be updated also on the target system (there is a list of active alarms on each target), and in the database as well. What are we doing now? We are performing transactions, as shown in Figure 2.2:

![Figure 2.2](image)

**Figure 2.2** Our application demands transaction services

And EJB provides support for transaction management. The reader might know that even transaction services are easy to find on the market, both on the CORBA and Microsoft sides. Now it is about time to point out the strength of EJB: EJB is a specification that has been created and is being developed in order to provide all the mentioned services along with others such as security, support for pool instantiation, instance passivation etc. We would also like to be able to control the visibility of functionality and data so that we can discriminate between different groups of operators.
In other words, we are also interested in EJB’s support for security, as illustrated in Figure 2.3:

Another strength of EJB is its declarative form of support for transaction and security, and that you declare these properties at deployment time. Still another strength is that you deal with components. Components are or at least should be reusable and coarse grained, and component-based development is known to help you save time and money. It just makes it impossible for you to do the same mistakes that you or others have been doing over and over again when writing troublesome code to handle concurrent transactions or updating your database using SQL command lines.

So if you believe that you do better without EJB in your kind of application area, you may need stronger arguments that simply “this service and that service is provided elsewhere”. EJB offers an opportunity of having all these services in one place combined with the all the more frequently used component-based modelling. EJB does not suite all areas or all applications, and it certainly will cost you execution time, memory and system resources - it is simply not a silver bullet. We will examine EJB based solutions for our alarm handling application in order to get a picture of the practical advantages and limitations of the technique. But to begin with we can state that our application certainly could use the services provided by EJB.
2.2 Application Description

Our system has three parts: client(s), an application (EJB) server and target system(s). There is also a database that should actually be considered as the fourth part of our system, but we have chosen to see the database as part of the middle-ware server. Note that there are normally several clients and several target systems. The client is interested in the target system, and the server works as a middle-ware carrying the business logic part of the system as well as the services needed such as transaction management, persistence and security. Thus the server provides services to the client, and the client will never have direct access to the target system, as shown in Figure 2.4.

![Figure 2.4](image)

**Figure 2.4** Our EJB based server is a middle-ware having access to a DB

The target system could realistically be e.g. an AXE system, or as in our case the distributed operating system TelORB, which is used for telecom applications. On TelORB there is an entity, called AlarmTool (see Figure 2.5), responsible for activation of an alarm when something goes wrong. As users or supervisors of some TelORB based entity/system, the clients want to be made aware of the alarms, and probably do something about them.

![Figure 2.5](image)

**Figure 2.5** AlarmTool generates alarms and is TelORB-based. AlarmHandler is a CORBA object that sends alarms to its subscribers AlarmConsumer (also CORBA), and takes transactional requests from them.
Sequence diagram for the application

Client

getTargetIORList()

Find needed EJB components using naming service on server

Use EJB components (beans) to subscribe on targets’ alarms

Server had already subscribed on some of the client’s targets, so here come their active alarms

Send the received spontaneous generic alarms to the clients that are registered as subscribers of the alarm source (target)

Acknowledge a specified alarm

Return the ack-result to the client that acknowledged

Send the ack-result to all clients that have subscribed on the same target

Get some stored alarms’ data

Send the found alarms

Ejb-based Middle-ware

If client asks for subscription on a new target, then create a new plug-in and make it subscribe on the target alarms.

But if we have already subscribed on the client’s target, then just add the client to the clientList of the plug-in, and send the active alarms to the client.

When new alarms are generated, or when other systems update alarms, they are sent to our server as new (spontaneous) target-specific alarms.

Convert target-specific alarms to generic alarm objects.

Save received generic alarms as persistent components. These components’ data are automatically written into the database.

Middle-ware Database

Find the alarms that match the client’s query

Target

Acknowledge a specified alarm and receive the result (a new alarm) synchronously, returned either by the plug-in or by the target
Being a user of the system we start a **client** program. The client is there to inform us about the state of the parts that we are interested in on TelORB. The client must be able to tell us right away when a new alarm is activated, and it must also let us do several things. We use the client to do the following things:

- Receive the alarms that were activated before the client was started, i.e. those alarms that are currently active on the target.
- Acknowledge alarms: we acknowledge on an alarm if we want to tell other users that we will attend to the alarm. Acknowledging an alarm results in update of the alarm on the target, after which the updated alarm is sent to all subscribers by the target as if it were a new alarm.
- Clear alarms: removes the specified alarm from the active list of the target object, followed by sending the removed alarm to all subscribers of the target. A cleared alarm will be marked as “cleared” when it is sent by the target.
- Reload alarms from target systems: all the active alarms of the specified target system will be sent to the client.
- Look at the information (alarms) saved in the server's database.

The **server** has to do several things: it must establish connection to the TelORB based entity that handles alarms. This entity is an object called alarmHandler, and lets the server subscribe on alarms. As soon as AlarmTool has an activated alarm, it sends the alarm to alarmHandler who in turn ships it over to all subscribers. The server must do two things with the arrived alarm: first the alarm must be stored in a database, and then it must be sent to the client(s).

The server has three communication channels: the plug-in part that is used for communication with the target system(s), a database connection, and an entity for communication with the client. Actually, the database connection handler can be viewed as an internal part of the server, which leaves us two channels.

The **plug-in** is the object that connects itself through RMI or CORBA to the target system and makes a subscription on alarms (in the case of TelORB CORBA is used). The plug-in object finds the TelORB based CORBA object by obtaining a reference of type AlarmHandler from the IOR string provided by the client. The client in turn can
either get this information via a name server, or have it hard coded if the target “always” is the same. In our case, the client expects the IOR string to be in a file in the current directory. When the plug-in object receives an alarm (an object), it makes a new object from it, i.e. it converts the specific alarm to a generic one. This makes the rest of the server independent from the structure – and thereby from changes – of the specific alarm(s). It also makes database handling much simpler, and makes it easy for the server to get alarms from different target systems.

A simple version of the system described above (without EJB) was implemented as a first-stage prototype, though without acknowledging of alarms. The next thing to do is to design the system considering the client’s needs, and of course, now we will be thinking in EJB. Although we already have talked about what the client wants to do, it is necessary that we go through a detailed specification of the client’s demands. It is important to know how often the client might want to make a specific request, and how big (in computational terms) a requested task can be. EJB has been introduced and developed regarding, among other things, efficiency and resource usage. If the client’s communication with the EJB server is done via RMI, we must think about which requests will result in transfer of objects to the client.

**Client**
The client will be able to do the following:

1. Get IOR string(s) of the target object(s), i.e. the object(s) that the server must communicate with. Each IOR string corresponds to a global name that is the target’s name. The client gets each target IOR from a file having the target’s name as prefix.
2. Send IOR and targetName to the server, and ask for subscription of information (alarms) produced by the target system (and received by the server).
3. View alarms that are stored in the database.
4. Acknowledge on active alarms.
5. Clear alarms (the process of clearing an alarm is the same as acknowledge and therefore it was not shown in the sequence diagram).

Active alarms are those that are known by a target system at the moment. Let’s assume that there are three alarms, A, B’ and D, known by the alarm handling entity at the target system T. These alarms were activated before our client C1 got a subscription through the server, and they are sent to C1 by the server. If the server already had a
subscription as default or due to another client’s request, then the active alarms of T are kept in some internal component of the server. But if the server has no subscription on T, then a plug-in object is created for communication with the target. Eventually the server sends the alarms A, B’ and D to the client C1. C1 observes that another client (C0) has acknowledged B’ (there must have been an alarm B, and C0 did acknowledgement on B).

Our client C1 acknowledges on alarm A. The target system T receives this information, generates a new alarm A’, and sends it via the server to all clients. Now, on T, the active alarms are A’, B’ and D. Here we could get a synchronisation problem: we (C1) can in some graphical interface see the list of alarms containing A’, B’ and D, but we have also A in that list. If C0 has been running since his acknowledgement on B, he can see A, A’, B, B’ and D. There are at least three ways to solve this problem. One is to let the client refresh himself now and then; preferably after receiving an alarm from the server. The second solution is to let the server have the latest version of the list of active alarms and send all clients a copy when a change occurs. If acknowledgements are rare this solution could work quite well. When the server has sent a client’s acknowledgement to the target system, the server asks for a re-subscription (for his own use) from the target system, and sends the arriving alarms to all clients. We prefer a third alternative: the client will replace the graphical representation of A with A’.

The alarms stored in the server’s database must be sent to the client as the client makes requests about this. Of course, only alarms from the specific target system (the client has specified which) is considered. For practical reasons only the latest alarms in the database should be sent to the client. We could have a default value for the amount of time going back, and the client has the possibility of looking back further. Updating the list of active alarms, or arrival of new spontaneous alarms do not affect the client’s version of the database, so the client must explicitly ask the server for update of his own database view.
Target System (TelORB)

We need to know two things about our target system TelORB: the IOR data for finding the CORBA target object, and the IDL interface `AlarmHandler` that is implemented by the target.

```idl
interface AlarmHandler
{
    unsigned long subscribe ( in AlarmConsumer consumer );
    unsigned long resubscribe ( in unsigned long consumerId,
                               in AlarmConsumer consumer );
    void cancelSubscription( in unsigned long consumerId );
    void acknowledge ( in AlarmUpdateAndClearInfoIn data, in string userId );
}
```

The first three methods are used by plug-in CORBA objects implementing the interface `AlarmConsumer`. These methods are used for starting, restarting and cancelling a subscription on the alarms generated by the target object. The fourth method is called by the plug-in as the result of a client’s request to acknowledge on an active alarm.

As you will see in the design descriptions (see 3.1.1.3), I have added a method named `syncAck` to the interface `AlarmHandler`. This new method is used in some designs and their implementation, and it gives us synchronous communication when the plug-in CORBA object `AlarmConsumer` acknowledges alarms.

Alarms

Spontaneous alarms could be real, new alarms. That is, there was no alarm with the same Id value (as the arrived spontaneous alarm) at the target, and then a new alarm was generated. But spontaneous alarms could also be the result of a change in an already existing alarm at the target. This could happen if the target object has other users than ourselves, and those users can perform updating operations on the active list of the target - just as we perform transactions when we acknowledge alarms. Thus, spontaneous alarms to us are all alarms that are not result of actions taken by our system. Note that even if we deal with both of these categories of spontaneous alarms similarly inside our system, on the logical level - especially when presenting data to the users this similarity might disappear. That is, we may present only real, new alarms as being spontaneous. Changed alarms are those that have been updated on the target system. A changed alarm is one that existed on the target system prior to a transaction that changed the state of the alarm.
3. DESIGN

There are four different designs. All of the design alternatives below have been implemented, and different kinds of management policies have been tried out for some of the implementations of the designs (bean-managed vs. container-managed Entity beans, or stateful vs. stateless Session beans).

- The first design lets all communication between the clients and the AlarmConsumer plug-in objects go through one single Entity bean called gateBean. In this design AlarmConsumer sends new alarms to an instance of a Session bean, not to the bean's proxy object, and thereby it breaks a rule of EJB. We still designed and implemented such a relation in order to see whether it ‘works’ at all (see Figure 3.1):

**Figure 3.1** Design 1 breaks against EJB rules; The stateful sessionBean should not be able to be called by several clients

- The second design makes use of an idea similar to the first one: an Entity bean takes care of requests from all clients and provides them to one single AlarmConsumer, but here the Entity bean’s proxy object is accessed by AlarmConsumer – the bean instance is never called directly, as shown in Figure 3.2:
The third design uses an Entity bean only to register new clients and/or to make new instances of AlarmConsumer (see Figure 3.3). The rest of the communication between clients and AlarmConsumer objects goes through two one-way Session beans that carry messages to specified destinations (see Figure 3.4). The Session bean used by the client can be either stateless or stateful and both alternatives are discussed in the design descriptions below.

![Diagram](image)

**Figure 3.3** Design 3 keeps the persistent single component (communication channel) of design 2 for the subscription phase
The fourth design is a modified version of the third one, and it is actually our most interesting design. Here we let transactions (acknowledgements on alarms) be handled by the Entity bean corresponding to the specified alarm (see Figure 3.5). Thus the persistent objects actively take part in transactions. Figure 3.6 shows the subscription phase.

- The fourth design is a modified version of the third one, and it is actually our most interesting design. Here we let transactions (acknowledgements on alarms) be handled by the Entity bean corresponding to the specified alarm (see Figure 3.5). Thus the persistent objects actively take part in transactions. Figure 3.6 shows the subscription phase.
Scaleability Issues

Before we get into our actual designs, let's look at an alternative faulty design (see Figure 3.7). We want to let several clients subscribe on one and the same target system. One way to do this is to establish as many connections to the target system as there are clients. This kind of solution has several principal disadvantages. The burden of the target system is by no means decreased, and the system will not be scalable. Still, if we are sure that some quantities will have a small upper bound, then this kind of design could suite our needs quite well as long as we remember the lack of scalability, generality and flexibility.

So we let all the clients have their own connection with the target system. Each client will have a plug-in object that communicates directly with the target object. In order to do transactions on shared information different clients must somehow be able to access the same instance of the component containing the shared information. This
component does not have to be persistent, but in this design we are actually forced to use persistent components in order to make them available to several clients.

We want all clients to communicate with the same instance that has a subscription on the target object. How do we do this? There are different solutions, and one of the differences between our designs lies in the different answers given. The first thing to do is to make the system have one single connection with the target system, so that the capacity of the whole system will not be bounded by the capacity of the target object. This means that no matter how many clients we have, there will always be only one object communicating directly with the target system. For simplicity, we omit the beans in Figure 3.8.
Throughput Issues

Above we discussed scalability with respect to the load on the target system: the number of connections and the amount of used resources (including memory) in the middle-ware will be affected by the amount of load on the target system. Of course, the amount of load on the middle-ware is a function of the amount of load on the whole system, but in order to achieve scalability we want to minimise the load on the target and be able to increase the load on the middle-ware whenever that is required. We also said that we want to have one single point of connection to each target system. Another interesting issue, which probably has more to do with performance than scalability, is throughput. Having a single connection between our middle-ware and the target means that we are going to have one object that takes care of many client’s requests, and therefore this single object must be able to make concurrent transactional calls serialisable – but only if the calls have some common points. That is, two independent concurrent/simultaneous transactional calls should be handled in parallel by the single object since there is no risk for inconsistency (by independent calls we mean two calls that do not access the same modifiable data).

For example, if two clients want to update two different alarms on the same target at the same time, then the single plug-in object (AlarmConsumer) that communicates with the target should be able to receive the two calls concurrently. Thus AlarmConsumer would be able to process the second client call and call the target before the result of the first call arrives from the target. Even though the plug-in runs both calls on the same CPU, meaning that the total process time for handling both calls is the same as if they were handled serially, the concurrency should improve the overall performance of the middle-ware. The first call might take much longer to complete, or it might get rolled back after the result is sent by the target, and then we gain time if we already have handled the second call as well.

But concurrency in throughput can be achieved only if

1. the plug-in can be accessed by concurrent threads and
2. there is no shared component between the clients and the plug-in.

We shall explain these two throughput conditions:

**Condition 1:** this means that the plug-in can not be a Session bean, since the same instance of a Session bean can not be accessed by more than one client. The plug-in could be an Entity bean having the value TRANSACTION_READ_UNCOMMITTED as its
Transaction Isolation Level. But we will not be able to test the behaviour of such a plug-in since in Weblogic a bean can not be a CORBA object – and our plug-in AlarmConsumer must be a CORBA object. To satisfy condition 1 we use a CORBA object that is not a bean.

**Condition 2:** what we mean is that the communication between the clients and the single plug-in should not go via a single component. In designs 1 and 2 there is such a single component, and it is an Entity bean. If the bean has received a client call in order to update the alarm A1, then no other client’s request to update A1 should be processed until the result of the first call is received by the single bean (sent by the target). But if another client (or the same client) wants to update another alarm A2, then the single Entity bean should handle this request and let it through. There is one way to make the Entity bean have this behaviour: declare the bean as TRANSACTION_READ_UNCOMMITTED. This speeds up throughput, but the bean’s business logic methods must explicitly handle concurrent client calls that aim to update alarms.

We have used the isolation level TRANSACTION_SERIALIZABLE on the single Entity bean in all designs. This makes the bean receive client calls one at a time, regardless of what the client wants to do. Of course, the performance of the system is decreased by having only serial request handling at the Entity bean. Furthermore, if there are 20 clients queuing to call the bean, and if a spontaneous alarm occurs at that time, the alarm will be received by the bean after it has handled the other 20 requests. And since the bean receives a client call only when its previous invoked method has returned and the transaction committed or rolled-back, there is no need for explicit checks to handle concurrent calls in the business logic methods of the bean. Nevertheless we have implemented these explicit checks in order to make it easier to change the isolation level of the bean and test the alternative which requires checks for concurrent client calls.

There is another thing we could do to improve the performance of the single Entity bean of design 1 and 2: we could use the transaction attribute TX_NOT_SUPPORTED on the bean in order to avoid the unnecessary writes to the database. With this mode the bean runs without a transaction, and therefore the method ejbStore() will not be called – thereby no writing to the database. But we have not bothered to improve designs 1 and 2 that much, because we don’t like the idea of making all
communication go via a persistent component in the first place, as the communication is slowed down by all database access in that component.

Designs 3 and 4 avoid using a single shared component as channel for all communication and use it only to establish/cancel connection to the plug-in objects of the target systems. Thus both conditions (1 & 2) mentioned above are satisfied in designs 3 and 4. In design 3, the plug-in takes care of the explicit checks to handle concurrent calls, whereas in design 4 there is no need for such checks.

**Alarm Conversion**

The plug-in objects convert the target-specific alarms (sent by the target) to generic alarm objects. The inner components of the middle-ware and also the clients know only about generic alarms. In all designs we have one object in which we keep a list, or several lists, of generic alarms. These alarms correspond to the active alarms on each specific target. Each generic alarm is given a unique primary key that works also as the primary key in the corresponding database table. When a client refers to an alarm when calling a bean, it provides the primary key of the alarm. Eventually the primary key is used to get the generic alarm from the list in the object which keeps the list(s), and if the target should be called with the specific alarm data as parameter, the plug-in converts the generic alarm to a specific one, and therefore the plug-in keeps a list of specific alarms. There is a 1-1 relationship between the generic active alarms in the system and the target-specific alarms kept in the plug-in (and on the target).

So please note that the conversion from generic to specific is actually not a process of converting, but is about finding the specific alarm in the plug-in. Also observe that the lists of active generic alarms are used to check the current state of the alarm object, rather than performing a database lookup each time an alarm is referred to.
3.1 Design 1

In this design we let all communication go through an Entity bean.

As you can see in Figure 3.9 there is a Session bean between the new Entity bean and the alarm consumer C. The Session bean must be stateful, since we want it to keep track of its own alarmConsumer objects and the active alarms at corresponding target systems. There are two reasons for not having a direct communication between the Entity bean and C.

The first reason has to do with common rules of object oriented programming. Assume that we have five kinds of target systems, and there are five instances of each kind, giving us 25 target objects. When we use a Session bean, the Entity bean just needs to communicate with five Session beans, or even better, if the Session beans can be modified and extended easily, the Entity bean will just have one Session bean to worry about.

The second reason has to do with flexibility and performance. We might want to have our consumer on another machine; that is, C will be a remote object. Remote communication with Entity beans should be avoided, and Session beans wrapping the Entity beans should be used instead. This increases performance, since remote access to an Entity bean is costly as it is connected to a database. You may wonder why we let the clients access the Entity bean directly. After all, the clients are most certainly remote objects. Well, that's a good point, and we will use Session beans between the clients and the Entity bean. But in order to measure the difference in performance, we
stick to the current design for the time being, and we introduce the changes further down.

3.1.1 GateBean
This Entity bean does not provide the system any meaningful persistence properties, and the only thing it does is to help us have a single component that can be used by more than one object. The bean knows what target systems the system is connected to, and what clients have subscriptions on alarms from each target. GateBean has been implemented both as bean managed and container managed.

When alarms arrive from a target object (via AlarmConsumer and AlarmPlug2Bean), GateBean saves them in the underlying database. The saves and updates of alarms can be done either by directly accessing the database using SQL instructions, or by using Entity beans (AlarmBean) corresponding to rows in the table. In the bean managed version of GateBean both of these alternatives have been implemented, while the container managed GateBean only uses AlarmBean for creation and update of alarms.

3.1.1.1 Creation
The single instance of GateBean is 'found' by all clients - not created, since it will have been created once before the delivered system is started. As a client asks for subscription from a target system T4, GateBean looks in its list of target systems, and if there is already a subscription on T4, the client's reference will be added to the list of those clients that have subscribed on T4. Otherwise GateBean tells the Session bean AlarmPlug2Bean that it wants to subscribe on T4.

3.1.1.2 Transactions
The transaction mode of this bean must be TX_REQUIRES_NEW. We want the bean methods to run in different transaction contexts when invoked by the clients and the Session bean AlarmPlug2Bean. When AlarmPlug2Bean receives a spontaneous alarm, one of two things happen:

1) AlarmPlug2Bean has an open transaction (opened by GateBean) which is used for the method handling the alarm and sending it to GateBean. If GateBean has TX_SUPPORTS or TX_REQUIRED, then it must be reentrant in order to be able to handle the alarm within the same transaction context that is being used for acknowledge/clear on an alarm. Obviously we don't want GateBean
to handle spontaneous alarms in the same transaction context that is used to update other alarms. Thus we must have TX_REQUIRES_NEW.

2) `AlarmPlug2Bean` has no transaction in effect when a spontaneous alarm arrives. The alarm is sent to `GateBean` which must handle it in a new transaction context. Thus we could have either TX_REQUIRED or TX_REQUIRES_NEW.

The two cases above agree upon only one mode: TX_REQUIRES_NEW, and the bean will not be reentrant. Each client call on `GateBean` will run in a new context, which is what we want, since a client may try to acknowledge two different alarms. There is also another issue related to this transaction mode, and it is important in our application:

If `GateBean` has an open transaction Tx1 for a client's acknowledge on an alarm, and `AlarmPlug2Bean` sends a spontaneous alarm to `GateBean` (within some context opened by `GateBean`), then Tx1 is temporarily suspended, and a new context, Tx2, is used in `GateBean`. When `GateBean` has done everything to handle the new alarm and Tx2 commits, Tx1 is resumed. Now the important thing is that the new alarm will remain in the database even if the client's acknowledge fails and Tx1 is rolled back.

Alarms arriving from `AlarmPlug2Bean` are saved in a database. As mentioned earlier, two kinds of alarm arrive: spontaneous alarms and changed alarms.

When we perform a transaction on `GateBean` that will lead to change in an alarm's state on the target system, `GateBean` will be expecting arrival of the changed alarm. In this design a changed alarm - or more precisely, an alarm that has been changed due to a transaction in our system - will arrive at `GateBean` twice. It arrives as the return value of the method that performs the transaction on `AlarmPlug2Bean`, and it also arrives when `AlarmPlug2Bean` makes a callback to send the alarm to `GateBean`.

Alarms that have already been returned by `AlarmPlug2Bean` (to `GateBean`) are already saved so the question is whether we should save them again. We want to let through other clients that want to acknowledge the same alarm that was just
acknowledged. If *AlarmPlug2Bean* would not return the alarm A1 that is the result of an acknowledgement on A, and would instead just send A1 to *GateBean* by calling it, then A1 might arrive at *GateBean* after twenty other new alarms (not results of any acknowledgements). Meanwhile, a client's repeated requests for acknowledging the same alarm are being rejected by *GateBean*.

This issue must be examined more closely. The above described scenario should actually not worry us in our application, because no client should be allowed to acknowledge the alarm A before it has seen that A has become A1. It seems like it would be a good idea to save the alarm A1 only when it is sent to *GateBean* via its method `handleAlarm`. A1 is sent to the clients when `handleAlarm` is invoked on *GateBean*, because *GateBean* must return a value indicating the success/failure of the client's transaction. But in that case, if a client wants to look at today's alarms in our database, it will get to see A despite the fact that our beans know that the database at that moment does not reflect the system's state. We end in an inconsistency. Thus we must save the alarm when it is returned from an acknowledge invocation on *AlarmPlug2Bean*.

But if we don't save A1 a second time (when it is sent to *GateBean*), how do we know that it really is a copy of the alarm that we saved before? The A1 that *GateBean* receives in its `handleAlarm` could be the result of some updates on the target system without our system's interference. So wouldn't it be better to save the alarms twice (first when acknowledge returns the new alarm A1, and second when A1 is sent to *GateBean* as any other alarm)? The answer is yes, and this is one of the changes that must be done in the current code. Still the following scenario is not impossible (see [Figure 3.10](#)):

We have two clients named c1 and c2, an active alarm A that is stored in the database, the Entity bean *GateBean* that handles the database, and the Session bean *AlarmPlug2Bean* that contacts the target object alarmHandler. This means that we do not have an *AlarmConsumer*, which has been omitted from this example just to keep things simpler. There is also someone called X who is operating on *AlarmHandler*, but not via our system. Note that calls from *AlarmHandler* to *AlarmPlug2Bean* go directly to the bean instance, which means that the bean instance can be called concurrently by *GateBean* as well.
c1 will see an unexpected result, since it got the value 1 (resultInt) which meant that the transaction succeeded. Although c1 might understand why this happened, and despite the fact that it will soon get A2, it should not end up in an unexpected state in any case. A2 is not a correct result, but A1 is correct. Furthermore, from now on there is a possibility that our system will fall behind one step in all transactions on A1,A2,... This is illustrated below:
**Time** | **Event**
--- | ---
8 | **alarmHandler calls:** plugBean.handleAlarm(A2);
9 | **plugBean calls:** gateBean.handleAlarm(A2);
10- | **gateBean receives A2,** saves it and sends it to c1 & c2. Thus c2 receives A2 at t11
10+ | **c2 calls:** resultInt2 = gateBean.acknowledge(A1); and its call is suspended until t11 (gateBean is already called by plugBean)
11 | **gateBean calls:** resultAlarm2 = plugBean.acknowledge(A1); and c2 has now received A2
12 | **plugBean calls:** alarmHandler.acknowledge(A1); and then it falls asleep
13 | **alarmHandler calls:** plugBean.handleAlarm(A3);
14 | **plugBean wakes up and returns A3 (to gateBean)**
15 | **gateBean receives resultAlarm2 (A3), saves it and returns**
16 | **plugBean calls:** gateBean.handleAlarm(A3);
17 | **gateBean receives A3 due to plugBean's call, saves A3 (for the second time), and sends A3 to c1 and c2.**

Now this situation is almost the same as in t1-t8, but there are 3 important differences:

1) There is no acknowledgement from X this time, so we cannot blame X for making us fall behind.

2) There is no reason for gateBean to let c2’s call through (at t11) although c2 has the same information as the database (A2).

3) c2 received 1, as the result of its transaction on A1, after it received A2. Our system is stupid because instead of rolling back c2's transaction it commits, although A3 is not the expected result of the transaction.
Moral of the story: we must have a mechanism to check the result of a transaction. In \textit{AlarmPlug2Bean}, if we acknowledge alarm A, later we must be able to tell if the received alarm A1 is the result that we expected. If A1 is not the expected result, then \textit{AlarmPlug2Bean} must return 0, indication that the request for transaction is rejected. But the problem is that we have already called \textit{AlarmHandler} and thereby started an external transaction. Considering the target object's interface, the only thing we can do is to make another call to \textit{AlarmHandler} to erase the result of our transaction. Of course we should let the database contain all the alarms sent by \textit{AlarmHandler} - even the ones that were not expected. Besides, we may not always be able to determine whether the result of a transaction is correct, even if that check can be done for the acknowledgement operation.

### 3.1.1.3 Improvements

I suggest that the interface \textit{AlarmHandler} be changed, so that:

- The method \texttt{acknowledge} returns either the new alarm (the result of the transaction), or a value to indicate success or failure of the acknowledgement. Even if \texttt{acknowledge} will return the new alarm, it should still send it to all consumers (including us) using the \texttt{handleAlarm} method.

- If the method \texttt{acknowledge} is changed so that it also takes the current \texttt{acknowledgedBy} field (and any other field that will be affected by the operation) seen by the consumer, then all the above mentioned problems will be solved:

  In that case \textit{AlarmHandler} will perform the transaction only if the current \texttt{acknowledgedBy} field of the consumer is the same as inside \textit{AlarmHandler}. But if the arguments of \texttt{acknowledgedBy} remain the same, it would be nice for the consumer to be able to able to cancel \textit{AlarmHandler}'s \texttt{acknowledge} operation when \textit{AlarmHandler} returns an unexpected result, and then no updates occur in \textit{AlarmHandler}. Otherwise (if consumers cannot undo acknowledgements at \textit{AlarmHandler}), the consumer must make an erasing acknowledgement to undo its previous acknowledgement.
3.1.1.4 Persistence

The way we have organised our database makes the faulty steps vanish from storage. This has to do with our policy to use one and the same row in the database for an alarm A, and the resulting alarms A1, A2, ... due to transactions. Currently we have:

<table>
<thead>
<tr>
<th>Time</th>
<th>ack field in database</th>
<th>alarm Ai</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>&quot;&quot;</td>
<td>A</td>
</tr>
<tr>
<td>t6</td>
<td>&quot;X&quot;</td>
<td>A1</td>
</tr>
<tr>
<td>t8</td>
<td>&quot;X&quot;</td>
<td>A1</td>
</tr>
<tr>
<td>t10</td>
<td>&quot;X c1&quot;</td>
<td>A2</td>
</tr>
<tr>
<td>t15</td>
<td>&quot;X c1 c2&quot;</td>
<td>A3</td>
</tr>
<tr>
<td>t17</td>
<td>&quot;X c1 c2&quot;</td>
<td>A3</td>
</tr>
</tbody>
</table>

If AlarmPlug2Bean makes an erasing acknowledgement when it discovers that X has been operating on alarmHandler, but still sends all incoming alarms to GateBean, and if GateBean saves them all, then we get:

<table>
<thead>
<tr>
<th>Time</th>
<th>ack field in database</th>
<th>alarm Ai</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>&quot;&quot;</td>
<td>A</td>
</tr>
<tr>
<td>t6</td>
<td>&quot;X&quot;</td>
<td>A1</td>
</tr>
<tr>
<td>t8</td>
<td>&quot;X&quot;</td>
<td>A1</td>
</tr>
<tr>
<td>t10</td>
<td>&quot;X c1&quot;</td>
<td>A2</td>
</tr>
<tr>
<td></td>
<td>&quot;X&quot;</td>
<td>A2.1 = A1</td>
</tr>
<tr>
<td></td>
<td>&quot;X c2&quot;</td>
<td>A3</td>
</tr>
</tbody>
</table>

As you can see we have assumed that c1 does not want to acknowledge A1, but c2 will still acknowledge A1. Thus AlarmPlug2Bean could instead avoid sending A2 to GateBean (the faulty A3 will not be generated if A2 is erased) in the table above (see t10 and t14).

There are two implementations of GateBean. One uses container managed Entity beans, alarmBean, to represent each row of the database table, as illustrated below. The other version uses common JDBC connections and statements containing SQL to explicitly create and update rows in the database table.
3.1.2 AlarmPlug2Bean

The instance of this stateful Session bean is mentioned as *AlarmPlug2Bean*. This bean is created by *GateBean*. Since communication between these beans is asynchronous, they have to call each other and they have to keep references to each other. *AlarmPlug2Bean* can be client of *GateBean*. That is, *AlarmPlug2Bean* can look up and use *GateBean* regardless of how many other clients there are, or as an alternative, *GateBean* can give the reference of its own proxy object to *AlarmPlug2Bean*. Thus *GateBean* will be called both by *AlarmPlug2Bean* and by the remote client. *AlarmPlug2Bean* on the other hand, cannot be called by any other object than *GateBean*, simply because a Session bean cannot be used by more than one client.

Before we start to discuss solutions to this problem we would like to know whom else needs to use *AlarmPlug2Bean*. The answer depends on where we put our plug-in instance; if *AlarmPlug2Bean* is also the implementation of CORBA plug-in interface *AlarmConsumer*, then *AlarmHandler* would communicate with *AlarmPlug2Bean* in
order to send alarms to our system. But if the plug-in is an object created by AlarmPlug2Bean, then AlarmHandler will call the plug-in AlarmConsumer which in turn talks to AlarmPlug2Bean. In any case, the point is that the proxy object of AlarmPlug2Bean can not be called by AlarmHandler or by AlarmConsumer. Instead the bean instance will give its own reference to the target/plug-in, so that the alarms will be sent to the bean instance without interference of the proxy object.

If we choose the first approach, and let AlarmPlug2Bean be also the plug-in object AlarmConsumer, then we will be exposing the bean instance to an exterior object. Since the instance variables of stateful Session beans must be public (so that the container will be able to access the variables and retrieve/assign their values at passivation/activation time), the target object will be free to do almost anything with the bean instance’s state. Thus this approach has major disadvantages in terms of security, safety and consistency, as shown in Figure 3.11.

In our design we have chosen the second approach: AlarmPlug2Bean creates an instance of AlarmConsumer for each target object. The target, AlarmHandler communicates with the CORBA object AlarmConsumer (using IIOP) which in turn calls the bean instance as alarms arrive from the target, as shown in Figure 3.12:

![Diagram](image_url)  
**Figure 3.12** A bad solution for the undesirable scenario of Figure 3.11
Still, the problem of shutting out the container from this communication remains. If there would be a demand on the container's awareness of and interference with the communication between `alarmConsumer` and `AlarmPlug2Bean`, then we must try other alternatives. These other alternatives are our other designs.

For each target object of type `AlarmHandler`, `GateBean` creates an instance of `AlarmPlug2Bean`. The transaction mode of `AlarmPlug2Bean` must be set with respect to the mode on `GateBean`, which is TX_REQUIRES_NEW. So `GateBean` will open new contexts for spontaneous alarms. But for each changed alarm returned by `AlarmConsumer`, a transaction has already been started by the container (for `GateBean`), and the same context is being used by `AlarmPlug2Bean`. Thus an alarm returned by `AlarmConsumer` to `AlarmPlug2Bean` to `GateBean` must be handled in the once opened context. This means that the mode of `AlarmPlug2Bean` must be TX_SUPPORTS (but since this mode does not work in Weblogic, we take TX_REQUIRED instead), as illustrated in Figure 3.13:

---

**Figure 3.13**  AlarmPlug2Bean has the transaction mode TX_REQUIRED
3.2 Design 2

Figure 3.14 Design 2: clients subscribe and acknowledge via a Session bean, and new/changed alarms are sent to client via another Session bean. An Entity bean is the single point of connection towards the target.

Figure 3.15 Design 2 does not break against EJB rules regarding bean access. All spontaneous alarms (all invocations) go via TraderBean’s proxy.
Here is the idea behind Design 2 (see Figure 3.14): use an Entity bean, TraderBean, in order to make the system scalable, i.e. to have one single connection with the target system. Let AlarmConsumer lookup and find the Entity bean via its proxy object, or even better: let TraderBean provide its own proxy object’s reference to AlarmConsumer. Session beans called ForwardBean are used for communication between clients and TraderBean. ForwardBean can be either stateful or stateless, depending on the client’s needs. If the client is going to be more of an observer who just receives alarms and rarely performs transactions, then ForwardBean should be stateless. But if the client is going to perform transactions continuously, then a stateful ForwardBean is more appropriate.

The client creates a Session bean, ForwardBean, gives its own reference to the bean so that the reference can be given to TraderBean. Then ForwardBean looks up TraderBean, delivers the message, and receives a return value (the list of currently active alarms). Finally ForwardBean returns the list to the client. Now TraderBean can send alarms to the client via BackBean, asking it to deliver the alarm to those clients that have subscriptions on the specific target system.

3.2.1 ForwardBean

First we use stateless Session beans here to keep the number of bean instances low compared to the number of clients. In order to make this possible, it is required that the clients will not be using the bean instances continuously. Even if the clients take pauses between their calls to the beans, the number of bean instances will be large if all the clients make calls simultaneously. But in our application none of these cases are likely to occur. As an alternative, we also provide a stateful Session bean. This keeps information about the targets that the client has subscribed upon.

3.2.1.1 Creation

The clients create and use instances of this bean to make requests to the target system. Assume that our client wants to subscribe on some target system, and therefore creates a new instance of the stateless ForwardBean. As we mentioned before, instances of stateless Session beans can be used by different clients, though not at the same time. Let's say that there has been some client using an instance of ForwardBean before our client, and let's add that the previous client is done. Then there is probably some instance of ForwardBean pooled by the container. This pooled instance is the one that was created and used by the previous client (if there was only
one client before). If the timeout values of all pooled instances have expired, then there will be no pooled instance for us to use, and a new instance will be created due to our client's creation request.

3.2.1.2 Transactions
The client of ForwardBean is the remote client that has been created by the user; it is our system’s real client. This client does not open and manage any transactions, so the container must start a transaction before the client’s calls are delegated to the bean. Thus the transaction mode for ForwardBean must be TX_REQUIRED. But this requires a discussion, since you may wonder why we don't use TX_REQUIRES_NEW.

Assume that we would like to let the clients handle their own transactions. Clients starting and managing their own transactions could actually be useful if the client wants to undo a transaction. The client might decide that its previously started transaction was a mistake, or that it resulted in an unwanted state at the target system, and therefore the transaction is rolled back by the client. Such a mechanism would follow the guidelines given by the Alarm IRP. Now if ForwardBean would have the mode TX_REQUIRES_NEW, then the bean would be given a new transaction opened by the container, and the client's transaction would be suspended until the bean method returned. Furthermore, if the new transaction of the bean would commit, it
would remain committed even if the client later rolls back its transaction, and this is not what we want. Thus *ForwardBean* is given the mode `TX_REQUIRED`, so that it works both now and in the future if/when we have a client managing its own transactions, as shown in *Figure 3.16*.

We know that transactions are supposed have ACID properties. Some problems concerning the properties of consistency and atomicity were discussed briefly in design 1. Those problems exist also in this design, and the suggested solutions apply here too. Just as a reminder, the problems disappear by replacing the asynchronous communication between *AlarmConsumer* and *alarmHandler* with a synchronous method invocation, so that the changed alarm that is the result of a transaction (acknowledge) is returned by *AlarmHandler*. As it is now, *AlarmHandler* calls *AlarmConsumer* in order to send the changed alarm.

### 3.2.1.3 Persistence

Let's see if *ForwardBean* could be given the responsibility for updating changed alarms, which are those alarms that are returned by *TraderBean* as result of a transaction. But this would imply that *ForwardBean* must lookup the alarms' Entity beans from the database. These lookups have already been done in *TraderBean*, and doing so again affects the performance. Besides, as we said, instances of *ForwardBean* just invoke methods on *TraderBean*, which means that two client's transactional calls on the same alarm are just let through to *TraderBean*. Thus, update of changed alarms must occur in *TraderBean*.

### 3.2.2 TraderBean

This Entity bean is very similar to the Entity bean *GateBean* of design 1. It exists for the same reason: to have one single connection to each target system. Its persistence properties are not used, and it corresponds to some dummy data in a dummy table in the database. It contains almost the same information as *GateBean*: references to *alarmConsumer* instances (*GateBean* had references to Session bean instances) and a list of generic alarms corresponding to the active alarms of each target. But despite all these similarities, the reversed order of Session beans and Entity beans (see *Figure 3.17*) in this design provide improvements in security and transaction processing. In design 2 the container supports *TraderBean* with transaction processing when the
plug-in \textit{AlarmConsumer} calls the bean as spontaneous alarms arrive (see \textbf{Figure 3.15}), whereas \textit{Plug2Bean} in design 1 cannot be given transactional support when called by \textit{AlarmConsumer}. The same thing applies to security support from the container.

\textbf{Design 1}

\begin{center}
\begin{tikzpicture}
  \node [shape=circle,draw=black,fill=blue!20] (client) at (0,0) {client};
  \node [shape=circle,draw=black,fill=green!20] (gate) at (2,0) {Gate Bean};
  \node [shape=circle,draw=black,fill=yellow!20] (alarm) at (4,0) {AlarmPlug2 Bean};
  \node [shape=circle,draw=black,fill=red!20] (alarmc) at (6,0) {Alarm Consumer};
  \path [->] (client) edge (gate);
  \path [->] (gate) edge (alarm);
  \path [->] (alarm) edge (alarmc);
\end{tikzpicture}
\end{center}

\textbf{Design 2}

\begin{center}
\begin{tikzpicture}
  \node [shape=circle,draw=black,fill=blue!20] (client) at (0,0) {client};
  \node [shape=circle,draw=black,fill=green!20] (forward) at (2,0) {Forward Bean};
  \node [shape=circle,draw=black,fill=yellow!20] (trader) at (4,0) {Trader Bean};
  \node [shape=circle,draw=black,fill=red!20] (alarmc) at (6,0) {Alarm Consumer};
  \path [->] (client) edge (forward);
  \path [->] (forward) edge (trader);
  \path [->] (trader) edge (alarmc);
\end{tikzpicture}
\end{center}

\textbf{Figure 3.17} Design 1 vs. Design 2: the reversed order of the Entity beans and the \texttt{sessionBeans} in Design 2 remove some problems of Design 1 as is shown in \textbf{Figure 3.15}

\subsection{3.2.2.1 Creation}

We want the same instance of \textit{TraderBean} to be used by \textit{AlarmConsumer} and by all instances of \textit{ForwardBean}. Thus clients of \textit{TraderBean} must lookup - not create - the bean. To make this work, we simply create the Entity bean once before the system is released, so that the dummy data exists in the dummy table in the database. Furthermore, everyone that will lookup the bean must know what to look for, i.e. the clients must know the Id value corresponding to the primary key in the dummy table, and they must also know the name of the table (otherwise, this information must exist as a constant in \textit{ForwardBean}). In our case there are only two classes that need this information and therefore we have hard-coded this data. The fact that these values do not change supports our simple solution. But if this cannot be accepted, then we could let the client provide the necessary data to \textit{ForwardBean}.

In this design it is important that all clients of \textit{TraderBean} really access the same instance of the bean. This is analogous to the single instance of \textit{GateBean} in design 1. As we mentioned earlier, some vendors might create one instance of the Entity bean per client. This means that we cannot be sure that this design is going to make a portable system. To make it portable when it comes to \textit{TraderBean}, the solution is to have a start-up client that finds an instance of \textit{TraderBean} and gets a \textbf{handle} (a serialisable reference) to the bean. The handle could then be stored in the database, so that \textit{ForwardBean} can look it up either by using JDBC commands or by using another
Entity bean *StringBean*. *StringBean* would contain the single handle corresponding to the single *TraderBean* and *ForwardBean* would convert the handle back to a reference to *TraderBean*.

### 3.2.2.2 Transactions

If two instances of *ForwardBean* invoke a transactional method of *TraderBean* in order to update the same alarm on the target system, then the second invocation must explicitly lead to rollback, fail or suspension. That is, *TraderBean* must handle problems arising from concurrent transactional calls as part of its business logic code, and this is not a desirable situation.

The transaction mode of *TraderBean* can be TX_REQUIRED since its methods are invoked both by *ForwardBean*, which calls from an open transaction, and by *AlarmConsumer*, which has no open transaction at call time. When *ForwardBean* is the caller, *TraderBean* should run in the same context as *ForwardBean*, as is shown in Figure 3.18:

**Figure 3.18**: The transaction mode TX_REQUIRED on *TraderBean* is needed in order to make the bean play in the same transaction context as *ForwardBean*. 
In Figure 3.18 you can see that BackBean runs in the same transaction context as TraderBean. This happens if the mode of BackBean is TX_REQUIRED. When a spontaneous alarm arrives at AlarmConsumer, if TraderBean is not running in a transaction we get the situation shown in Figure 3.19:

![Figure 3.19](image)

**Figure 3.19** Considering the consequence of having TX_REQUIRED on TraderBean with regards to spontaneous alarms sent from AlarmConsumer

So what happens if TraderBean is running in a transaction (it has been called by ForwardBean in order to update the alarm on the target system) when a spontaneous alarm arrives at AlarmConsumer? Well, AlarmConsumer will try to call the busy TraderBean, but its call will be suspended until TraderBean’s current transaction is either committed or rolled back, and thereafter AlarmConsumer’s call will get through to the bean in the new transaction that is started by the container. Analogously, if TraderBean is running in a transaction due to arrival of a spontaneous alarm, then ForwardBean will not get through to TraderBean until the first transaction is terminated.

One weakness of this design is the fact that AlarmConsumer both returns changed alarms to TraderBean and also sends them to the bean by calling it. The improvements suggested further down discuss this issue. What we have now is shown in Figure 3.20. As you can see the call which sends the changed alarm from AlarmConsumer to TraderBean does not conflict with returning the same alarm:
### 3.2.2.3 Persistence

As mentioned in 3.2.1.2, the responsibility of creation and updating of alarms in the database must be given to *TraderBean*. When *TraderBean* receives a call from *ForwardBean* in order to perform a transaction on an alarm, it first looks for the alarm in its own copy of the list of active alarms. If the alarm is in the active list, *TraderBean* calls the corresponding plug-in object, i.e. the *AlarmConsumer* instance that communicates with the alarm’s source. If the alarm’s update succeeds at the target, *AlarmConsumer* returns a new alarm as result. This new alarm has (or should have) the same primary key as the alarm that we wanted to update. With this primary key, *TraderBean* looks up and then updates the alarm in the database.

Changed alarms, i.e. those returned from *AlarmConsumer*, are stored before the result of the transaction is returned to *ForwardBean*. Also spontaneous alarms are stored before being sent to the clients via *BackBean*, but it should be pointed out that this task could instead be done by *BackBean*.

### 3.2.3 BackBean

This stateless Session bean is used for delivery of alarms to the clients. Spontaneous alarms and changed alarms are both sent to a given list of clients that is provided by *TraderBean*. If the spontaneous alarms are not stored in the database by *TraderBean*, then *BackBean* must do that. The transaction mode of *BackBean* should be

![Diagram](image-url)
TX_SUPPORTS, but since this mode does not work in Weblogic, TX_REQUIRED can be used instead.

3.2.4 Improvements

In this design ForwardBean is specified to return a value (e.g. an integer) denoting the success or failure of a transaction on alarms. Thus when a client gets to know that its transaction succeeded, it does not receive the changed alarm from ForwardBean. Instead BackBean will call the client. Let's say that client c1 has started a transaction on alarm A1 by calling ForwardBean (see Figure 3.21):

1) Eventually AlarmConsumer returns the changed alarm A2 to TraderBean.
2) TraderBean saves A2 in the database.
3) TraderBean returns A2 to ForwardBean.
4) TraderBean gets called by AlarmConsumer and thereby receives A2 (again), without knowing that it is the same alarm that was returned to ForwardBean.
5) TraderBean calls BackBean and gives it the alarm A2.
6) BackBean saves A2 in the database.
7) ForwardBean returns a value denoting success to the client C1.
8) BackBean sends A2 to the clients.

Figure 3.21  Detailed description of the problems mentioned in Figure 3.20. Chnaged alarms get stored twice in AlarmBean.
We save A2 twice. Can we avoid doing so? In design 1 we had a similar problem, and we had good reasons for saving alarms twice in GateBean. But the situation is not quite the same here. We could actually avoid saving A2 the second time, i.e. in BackBean. This means that there must be a method in BackBean that skips step 6 and just delivers the alarm to the clients. Thus TraderBean must be able to detect that the alarms in steps 1 and 4 are actually the same, i.e. A2 - or even better, let AlarmConsumer detect that. If this detection is generally not possible, then the only solution is to use synchronous communication between AlarmConsumer and AlarmHandler, as suggested before. In any case we would get **Figure 3.22**: 

**Figure 3.22**  The result of the improvements to solve the problems of **Figure 3.21**

But even if the communication between AlarmConsumer and AlarmHandler remains asynchronous, AlarmConsumer could still solve our problem concerning alarm storage, since in realistic terms of our application, AlarmConsumer can determine whether a received alarm is the result of one of its previous transactional calls to the target. In other words, AlarmConsumer can tell whether an alarm is the expected result of its previous acknowledge/clear call the target. So if AlarmConsumer only returns changed alarms to TraderBean, and if it only sends spontaneous alarms to the bean (by calling it), then TraderBean will automatically store each alarm only once.

In addition to the inefficiency caused by mandatory database access as all communication goes via TraderBean, this design suffers from another misbehaviour that is even more unwanted: only one transaction at a time can be performed via TraderBean. This means that only one alarm at a time can be acknowledged.
Sequence Diagram for Design 2: client subscribes on targets

Client
- GetTargetIORs()
- getInitialContext()
- lookup ForwardHome
- create ForwardBean
  - subscribe( targetIORs, this )

Forward Bean
- lookup ForwardHome
- find the singleton Entity bean
  - subscribe( targetIORs, client )

For those targets that were already subscribed on, send their active alarms (kept in TraderBean) to the new client

Trader Bean
- subscribe( this )

Find existing alarmConsumers of targetIORs
- Create non-existing alarmConsumers
- Add the new client to the list of subscribers of each created/found alarmConsumer

Alarm Consumer 1
- Convert string ior1 to CORBA ref AlarmHandler
- subscribe( ior1 ) if not created earlier

Alarm Handler 1

Name Service Forward Home Forward Remote Forward Bean Trader Bean Alarm Consumer 1 Alarm Handler 1
**Sequence Diagram for Design 2:** spontaneous alarm arrives from target

**Client**
- Alarm Home Remote

**Back Bean Remote**
- Trader Bean
- Alarm Consumer 1
- Alarm Handler 1

- `handleAlarm(1A)`
- `Convert 1A to generic GA`
- `handleAlarm(GA)`
- `getSubscribers(clients) of GA.target`
- `deliver(GA, clients)`

- `findByPrimaryKey(GA.id)`

- Stored alarm GA was found, so GA must be updated: `update(GA)`

- Alarm GA was not found. GA must be created: `create(GA.data)`

- `handleAlarm(GA)`
Sequence Diagram for Design 2: client acknowledges an alarm

Client

ack(GA.id, userId, myState)

check if GA is being acked by other clients

Check state of GA with clientState

ack(GA, currentState, userId)

Convert 1A' to generic GA' and then return GA' to TraderBean

findByPrimaryKey( GA'.id )

update( GA'.data )

updates its own list of activeAlarms' states

handleAlarm( GA' )

findByPrimaryKey( GA'.id )

update( GA'.data )

Get subscribers (clients) of GA'.target

Deliver GA' to all other subscribing clients

This is done via BackBean just as in the diagram on the previous page
3.3 Design 3

The actors in design 3 are the same as those in design 2; we have clients, *ForwardBean*, *TraderBean*, *BackBean*, *AlarmBean*, *AlarmConsumer* and *AlarmHandler*. But in this design the aim is to avoid the usage of the Entity bean *TraderBean* for all communication between clients and their target systems. Actually the only object that will use *TraderBean* is *ForwardBean*, which looks up the bean from the dummy table. Then *ForwardBean* asks for references to those instances of *AlarmConsumer* that communicate with the target systems that the client asks for. In fact, we do not need *TraderBean*, so let's illustrate an alternative design first (see Figure 3.23):

In the picture above, the client first wants a subscription on the target object *AlarmHandler*, and therefore it creates an instance of *ForwardBean*. If there is already an instance of *AlarmConsumer* that communicates with the specified target, then *ForwardBean* must find that instance. Otherwise *ForwardBean* must create a new *AlarmConsumer*. How does *ForwardBean* get this information? A simple way to do this is to let *ForwardBean* look in a database, or contact a name server in order to find out whether there is already a connection to the target system. If there is no connection (a proper instance of *AlarmConsumer*), then *ForwardBean* establishes a
new connection and updates the database/name-server. **Figure 3.24** illustrates the upper part of design 3 (the lower part is the same as in the alternative design shown in **Figure 3.23**):

![Diagram](image-url)

**Figure 3.24** Parts of Design 3. The missing components are BackBean and AlarmBean, and they are identical with the ones shown in **Figure 3.23**

We will describe *ForwardBean* and *AlarmConsumer* below. *BackBean* is similar to the one we used in design 2, except that it provides methods for delivery of alarms without updating the database. This new feature was suggested as an improvement in design 2, and it is used by *AlarmConsumer* when a changed alarm must be delivered to the clients, in which case the update of the database is performed in *ForwardBean* - not in *BackBean*.

### 3.3.1 ForwardBean

In design 3 *ForwardBean* gets its information from the single instance of *TraderBean*, which is responsible for creating instances of *AlarmConsumer*. Thus, when *ForwardBean* contacts *TraderBean* and asks for the object that communicates with a specific target, *TraderBean* creates an instance of *AlarmConsumer*, or finds the already existing instance, and provides its reference to *ForwardBean*. Since *AlarmConsumer* is a CORBA object, its reference is provided as an IOR string, so *ForwardBean* must convert the string to an object reference in order to call *AlarmConsumer*. The client receives the IOR strings for those instances of *AlarmConsumer* that communicate with those target objects that the client has asked
for. The client provides these IOR strings to *ForwardBean* each time the bean's methods are called in order to communicate with an instance of *AlarmConsumer*.

Is it possible for *TraderBean* to return a reference to *AlarmConsumer* to *ForwardBean*? That is, can we avoid using IOR strings? Well, the answer is partially dependent on the protocol used in communication between the two beans, which means that it is vendor dependent. As mentioned in 1.2.2.2, it is up to the vendor to decide whether to provide optimised method invocation:

- If this optimisation mechanism is not used, a call to *TraderBean* will go through the stub and skeleton objects of the bean. Assume that *AlarmConsumer* is not declared as remote. Then, when *TraderBean* wants to return a reference to *AlarmConsumer*, it must return a copy of it, and therefore *AlarmConsumer* must be serialisable. But we can not count on CORBA objects when it comes to serialisation (as a matter of fact they are not serializable), and *AlarmConsumer* is such an object. Besides, we do not want *ForwardBean* to carry a copy of a CORBA plug-in. So a reference must be returned instead, which implies that *AlarmConsumer* must be declared as a remote class.

- If the container short-circuits the RMI protocol between its internal beans, then *ForwardBean* will automatically receive a reference to *AlarmConsumer* returned by *TraderBean*.

It might seem that making *AlarmConsumer* remote would be a good solution. If *AlarmConsumer* would reside on a different machine than the server, then it must be remote, and *TraderBean* could return the reference to *ForwardBean*, and our problem would be solved. But we are looking to build a general solution that works even if *AlarmConsumer* is local to the server. After all, *AlarmConsumer* is a plug-in object and it must reside on the same machine as the server. Besides, if the optimisation mechanism is provided by the vendor, then the RMI overhead for communication with *AlarmConsumer* will only be a waste, and we do not want our design to be coupled with a specific vendor's product. Thus, we keep to our solution: we use IOR strings.
3.3.1.1 State of ForwardBean

Why is ForwardBean stateless? If we use a stateful bean instead, the client will not have to keep any IOR strings, and ForwardBean will not have to convert them each time before it calls AlarmConsumer. Therefore we have also another version of design 3, in which ForwardBean is stateful. The stateful bean gets the IOR strings from TraderBean, converts them to object references and keeps those references in order to use them when the client makes calls in the future. So why do we have stateless beans at all? This version of design 3 (stateless ForwardBean) has been implemented for two reasons. The first reason is that we can compare the performance of stateful and stateless Session beans. The second reason has to do with the application and the needs and behaviour of the clients: if some clients just want to subscribe on alarms without doing any transactions on alarms (or only a few transactions), then stateless Session beans are better suited. They can be shared by different clients and they will not waste any time (or little time due to the few transactions) on converting IOR strings.

The comparison between the two versions showed that the performances of the stateful and stateless beans were equal for our kind of transactions. The transactional method used in the test did acknowledgements on alarms.

3.3.1.2 Transactions

As in design 2, ForwardBean has the transaction mode TX_REQUIRED. Since the clients of ForwardBean do not manage their own transactions, the container will start a new transaction for each client request. Actually we want to use the mode TX_SUPPORTS for the bean, but as we mentioned before, this mode does not work in Weblogic. Note that TX_SUPPORTS would work only if all clients started their own transactions before calling the bean, whereas TX_REQUIRED works fine with both kinds of client.

Thus the container starts a transaction before methods of ForwardBean are invoked, and the same transaction context will be used in TraderBean (for subscription) and AlarmBean (to view and update alarms).
3.3.2 AlarmConsumer

In design 3 AlarmConsumer is the component that is responsible for explicit handling of parallel transactions on the same alarm. This means that if it has an ongoing transaction that updates an alarm on the target object, then it must reject all other transactional requests (on the same alarm) from ForwardBean. AlarmConsumer has a list containing a copy of the active alarms of the target system. ForwardBean makes a call in order to perform a transaction, i.e. update a specific alarm. So AlarmConsumer checks in its active-list to see whether the alarm really is known to be active, and if so, it checks if there is any ongoing transaction for that alarm. If there is no other transaction, the alarm is marked for transaction so that no other transaction on the same alarm will be allowed to proceed.

AlarmConsumer also keeps a list of the clients that have asked for subscription on alarms from its target object. This helps AlarmConsumer to use BackBean in different ways for delivery of spontaneous and changed alarms, respectively. When ForwardBean calls AlarmConsumer to perform a transaction, it provides the callee with the client's reference. So when AlarmConsumer receives the result of the transaction, i.e. an alarm, it asks BackBean to pass the alarm to all the clients except the one who started the transaction. Thus BackBean will not update the database. The update occurs in ForwardBean after it receives the changed alarm from AlarmConsumer. Thereby we have solved the problem of double updates that exists in designs 1 and 2. On the other hand, spontaneous alarms will be stored in the database by BackBean before they are sent to the clients.

One noteworthy point of design 3 is the way the responsibilities are divided between AlarmConsumer and ForwardBean. ForwardBean gets the alarm from the database to check whether the client's information is up to date, i.e. whether the client's copy of the alarm has the same state as the alarm stored in the database. But it does not check if the alarm is active, since it does not have a list of the active alarms. For the stateless ForwardBean having such a list is impossible, while the stateful version could contain the active-list. But this would force the client to notify ForwardBean about changes in the active-list as spontaneous alarms arrive. An alternative solution would be to let ForwardBean read the active/inactive state of the alarm from the database, i.e. whether the alarm has been cleared. But this would lead to overkill, since AlarmConsumer must still do the same check as it always has the freshest copy of the active-list.
3.3.3 BackBean

This is a stateless Session bean, and it is very similar to the bean with the same name in design 2, except for one difference: in design 3 BackBean does not save alarms that also have been or will be saved in ForwardBean. That is, when AlarmConsumer receives an alarm that is the result of a transaction, the alarm is delivered to the clients via BackBean, and then it is returned to ForwardBean for database update.

AlarmConsumer is the client of BackBean, which means that the bean must have a transaction context (created by the container) when spontaneous or changed alarms arrive (remember that AlarmConsumer does not manage transactions). Thus the mode of BackBean must be TX_REQUIRED, as shown in Figure 3.25:

![Figure 3.25](image)

**Figure 3.25** BackBean has transaction mode TX_REQUIRED, since AlarmConsumer does not manage transactions

The task of storing spontaneous alarms could be handled by AlarmConsumer instead of BackBean. This gives the system the same general behaviour as if BackBean would save the spontaneous alarms. Though, there are two benefits in this change: if there are no clients for the target that just sent the spontaneous alarm, then AlarmConsumer does not need to call BackBean at all, and it will just save the alarm in the database. The second benefit is that the process of storing a spontaneous alarm will not be coupled with the process of notifying the clients about the alarm. Due to this
modification, if something goes wrong while BackBean is delivering the alarm to 500 clients, or if that process takes time, then this will not affect the transaction context in which the alarm has already been stored successfully. And BackBean can do without a transaction context; its mode will then be TX_NOT_SUPPORTED, as is shown in **Figure 3.26**:

**Figure 3.26**  BackBean could do fine without a transaction context since AlarmBean will start a new transaction anyway

If we do not introduce the modification mentioned above, and let BackBean store the spontaneous alarms with the mode TX_NOT_SUPPORTED as before, then we should consider the mode TX_REQUIRES_NEW for AlarmBean. The purpose of this would be to have spontaneous alarms successfully stored, even if the transaction of BackBean gets rolled back after storing the alarm. For example if something goes wrong during BackBean’s conversation with the clients, or if its caller, TraderBean, gets in trouble and dies or rolls back the transaction in which BackBean is running, then it would make us feel safer if the spontaneous alarm had been stored by our system. Therefore we should use the mode TX_REQUIRES_NEW on AlarmBean, as shown in **Figure 3.27**:
This has a complicated backside too: *ForwardBean* or its caller, a remote client that manages its own transactions, will no longer be able to undo a database update. That is, if *ForwardBean* or its client decides to rollback a transaction so that the updates in the database and on the target fail, then someone must explicitly call *AlarmBean* in order to update it with its old state. This is not as bad as it might sound, since someone must explicitly undo the update on the target too, and we want our database to reflect the state of the target as well as the steps that have lead to its current state.
Sequence Diagram for Design 3: Client subscribes on targets

Client

GetTargetIORs()
getPropertyContext()
lookup ForwardHome
create ForwardBean
subscribe(targetIORs, this)

Find/create targets(targetList)

ForwardBean

lookup ForwardHome
find/create targets(targetList)

Forward
Remote
Bean

AlarmConsumer1

AlarmHandler

Forward
Home

Forward
Remote

Alarm
Consumer1

Alarm
Handler1

Trader
Home

Trader
Remote

Trader
Bean

Find the singleton Entity bean
find/create targets(targetList)

Trader
Bean
Convert 1A to generic GA

handleAlarm(GA)

Get subscribers (clients) of GA.target

deliver(GA, clients)

handleAlarm(1A)

findByPrimaryKey(GA.id)

Stored alarm GA was found, so GA must be updated: update(GA)

Alarm GA was not found. GA must be created: create(GA.data)

handleAlarm(GA)
Sequence Diagram for Design 3: client acknowledges an alarm

Client

- `ack(GA.id, userId, myState)`

- `findByPrimaryKey(GA.id)`

  - Check `GA.state` with `clientState`

  - `ack(GA.id, userId, currentState)`

  - Convert `1A'` to generic `GA'` and then return `GA'` to ForwardBean

  - Get list of clients subscribing on the target alarms except the acknowledging client

  - `notify(clientList, GA')`

- `handleAlarm(GA')`

- `1A' = ack(1A, currentState, userId)`

- `GA'` will be returned to the acknowledging client

- `findByPrimaryKey(GA.id)`

  - Check `GA.state` with `clientState`

  - `ack(GA.id, userId, currentState)`

  - Convert `1A'` to generic `GA'` and then return `GA'` to ForwardBean

  - Get list of clients subscribing on the target alarms except the acknowledging client

  - `notify(clientList, GA')`

- `handleAlarm(GA')`

- `1A' = ack(1A, currentState, userId)`

- `GA'` will be returned to the acknowledging client
3.4 Design 4

This design has exactly the same components as design 3, but there are important differences in functionality of some of the same components. In design 3, before we introduced the Entity bean *TraderBean*, we presented our view of a design that could do without that dummy Entity bean. In analogy with design 3, let's look at an alternative to design 4 (see Figure 3.28). In this alternative, we have omitted the Entity bean *TraderBean*, whose role is only to create or find plug-in objects.

![Diagram](image)

**Figure 3.28 An alternative instead of Design 4**

Thus we want to show that even design 4 really has no need for the dummy Entity bean *TraderBean*. And as we explained in design 3, in order to find a reference to an existing instance of *AlarmConsumer*, some database table could be looked up by *ForwardBean*. But we will use *TraderBean* for this purpose, and the bean's only task is to find or create an instance of *AlarmConsumer* for each target system. **Figure 3.24**, which shows the upper part of design 3, also fits into design 4.

The difference between designs 3 and 4 is the role of *AlarmBean*. In design 3 *ForwardBean* first viewed the stored alarm, i.e. *AlarmBean*, in order to verify whether the client is up to date with respect to the alarm's state. And then *ForwardBean* called *AlarmConsumer*, which continued the transaction and returned the result to
ForwardBean, and finally ForwardBean called AlarmBean to update the state of the stored alarm. Before looking into design 4, let's see what we wanted to change in design 3.

The major disadvantage of design 3 was that AlarmConsumer had to explicitly handle concurrent calls from ForwardBean to assure that only one transactional call per alarm would be allowed through to the target. A solution might seem to be given by letting ForwardBean mark the stored alarm for transaction, but this would mean that someone still had to explicitly lock and unlock the AlarmBean instance. In design 4 we let each alarm take care of its own transactions. So if ForwardBean wants to update the alarm A1 it just calls a business logic method of the instance of AlarmBean corresponding to A1, and the alarm will take appropriate actions to update itself. What AlarmBean does is to take the IOR string given by ForwardBean and convert it into a reference to an interface implemented by AlarmConsumer. Then the appropriate method of AlarmConsumer is called, and its return value is used by AlarmBean to update its own persistent fields. This way we do not need to worry about concurrent calls on alarms. The container guarantees that only one client at a time will have access to the bean. If we set the transaction attribute to TX_SERIALIZABLE (not to be confused with transaction modes), then we can be sure that the target will not get a transactional request on an alarm that has already been requested.

The transactional modes of the beans in design 4 are the same as in design 3 (see Figure 3.29):

Figure 3.29  The transaction contexts of the beans
As in design 3, spontaneous alarms are stored by *BackBean*. So we could introduce the modification concerning storage in this design too. That is, we let *AlarmConsumer* store spontaneous alarms, so that the only task of *BackBean* will be delivery of alarms to the clients, and *BackBean* is given the mode TX_NOT_SUPPORTED, as shown in **Figure 3.30**:  

![Figure 3.30](image)  

**Figure 3.30**  
The transaction mode of *BackBean* can be TX_NOT_SUPPORTED

And analogous to design 3, we could regret the modification introduced in the above picture, and let *BackBean* store spontaneous alarms while *AlarmBean* runs with the mode TX_REQUIRES_NEW, as shown in **Figure 3.31**:  

![Figure 3.31](image)  

**Figure 3.31**  
The transaction mode of *AlarmBean* is TX_REQUIRES_NEW
We can see that for changed alarms BackBean does not contact any entities other than the remote clients. The bean is stateless and it does not perform any direct database updates either. So let’s not waste the container’s resources on providing the bean transactional support. Regardless of whether BackBean stores spontaneous alarms, it would be better to give it the mode TX_NOT_SUPPORTED.

It is possible to involve AlarmConsumer in the transaction contexts of the beans. This plug-in is a CORBA object, and in some servers you can let your beans interact with a CORBA object. For example, in Weblogic we would use Weblogic Enterprise for this purpose. But in some other products you can let the container support such a CORBA object with transactions, and in fact you could have your plug-in declared as a bean. For example in Inprise Application Server all beans are CORBA objects. So we could think of two plug-ins, and then we could let these CORBA objects share their transaction contexts with the target object, as shown in .

![Diagram](image)

**Figure 3.32** An imaginary design in which the CORBA plug-in object get transaction support from the container

We have two CORBA objects as plug-in. AlarmAckerBean is used to perform transactions stated by our system in order to update alarms on the target. The target has a reference to the other plug-in, AlarmConsumer, which receives only spontaneous alarms. This idea could work even if the plug-ins were not beans, but they must still be CORBA objects that can take part in the beans' transaction contexts.
Sequence Diagram for Design 4: client acknowledges an alarm

Client

1. Client sends `ack(GA.id, userId, myState)`
2. Forward Remote
3. Forward Bean
4. Back Remote
5. Back Bean
6. Alarm Home
7. Alarm Remote
8. Alarm Bean
9. Alarm Consumer 1
10. Alarm Handler 1

- **findByPrimaryKey(GA.id)**
- **ack(GA.id, userId, currentState)**

Check **this.state** with clientState

- **ack(this.id, this.state, userId)**

**1A' = ack(1A, currentState, userId)**

Convert **1A'** to generic **GA**

- Get list of clients subscribing on target alarms except the acknowledging client
- **notify(clientList, GA')**

**GA'** will be returned to the alarm Entity bean

- **update this.state with GA'.state**
- **handleAlarm(GA')**
3.5 Comparison of the Designs

**Design 1**  
1. Breaks access rule: plug-in CORBA object could access bean instance directly, and thereby the calls from the plug-in to the Session bean instance were hidden from the container.

2. Singleton Entity bean critical. Its persistence is not used, but it decreases scalability sufficiently.

3. Concurrent client calls’ management policy is handled explicitly in business logic methods’ implementations in the beans. The reason for this was that we want to provide the ability of throughput increase by using other transaction attributes than `TRANSACTION_SERIALIZABLE`. But since clients access the singleton Entity bean serially, the code for management of concurrent client calls can be removed.

4. Very slow due to serial client access to the singleton Entity bean that forces all communication to be slowed down by unnecessary database connections.

5. Two database accesses for changed alarms (results of acknowledge/clear). The database is updated twice as the Entity bean AlarmBean is updated twice: once when the new alarm is returned synchronously, and once when it is sent asynchronously.

**Design 2**  
1. Still slow, but faster and much more scalable. Clients access the system concurrently.

2. Only one database access for changed alarms.

3. Management of concurrent client calls (when requesting for updates of alarms on the same target) is still done in business logic methods. But as in design 1, this can be avoided if the transaction attribute remains `TRANSACTION_SERIALIZABLE`. 
**Design 3**  (++++)

1. Still explicit management of concurrent client calls.

2. Very fast and highly scalable (clients can use stateless or stateful Session beans).

3. The singleton Entity bean is really not needed. It is used to keep track of the existing plug-in objects (and thereby the target connections), and it is used when the client asks for subscription. But we could easily remove it and let Session beans access the database to get information about existing target connections and their corresponding plug-in IOR strings.

4. Only one database access for updating changed alarms.

5. Should not have portability problems.

**Design 4**  (+++)

1. Management of concurrent client calls (when trying to update alarms of the same target) is done by the container, not in business logic methods. This is really not an improvement compared to the previous three designs, since as we said, the code for management of concurrent calls could be removed from all other designs as well.

2. Very fast and highly scalable as in design 3.

3. The singleton Entity bean is really not needed (see design 3).

4. Only one database access for updating changed alarms.

5. **Has portability problems.** If we insert the code for management of concurrent calls, then the portability problems will be removed.

Although design 4 is interesting and well-behaved (since the persistent components actively take part in transactions that result in their own updates), it does not make a portable EJB based system. The reason is the fact that vendors might let several clients of an Entity bean have access to two different bean instances. Then, although the two bean instances are logically identical, if the two clients call the two beans...
simultaneously in order to update the Entity bean, the two beans will make one call each to the plug-in *AlarmConsumer*. Although the plug-in can check for concurrent transactional calls from different instances of the same Entity bean, the concurrent calls will not be detected before the back-end (plug-in) of our middle-ware. But having explicit concurrency checks in the plug-in means that the system becomes portable, which is very important (see Figure 3.33 and Figure 3.34).

**Figure 3.33**  Design 4 deployed on Weblogic

**Figure 3.34**  Design 4 - if deployed on Inprise Application Server
3.6 Design Afterthoughts

In 1.2.5.2 where we introduced bean-managed Entity beans (BMEB), I wrote that months after I had done the implementations and the tests, I realised that non-persistent BMEB could be used as connector components. Such a bean could very well have been used in our designs: it would be the component to be called both by clients (or by Session beans called by clients) and by the CORBA plug-in AlarmConsumer. In Design 1 this new component would replace the Session bean AlarmPlug2Bean (and thereby Design 1 would no longer be faulty) as well as the container-managed Entity bean GateBean. In Design 2 our new component would replace the container-managed (and thereby always persistent) Entity bean TraderBean. Let’s see what Design 1 would look like, if we introduced a non-persistent BMEB (see Figure 3.35):

Here are some thoughts about Design 1.1 (we name the BMEB NewBean):

- Since NewBean is an Entity bean, it cannot have transaction mode TX_BEAN_MANAGED.

- If NewBean is reentrant, its transaction mode cannot be TX_REQUIRES_NEW, since the combination if re-entrancy and that mode gives us a deadlock.

- A non-reentrant transaction-enabled NewBean would decrease performance enormously, since the bean would be able to receive spontaneous alarms only when an ongoing transaction (due to acknowledge on some alarm) is done.
Having the mode TX_MANDATORY on NewBean would imply that the clients must be transaction-enabled (which does not sound good) or that they call Session beans that in turn access NewBean. This clientSessionBean reminds us of Design 2, as shown in Figure 3.36:

**Figure 3.36** Design 1.1 compared to Design 2: the similarity is that clients can only access ForwardBean, which in turn accesses NewBean

Thus if the reentrant NewBean has the mode TX_MANDATORY, ForwardBean will always call NewBean within a transaction. AlarmConsumer must have an open transaction when it sends spontaneous alarms to NewBean. This means that AlarmConsumer must be either transaction-enabled, or get transaction support from the container (for example, AlarmConsumer could be a bean on Inprise Application Server). If we want to avoid making AlarmConsumer explicitly deal with transaction primitives, we should choose another transaction mode for NewBean.

If AlarmConsumer is not transaction-enabled, using the mode TX_SUPPORTS on NewBean would make the bean process incoming spontaneous alarms (sent by AlarmConsumer) without a transaction. Actually we can live with that specific problem, since we can let BackBean and AlarmBean run with the mode TX_REQUIRES_NEW. However, there is a more fundamental problem related with the transaction mode of NewBean: while the bean is involved in a transaction, all calls to the bean are executed in the same transactional context. After all, that’s what re-entrancy is all about: to enable the Entity bean to be called while it is inside a
transaction. But we do not want incoming spontaneous alarms to be processed in the
same transaction that has been started due to a client’s acknowledge call, because
committing or rolling back the acknowledge transaction should not affect the updates
that take place as result of the arrival of the spontaneous alarm. Similarly, two
independent acknowledge calls should not be processed in the same transaction inside
NewBean.

Thus, NewBean should not be transaction-enabled, which means that its mode should
be TX_NOT_SUPPORTED. This will enable the bean to use transaction primitives such
as commit and rollback, which in turn means that NewBean will start new a transaction
for each new spontaneous alarm as well as for each new acknowledge call.

In version 2.0 of the EJB specification, there is a new kind of bean named
MessageDriveBean. This new component is probably the kind of component that we
could be using together with JMS (Java Message Services) that is part of J2EE, and
then we would probably not have to deal with transaction primitives inside the bean
code. Whether this new kind of bean would solve our problem or not is outside the
scope of this work.
4. IMPLEMENTATION
4.1 Implementation of Clients

When a plug-in subscribes on alarms from a target, it receives a subscription-id that is known by the target. This subscription-id can be used to cancel the subscription, so that the target no longer sends alarms to the plug-in. If the server crashes and the plug-in object is lost, the client started by a system manager should be able to contact the target and cancel the subscription of the plug-in. Therefore the subscription-id of each plug-in must be delivered to the clients. In some designs the client may expect arrival of alarms right after it has asked for subscription, whereas in other designs the client must first receive information about the plug-in objects as result of its request for subscription. In the latter case the client must call the server-side component again in order to start receiving alarms.

All designs have a client that subscribes on a given list of targets, and starts a thread that acknowledges a given alarm repeatedly. The result of the transaction, i.e. the state of the object returned by the acknowledge method, as well as the time it took, is presented on the screen. The client acknowledges the alarms until the corresponding database field of the alarm can take no more data (the acknowledge field is a string of maximum length 180).

Some designs also have a client that subscribes on some targets, and reloads the alarms from one target. Then it looks in the database for all the stored alarms that have a specific date value, and also all the alarms that have been generated since a specific date. The implementation of this functionality should generally not be done by using Entity beans, which means that when the client wants to view the alarms stored in the middle-ware’s database, we should not use finder methods on Entity beans. We want to avoid finding 1000 Entity beans for those 1000 beans that match the specified date. A case where it might be a good idea to use finder methods on Entity beans is if the client asks for a limited - and small – number of alarms that match the specified date or any other attribute. Another situation where we could use finder methods on Entity beans is when we are sure that the number of beans that will be found is small enough, and the client might need to ask for update or removal of some beans. Thus, instead of using finder methods, we can let a Session bean take the client’s request and perform database queries explicitly using JDBC and SQL statements. But we have used finder methods just to show how it is done.
4.2 Implementation of Plug-in in Designs 1 & 2

As we said before, when alarms are cleared or acknowledged (the same method is used in the `AlarmHandler` interface), the updated alarm is sent – not returned - by the target to the plug-in object `AlarmConsumer`. This was changed in designs 3 and 4, so that the new alarm would be returned to the caller. But in the two first designs and in their implementations the original target interface was used. In order to make the communication synchronous for the client and the beans involved in the transactions, a solution was implemented in the implementation class of `AlarmConsumer`. What we have done in that class is to make the thread that is executing inside the acknowledge method fall asleep right after it has called the target’s acknowledge method. Note that the plug-in is not a bean but a CORBA object, and therefore we are allowed to use Java thread primitives (we can not do that in beans). Later, when the target calls the plug-in and sends it a new alarm, the plug-in checks to see whether the received alarm has the same identity as any of the alarm identities that are temporarily saved in a list. The list contains those alarms’ identities for which the plug-in thread has called the target and then fallen asleep, and the call result has not yet been received by the plug-in. If a received alarm’s identity is found in the list, then the corresponding thread reference (stored along the alarm Id in the list) is used to wake up the thread. The thread will then find itself inside a catch statement, from which it can return the new alarm (to the caller of the plug-in’s acknowledge).

4.3 Tests and Benchmarks

All the designs described above (section 3), and most of the mentioned alternatives for transaction modes have been implemented, tested and used in the benchmarks. The benchmarks showed that the designs 3 and 4 were superior in performance compared to design 2. Design 1 was not really an alternative, since as we showed it did not comply with the rules outlined for EJB components, but it was tested as well and it had the worst performance. The reason behind the bad performance of designs 1 and 2 is of course their usage of the single container-managed Entity bean for all communication.

The benchmarks show us that in our system we can not see great difference between the different alternatives in designs 3 and 4, such as stateful or stateless Session beans used by the client. Although we do not use reentrant Entity beans, we still made some tests on reentrant Alarm beans in order to see if that affects performance.
One major factor that limited the testing field was the fact that all three parts of the system were located on a single Windows NT machine. This was partly due to the limited resources available, and partly due to my choice of putting the focus and emphasis on other parts (various design alternatives and examining the transactional and persistent properties of the implemented beans) rather than measuring and optimising performance. The ideal (actually realistic, as it often looks that way in real life) testing and benchmarking could probably be done on the physical architecture shown in:

![Diagram](image)

**Figure 4.1** The desired physical architecture for testing our system

Instead, we had a lot of processes containing multiple threads, all running on the same processor, which makes the measured execution time values less interesting. In the beginning, my aim was to drive the EJB server to its limits and to test it with intensive and heavy transactions. However, I soon I realised that much larger applications and real large-scale transaction processing was required to perform meaningful testing.

In Table 4.1 we have shown the combinations used for testing and benchmarking the implementation of design 3, in which *BackBean* and *AlarmBean* have the transaction mode TX_REQUIRES_NEW. There are five *AlarmHandler* targets, and each group of rows in the table shows that each client acknowledges alarms of exactly one target. Of course, the client that makes acknowledge on target’s alarms must have a subscription on the alarm. Each client may also subscribe on other targets as well.
### Benchmark for Design 3

<table>
<thead>
<tr>
<th>Bean</th>
<th>TX_MODE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td>REQUIRES_NEW</td>
<td>Stateless Session bean</td>
</tr>
<tr>
<td>Alarm</td>
<td>REQUIRES_NEW</td>
<td>Container-managed Entity bean</td>
</tr>
<tr>
<td></td>
<td>(REENTRANT)</td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>REQUIRED</td>
<td>Stateless Session bean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3-T5</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min, max)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>average first</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Client subscribes on target alarms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>S/A = Client subscribes on target alarms and acknowledges one alarm.</td>
</tr>
</tbody>
</table>

When the result of the acknowledge operation (a new alarm) is returned to the client, the client performs an acknowledge operation on it again. An alarm will be acknowledged totally 180 times.

The time column named “first” shows the time for the client’s first successful acknowledge call.

#### Table 4.1 Execution times for design 3

<table>
<thead>
<tr>
<th>Client</th>
<th>S</th>
<th>S/A</th>
<th>S/T3</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(min, max)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>average first</td>
</tr>
<tr>
<td>1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
<td>(100,1152)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>250 270</td>
</tr>
<tr>
<td>2</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
<td>(100,1081)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>218 280</td>
</tr>
<tr>
<td>3</td>
<td>S</td>
<td>S/A</td>
<td>S</td>
<td>(70, 731)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>114 201</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
<td>(140,1793)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>377 280</td>
</tr>
<tr>
<td>5</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
<td>(190, 901)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>371 521</td>
</tr>
<tr>
<td>6</td>
<td>S/A</td>
<td>S</td>
<td>S/T3</td>
<td>(100,5087)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>473 4066</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client</th>
<th>S</th>
<th>S/A</th>
<th>S/T3</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(min, max)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>average first</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Client subscribes on target alarms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S/A = Client subscribes on target alarms and acknowledges one alarm.</td>
</tr>
</tbody>
</table>

When the result of the acknowledge operation (a new alarm) is returned to the client, the client performs an acknowledge operation on it again. An alarm will be acknowledged totally 180 times.

The time column named “first” shows the time for the client’s first successful acknowledge call.
In row group number 3 of Table 4.1 we can see that there is only one client, and that it subscribes on five targets, acknowledging alarms on the target T2. The execution time values for performing the acknowledge request of the client are almost the same as the ones that are shown on row group 2, where there are only two targets. Thus, we did not see any increase in the execution time when we added targets and subscribed on them. The same thing can be said about the implementation benchmarks shown in Table 4.2, Table 4.3 and Table 4.4.

In row group number 1 of each table we have two clients and two targets, where the two clients acknowledge alarms of the same target T2. If we compare the time values of this row group with those of the row group 2 (where one of the clients is removed), we can see that the execution time for acknowledge is doubled for each client. If we also study the values of row group 5 in Table 4.1, we realise that each client’s observed execution time for acknowledge increases almost linearly with the number of clients that acknowledge alarms of the same target (see also row groups 6 and 8 in Table 4.3 and Table 4.4 respectively). These values probably do not reflect real usage of the alarm application, since all clients attempted to invoke acknowledge continuously, and all these threads, as well as the threads running in AlarmHandler stole time and resources from the same CPU.

In Table 4.1, Table 4.3 and Table 4.4, there are two row groups with 4 clients in each. In both rows, all clients acknowledge alarms of target T2, and the difference between the two row groups is that in one case all clients also subscribe on alarms of T1. It is interesting to see that the implementation of design 3 (with reentrant AlarmBean) results in almost equal time values for these two cases. It seems that the performance in design 3 is not very sensitive to the number of targets that the clients subscribe on (this could be interesting to test in a more realistic environment). On the contrary, the performance is somewhat slowed down in the implementation of design 4, when we let all clients subscribe also on T1.
### Benchmark for Design 3

<table>
<thead>
<tr>
<th>Bean</th>
<th>TX_MODE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td>REQUIRES_NEW</td>
<td>Stateless Session bean</td>
</tr>
<tr>
<td>Alarm</td>
<td>REQUIRES_NEW</td>
<td>Container-managed Entity bean</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(non-reentrant)</td>
</tr>
<tr>
<td>Forward</td>
<td>REQUIRED</td>
<td>Stateless Session bean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3-T5</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(min, max) average first</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>Client 2</td>
<td>-</td>
<td>-</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client 1</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>Client 2</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>Client 3</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>Client 4</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Client 1</td>
<td>S/A</td>
<td>S</td>
<td>S (T3)</td>
</tr>
<tr>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>S (T3)</td>
</tr>
<tr>
<td>Client 3</td>
<td>S</td>
<td>S</td>
<td>S/A(T3)</td>
</tr>
</tbody>
</table>

**Table 4.2** Execution times for design 3

If we compare most values of Table 4.2 with Table 4.2, we can see that performance increases by making the Entity bean *AlarmBean* reentrant. This is best seen where we have 3 or 4 clients. However, since reentrant beans can be troublesome and non-secure, and since they can lead to deadlock (especially with the transaction mode TX_REQUIRES_NEW), we should not make use of them just because they seem to be faster.
### Benchmark for Design 4

<table>
<thead>
<tr>
<th>Bean</th>
<th>TX_MODE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td>REQUIRES_NEW</td>
<td>Stateless Session bean</td>
</tr>
<tr>
<td>Alarm</td>
<td>REQUIRED</td>
<td>Container-managed Entity bean</td>
</tr>
<tr>
<td>Forward</td>
<td>REQUIRED</td>
<td>Stateless Session bean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3-T5</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(min, max)</td>
</tr>
<tr>
<td>1</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>-</td>
<td>-</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>Client 1</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 3</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 4</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 3</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 4</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Client 1</td>
<td>S/A</td>
<td>S</td>
<td>S (T3)</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>S (T3)</td>
</tr>
<tr>
<td></td>
<td>Client 3</td>
<td>S</td>
<td>S</td>
<td>S/A (T3)</td>
</tr>
</tbody>
</table>

We can see that the time values in Table 4.4 and Table 4.3 are almost the same. The only difference in these two tests of design 4 was the transaction mode of the Entity bean *AlarmBean*, but it seems that the difference was sufficient enough to make the first alternative slightly more efficient. That is, *AlarmBean* performs best with TRANSACTION_REQUIRED, and it is easy to understand why: the container has more job to do when *AlarmBean* (with the mode TRANSACTION_REQUIRES_NEW) always...
must be given a new transaction, whether it is called by *ForwardBean* or by *AlarmConsumer* (compare Figure 3.29 with Figure 3.31). It would be interesting to test a larger system in order to see how much performance is dependent on the transaction modes of components, and whether the required number of new transactions (created by the container) slows down the performance dramatically or just marginally.

**Benchmark for Design 4**

<table>
<thead>
<tr>
<th>Bean</th>
<th>TX_MODE</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Back</td>
<td>REQUIRES_NEW</td>
<td>Stateless Session bean</td>
</tr>
<tr>
<td>Alarm</td>
<td>REQUIRES_NEW</td>
<td>Container-managed Entity bean</td>
</tr>
<tr>
<td>Forward</td>
<td>REQUIRED</td>
<td>Stateless Session bean</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3-T5</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(min, max)</td>
</tr>
<tr>
<td>1</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>S</td>
</tr>
<tr>
<td>4</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>-</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>6</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>-</td>
<td>S</td>
</tr>
<tr>
<td>7</td>
<td>Client 1</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 3</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 4</td>
<td>-</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>Client 1</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 3</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Client 4</td>
<td>S</td>
<td>S/A</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>Client 1</td>
<td>S/A</td>
<td>S</td>
<td>S (T3)</td>
</tr>
<tr>
<td></td>
<td>Client 2</td>
<td>S</td>
<td>S/A</td>
<td>S (T3)</td>
</tr>
<tr>
<td></td>
<td>Client 3</td>
<td>S</td>
<td>S</td>
<td>S/A (T3)</td>
</tr>
</tbody>
</table>

**Table 4.4** Execution times for design 4
4.4 The Java Code

Since I started working with the EJB implementation examples of Weblogic in packages under `examples.ejb`, all my Java implementation code came to belong to those packages. Code belonging to a package is placed in a directory corresponding to the package name’s path.

examples.ejb.alarm.callback

This package contains a container managed Entity bean that has been used in order to test and understand components’ behaviour with different transaction modes, re-entrancy and transaction-enabled clients. A lesson I learned in working with this package was that if we use the transaction attribute `TRANSACTION_REQUIRES_NEW` on reentrant Entity beans, we certainly end up with a deadlock. The reason is the fact that EJB has only flat transactions – not nested. It should also be pointed out that in the EJB literature there is an advice that is given every now and then: try to not make your beans reentrant!

examples.ejb.alarm.simulator

In this package we have the IDL generated types and interfaces, i.e. the CORBA interfaces `AlarmHandler` and `AlarmConsumer` as well as the target-specific alarm type `AlarmInfoOut`. Since the original IDL file also contained some other interfaces that are not used in our work (and since I have not removed those interfaces), we have several Java files that are not used.

examples.ejb.alarm.common

This package contains several Java classes and interfaces that are used by several different designs’ implementation.

examples.ejb.alarm.storage

This package contains the container managed Entity bean `AlarmBean`, i.e. the persistent alarm component that is used in all designs. The directory ’storage’ contains two subdirectories. The subdirectory ’examples’ contains a sample of the code (for `AlarmBean`) that has been generated by Weblogic at compile time. The other subdirectory, ’storageBeanMan’ contains the `bean-managed` version of the Entity bean `AlarmBean`. 

148
Implementation of Design 1

examples.ejb.alarm.test
The code related to design 1 is in the ‘test’ directory, and the package contains
the code for implementation of design 1. Although the code was tested and
proved to be working, it has not been maintained since the work went on with
the other designs, and we still let the reader have the possibility of studying the
code.

examples.ejb.alarm.test
The singleton Entity bean of design 1 is in this package.

Implementation of Design 2

examples.ejb.design2.fix
The singleton Entity bean TraderBean of design 2 is in this package.

examples.ejb.design2.notify
The stateless Session bean BackBean of designs 2, 3 and 4 is in this package.

examples.ejb.design2.passiv
The stateless version of the Session bean ForwardBean of design 2 is in this
package.

examples.ejb.design2.usr
The stateful version of the Session bean ForwardBean of design 2 is in this
package.

Implementation of Design 3

examples.ejb.design3.connect
The singleton Entity bean TraderBean of designs 3 and 4 is in this package.

examples.ejb.design3.stateFul
The stateful version of the Session bean ForwardBean of design 3 is in this
package.

examples.ejb.design3.stateLess
The stateless version of the Session bean ForwardBean of design 3 is in this
package.

Implementation of Design 4

examples.ejb.design3.sessSL
The stateless version of the Session bean ForwardBean of design 3 is in this
package.

examples.ejb.design3.sessSF
The stateful version of the Session bean ForwardBean of design 3 is in this
package.
5. DISCUSSIONS

5.1 EJB in Element Management Applications?

The applications of Element Management (a unit in Ericsson’s Network Core Products CNCP) involve element management, application design, operations and maintenance, product and system development for driving and maintenance of network nodes, switches and other data communication entities. Much of this involves providing two things: 1) tools for customising, supervising and control of traffic channels and stations on one hand, and 2) getting information about the current state of the objective – in other words: alarms.

What we are trying to say is that we will have asynchronous communication, just as in the alarm application that we used. Another important thing about the applications of this organisation is that they must provide user-friendly interfaces for operators. The operator could either use non-comfortable ways and access its target via some primitive interface, or it could use some middle-ware that provides better functionality. And in a big organisation you probably expect the middle-wares to provide similar functionality and similar user interfaces although different kinds of targets are being used (of course, we mean the areas where the operator’s needs are similar).

So we do not want to invent the wheel in our unit, and we want to be able to reuse our business logic components (probably after some minor modification, i.e. customisation), and we are certainly going to have at least some middle-ware systems that must be improved and developed constantly. So far, the needs of UAB/I seem to be of the kind where EJB is intended to help. The question is whether the applications have enough need for transaction management, and whether the cost in terms of speed and memory usage really can be neglected compared to the benefits gained by EJB. This question can not be answered without knowledge in the specific case.

But there are several other benefits to gain if the system is complex and large enough, i.e. systems that are in need of transaction management, security management and persistence of state. And especially if you want your new EJB-based system to be able to play in the same game as your older CORBA system when it comes to transactions
and security, and if you have heavy transactions, then EJB could be something to consider for your applications.

One of the areas in which EJB servers based on specification 1.0 and 1.1 have not been the most appropriate choice, is when you have asynchronous communication, call-backs and need for singleton components. In EJB Specification 2.0, there is a new mechanism, which enables beans to send and receive messages asynchronously. This is done by integrating EJB with Java Message Service (JMS), and it means that some difficulties that we encountered in our designs (also when solutions were easily found) should vanish if we use some future EJB server based on specification 2.0. The problem – if it is one – with singletons accessed by several components might still remain, but EJB is intended to be able to coexist and co-operate with CORBA components, which means that we could use a CORBA singleton whenever our EJB-based middle-ware must contain a singleton.
5.2 Commercial Products

Weblogic [E6]

Weblogic is the name of the EJB server made by BEA Systems. Weblogic was the first application server that was released as a beta version based on EJB 1.0, and in April 2000 (probably also in July 2000) it was the only EJB product that had a beta release based on EJB 2.0. As we have said before, Weblogic was the server used in this Master’s Thesis work. The support provided even for the evaluation licence is fantastic, and they have been responding quickly. There are also theoretical problems as well as practical tips and examples on the Weblogic sights and in their FAQ pages.

The optional optimisation for short-circuiting RMI protocol between two beans in the same container is not provided by Weblogic, which decreases performance. CORBA clients can not use EJB in Weblogic, since there is neither an interface for using pure CORBA nor RMI-over-IIOP. Another weakness of Weblogic is that bean implementation classes have relatively much vendor-specific code. When I began to work with the evaluation versions of Weblogic (from September 1999 and on), there was not an integration with Java environment products, but later they provided Symantec’s Visual Cafe (I did not use Visual Cafe projects to build my code, but just used it as an editor). All my statements about Weblogic are based on the evaluation product based on EJB 1.0, up until January 2000.

Weblogic provides integration with Microsoft’s Component Object Model (COM).

Inprise Application Server [E7]

The current application server of Inprise has version 4.0, on which EJB 1.1 is supported. One interesting aspect of Inprise Application Server is that all EJB components are CORBA objects, which enables the beans to be called by non-Java CORBA clients using the RMI-over-IIOP protocol. A disadvantage of this product is the unusual mechanism (though allowed in the JEB specification) for implementation of the client access model for Entity beans. Each client gets its own instance of an Entity bean, which makes portability difficult (to and from the Inprise product) when other major products let clients access the same instance of an Entity bean serially.

Please see also the other sources about commercial products ([T1, T6]) listed in the bibliography.
5.3 EJB vs. COM+

When we look at the component-based server-side world of today, we see two competing techniques, two philosophies and two systems: EJB and COM+.

COM+ is the architecture and the product of Microsoft, and that means that it runs only on Windows. In COM+ there are no persistent components, and neither do they have stateful components. In the philosophy of Microsoft, a component is by definition stateless and non-persistent. So if we remove the stateful Session beans and the Entity beans of EJB, we get somewhat closer to COM+. [M1,M2]

EJB components are all written in Java, while COM+ components are language independent, meaning that they can be written in C++, Java or even Visual Basic. Although COM+ components are platform dependent, they can interact with other platforms and other technologies. [M1,E5]

Since studying COM+ has not been part of this Master’s Thesis assignment, instead of trying to speculate on whether the votes in favour of COM+ (and against EJB) are based on facts and deep analysis or not, and instead of making speculations about purely commercial interests (rather than scientific or technical), we advice the reader to study COM+ in order to compare the two technologies. However, it should be helpful to consider the following issues: (see [M1])

- EJB servers normally cost (a lot of) money, whereas COM+ comes with Windows. This means that the customer probably would make a long-term decision and choose either Windows (including COM+) or other platforms, in which case he has to pay additionally for an EJB server. However, some companies include an EJB server (and also other J2EE products) in packages together with other tools.

- The whole idea with persistent and stateful components is rejected by some people who favour Microsoft and COM+. Their argue that persistent or semi-persistent state makes it impossible to build scalable and efficient systems, since an instance of a stateful component belongs to only one client. Though, EJB has a passivation and activation mechanism for Session beans, which in fact increases performance and scalability.
5.4 General Discussions and Conclusions

We shall list some of the most important topics and domains that are part of or related to Enterprise JavaBeans:

- Portability
- Scalability
- Reliability
- Support for transaction management
- Support for persistence
- Security
- Support for distribution and concurrency
- Separation between system services and business logic
- Performance
- Reusable components

We will discuss each of these issues, and we will try to answer questions about whether EJB seems to fulfil its promises, and also what benefits or weaknesses it has.

Portability

This is one of the most important features that EJB is associated with. As we said before, the heart of J2EE and its whole realisation is dependent on whether EJB will become the great server-side component-based platform of the future. Now that the world has Java, and now that we know that the words “write once – run anywhere” were true, it remains to see whether this Java-based idea of J2EE and EJB will transport us from the world of OOP into the new era of component-based development on the server-side for middle-wares.

One thing that we must keep in mind is that gain in portability often means loss in performance and optimisation. Software with strong portability is usually either non-efficient (since it probably is not using underlying system properties in an optimised way), or it is simply not a typical advanced and complicated system. Perhaps what we are looking for might be reasonably efficient systems that will also be portable with reasonable effort. That is, we could start thinking about components and component suites that can be customised before usage, so that before customisation the components will be portable, and afterwards they will be efficient.
Anyway, in our alarm handling application we saw that even though we designed according to the rules outlined by EJB, and even though we did not seek to achieve optimisation by utilising any optional (allowed by the EJB specification) or strongly vendor-specific features of Weblogic, we could not count on portability for some of our designs – not even to some major EJB vendors. On the other hand, it might be realistic to aim at achieving portability among some widely used EJB servers. Sun Microsystems has its own EJB server named NetDynamics (see [T5]), which is meant to be used as a sort of verifier for or portability and correctness of an EJB-based implementation. However, our great concern is whether we really can rely on the evaluation result even if our code passes through NetDynamics and turns out to be correct with respect to J2EE criteria.

Thus, portability among different vendor’s products has not been achieved, and it should become one of the main goals of the Sun Microsystems, all other EJB vendors and the EJB community. It should also be pointed out that some vendors (such as Inprise) have managed to make programmers write pure EJB code, which means that the code will not contain any vendor-specific code, whereas other tools (such as Weblogic) force the bean implementation classes (and even clients) to include Weblogic-specific code. Note that porting from Inprise to Weblogic is generally not easier than the other way around, since the code must be changed in both cases.

**Scalability**

There are several ways to design and implement scalable client/server or middle-ware systems using EJB:

- Stateless Session bean instances can be used by multiple clients, which means that we often can serve many clients with few bean instances.
- Stateful Session beans can be passivated in order to free up space and other resources, so that other instances of the bean (or of other component types) can be created.
- Lifetime and passivation-time of Session beans can be trigged to improve scalability

EJB has succeeded with scalability, and the different kinds of Session beans can be used in different combinations to achieve scalability in the specific application.
Reliability
One of the ideas behind using a container and a server that provide services to the EJB components is that the bean implementers (as well as the client-side programmers) let expert vendors take care of issues such as databases, transaction services, concurrency and security. Software developers are given advice about concentrating on their business logic and rely on the expert knowledge that has been used in the specific EJB tool. This sounds like a good idea, and it can result in powerful and reliable tools when the experts do a good job. But it is also possible that some vendors that are not so good in some areas (for example an expert of transaction servers who may be not so good at databases or security) still try to do the job by themselves.

Since EJB is a specification, and not a product, many different communities and major companies with different interest areas might be trying to have impact on some mandatory or optional features of the EJB specification. And even if each company do their best to make EJB better for everyone, some major companies probably will do things differently since they come from different application areas and have different approaches. This could very well have positive effects on the evolution of EJB, but it could also lead to less reliability in some products and less portability among them.

Support for Transaction Management
Choosing the right one among the six transaction modes that are defined in the EJB specification, and combining the mode with the right transaction isolation level, are the only two issues that designers using EJB have to be concerned about. The designer can also decide to let the person implementing the bean explicitly write transaction handling Java code.

Up until now, the underlying XA layers of database systems have not been able to support distributed transactions since they have not been designed for 2-phase commit. However, now that 2-phase commit has been introduced in JDBC, we should expect to be able to perform distributed transactions in EJB as well – the question is how long it will take before we get there.

In our design afterthoughts, we realised that when we have a component that works as a single point of connection towards a target in order to be called asynchronously by the target (get callbacks from the target), we cannot use the transaction services of EJB. If the component is already inside a transaction, it will handle the target’s
request in the same transaction context, which is not always desirable. There is no way to let component handle a request in a different transaction context when it is already taking part in a transaction. This is because EJB uses only flat – not nested – transactions.

**Support for Persistence**

There are two kinds of persistent components in EJB:

- **Container-managed Entity beans**: the mapping between component state and database state, and also transactions are handled by the container.
- **Bean-managed Entity beans**: transactions are handled by the container, whereas persistence must be supported explicitly in the bean implementation class using JDBC and SQL.

While a corresponding database table must always exist before a container-managed Entity bean is created, a bean-managed Entity bean (BMEB) can be created without any persistence, i.e. without a database. This special kind of BMEB can be used as a component that can handle callbacks, and thereby it can work as a connector (with some limitations) for asynchronous communication.

**Security**

Since the emphasis of this work was not on security, there is not much to be said on EJB’s benefits or weaknesses regarding security.

**Support for Distribution and Concurrency**

As we said, distributed transactions have so far not been supported, whereas concurrency, which is related to distribution, is very supported. Designing and implementing an EJB-based system should normally result in great support for concurrency – and that is without traditional concurrency concerns at the design stage, or concurrency code in the implementation of the EJB components; you get automatic support for synchronisation of concurrent threads.
Separation between System Services and Business Logic
Bean instances are never accessed directly by clients – instead, builder-tool-generated objects (proxies) implementing home and remote interfaces are used. The EJB server and the EJB container provide services to the proxies and also to the bean instances, and thereby, there is very clear separation between system services and business logic. Application developers can concentrate on business logic, and they only need to declare and implement business logic methods in the remote interfaces and the bean implementation classes respectively. The services supported by the container and the EJB server are: transaction management, database connections and persistence, security (authorisation and authentication), concurrency (thread synchronisation), RMI communication (with EJB you never write RMI code) and also some optimisation mechanisms to improve performance as well as scalability, such as instance management of pooling and the mechanism for passivation/activation.

Performance
Performance is a relative to the application area, and as we said before, it can usually be achieved by some loss in portability. In the EJB specification there are some optional mechanisms for increasing performance, such as short-circuiting RMI communication between two beans in the same container. In some e-business applications, and also in some element management applications, the respond time is not the main concern, and the reliability of the system is measured more in terms of correctness, consistence, security and scalability. But continues availability and respond time are very essential in most telecom areas and also in other systems where performance must fulfil some real-time requirements, and these are among the cases where EJB cannot be used.

Reusable Components
Reusability is one of the main benefits in OOP, and the idea is to keep and strength this feature with component-based development on the server-side. Although object are normally more reusable by being smaller than components, component developers should aim to build customisable components in order to increase their reusability. In many areas, different tools and applications would benefit from reusing components – and then they will not have to redo the job (services provided by EJB) that normally has to be done when objets are inserted to be reused in existing applications.
Bibliography

**EJB**


[E3] Vogel Andreas, Madhavan Rangarao: *Programming with EnterpriseJavaBeans, JTS and OTS*

This paper was received at a seminar held by BEA, where Edwards and Orfali were the lecturers.

[E5] Sun Microsystem’s official EJB web page  
http://java.sun.com/products/ejb/  
http://java.sun.com/products/ejb/docs.html  

[E6] Weblogic’s and BEA’s home pages  
These pages were used mostly before February 2000.  
http://www.beasys.com/  
http://www.weblogic.com/docs/classdocs/API_ejbdesign.html  
http://www.weblogic.com/docs/examples/ejb/Package-examples.ejb.html  
http://www.weblogic.com/docs/classdocs/index.html

[E7] Inprise Application Server (the FAQ page was used mostly)  
http://www.inprise.com/devsupport/appserver/faq/  
http://www.borland.com/appserver/

[E8] EJB Specifications  
http://www.javasoft.com/products/ejb/javdoc-2_0-1d2/  
http://www.javasoft.com/products/ejb/javado-1_1/  

[E9] EJB and RMI-over-IIOP  

**CORBA**

[C1] Vogel Andreas, Duddy Keith (1998):  

*Programming with Visibroker*, Wiley Computer Publishing, USA
Java, JDBC, RMI etc


[J2] Links from Sun Microsystem’s official Java page

[J3] J2EE and JDK 1.1.7
http://java.sun.com/products/

COM+

[M1] EJB vs. COM+. A debate between Ed Roman and Roger Sessions
http://www.objectwatch.com/Presentations_F.htm

[M2] Microsoft’s home page
http://www.microsoft.com

COMMRCIAL EJB PRODUCTS

[T1] Flashline’s comparison matrix for Application Servers
http://www.flashline.com/Components/appservermatrix.jsp

[T2] IBM’s Websphere Application Server

[T3] Inprise Application Server
See [E7]

[T4] BEA’s Weblogic
See [E6]

[T5] Sun Microsystem’s NetDynamics
http://www.netdynamics.com/

[T6] Links to EJB Products’ home pages