University of Oslo
Department of Informatics

Building real-time audio applications with component technology

Eivind Mork

Cand. Scient. Thesis

6th May 2005

Preface

This thesis is presented for the degree of Cand. Scient. at the Department of Informatics, University of Oslo. The context of this work is the QuA project at the Simula Research Laboratory (in partnership with the University of Tromsø and SINTEF).

At the end of this work, I realized that the thesis could have profited on moving some of the focus from the implementation work, and to a more thorough discussion about QoS issues. Time constraints prevented this shift of focus.

I would like to thank my supervisors, Richard Staehli and professor Frank Eliassen at the Simula Research Laboratory, for all their help and support during this work.

A special thanks to Heidi, my wife, who also helped with proofreading. You are always there!

Oslo, May 2005

Eivind Mork

A chapter overview

Contents

Chapter 1

Introduction

Contents

Component Based Software Engeneering (CBSE) is a common way of creating software, and there are now several different component middleware architectures and implementations available. This thesis investigates the benefits of creating real-time audio applications with components, and the limitations of current component technologies.

1.1 Background

1.1.1 Components

Component architectures are designed to make it possible to create distributed applications by composing it from reusable components. The components should function correctly regardless of the underlying hardware and software layers [1].

Szyperski among others, argues that CBSE can reduce the cost of software development [2]. The time of development can sometimes be reduced, the quality can be better and the maintenance easier.

1.1.2 Extra functional concerns

• quality of service (QoS)

In addition to the functional aspects of a component, there are extra functional aspects. These extra functional aspects are often called Quality of Service (QoS) and describe the other characteristics of a service than the functional, such as performance and security. The term QoS is both used when describing what a component can offer (in an extra functional aspect), and likewise which extra functional requirements a component requires from other components in order to fulfill its QoS offer.

• QSA

An application that tolerates some amount of error or imprecision, but cannot tolerate unbounded error, is referred to as a QoS Sensitive Application (QSA) [3]. A QSA should not need to specify physical requirements, but can instead specify logical (i.e. platform independent) QoS requirements. The hardware independence will simplify the design of QSAs that are supposed to be deployed in different hardware environments. The mapping from hardware independent to hardware dependent requirements will need a service within the platform.

The handling of the QoS, could either be left to the application itself, the component architecture, or both.

1.1.3 Real-time audio applications

Since the early nineties the interest in continuous media concerns has been considerable, and especially for video on demand [4]. This thesis investigates real-time audio as it is representative of the very large class of continuous media applications that can benefit from component technology, and yet, real-time audio is relatively simple to understand. Even though real-time audio applications are much less resource demanding than for example real-time video, many of the mechanisms and problems are the same.

There are many types of real-time audio applications available, such as Internet phones and conferences, radio broadcasting and applications with more complex data compositions (i.e. audio synchronized with real-time video). The applications could be quite complex, but many strategies and mechanisms are shared by these applications. For example, many of the applications use an audio recorder and an audio player, and most of them would use some kind of compression.

Some of the common mechanisms deal with QoS issues, and they have to be implemented either by the application itself, or by the underlying software or hardware layers. For instance, the audio packets sent through

the network have some delay constraints as the audio may not be delayed beyond a certain limit. Mechanisms for avoiding and/or handling packet dropping may also be present to avoid “gaps” in the audio play-out.

As many of the mechanisms in real-time audio applications are common, we believe that also this group of applications could benefit from being made using components.

1.2 Definition of problem area

Component Based Software Engeneering (CBSE) has successfully been used to implement business logic (the code that implements the functionality of an application [5]). It is believed that software in general could benefit from using CBSE [2]. The assumption of this thesis is that QSAs are no exception, even though modern component technology is not used for QSAs. This thesis looks into why this is so, and will suggest a list of issues that needs to be resolved in order to implement QSAs with this technology.

To identify the limitations of existing technology, the characteristics of the components would have to be known. The range of QSAs are huge, and to narrow down to a more manageable scope, this thesis looks at only one type of QSAs; real-time audio applications. There are two reasons for this choice: Real-time audio applications are relative simple to understand and create, and; they share interesting aspects with many QSAs such as real-time requirements of continuous media.

Just as there are many types of QSAs, there are also many existing component architectures. They share many of the same ideas, but they are also developed for different purposes and software and hardware architectures, and hence, they have many differences. There is more than one architecture that would be interesting to investigate in the setting of implementing QSAs. This thesis considers Enterprise Java Beans (EJB) - a technology within the Java 2 Enterprise Edition (J2EE) programming environment from Sun Microsystems[1] - as it has been here for some years, is widely used with success in many companies and government offices, and can be run on many different computer architectures (due to the nature of Java).

This thesis will answer the following questions:

- How can an Internet phone and similar real-time audio applications be built from components?

[1] `http://www.sun.com`

- What limitations of the EJB standard will cause problems implementing real-time audio applications?
- In what way can the EJB standard be changed or extended to eliminate the identified limitations of the current standard?

1.3 Research method

At the time of this work, there were several different EJB implementations available. We decided to perform the evaluation of just one EJB implementation, due to time limitations for the work. JBoss, an EJB implementation made by the JBoss Group[2], became the chosen implementation. This decision was based on the following reasons:

- We consider JBoss to be a well respected, reliable implementation. There are many examples of critical systems using JBoss, like the Dow Jones indexes, the German Parliament (election system) and the Norway Post. The use of JBoss has been increasing over time, and in 2002 it was downloaded more than 2 million times [6].
- JBoss is open source, which could be helpful in the evaluation process.

To identity requirements that real-time audio applications may have to JBoss, an identification of common components among audio applications must be done first. To be able to do that, the design of the applications will have to be known. Since there are an unlimited number of possible designs, the method is to inspect several existing applications to identify commonly used mechanisms and strategies, and to discuss their possible reuse.

The identified components will then be inspected to find their requirements. The requirements will be compared to the limitations of the EJB standard and JBoss to find out which components can be implemented in JBoss, and which can not. With the EJB architecture, the programmer has some limitations of what can be done within an enterprise bean. Two cases will therefore be investigated: The first case will look at possible implementations following the EJB 2.1 specification. Both the scenario where *all* components are made as enterprise beans, and the scenario where *some* components are made as enterprise beans will be discussed. In the second case there will be used non standard EJB features. Finally there will be discussed how the EJB standard can be extended to support this type of applications and what the consequences of this will be. This

[2] `http://www.jboss.org`

discussion is based on the requirements found in the investigation of the common components compared to the restrictions of the EJB standard, as well as the experience obtained during the implementation work.

The process of implementing fully functional Internet phone applications is made to help reveal the requirements that are not supported by the EJB standard: Some of the issues can possibly be overlooked in a theoretic discussion if the design of the application and the EJB standard are not investigated adequately. A working implementation can also contribute as a proof of concept. Even though it can not prove that the concept will work in any scenario, it will prove that it is working in the current setting. There are made two sets of applications: One does not implement any of the identified components as enterprise beans, and the other implements all the components as enterprise beans. The one without enterprise beans works as an reference application to form a basis for evaluation of the other applications using EJB.

The goal of this work is to learn about the common mechanisms in existing real-time audio applications and the requirements they have for a component framework. A second goal is to discuss in what way the EJB standard can be changed/extended to support the identified requirements and how the extensions will effect the standard.

1.4 The structure of this paper

This paper is organized into 11 chapters. After this introduction, related work is presented in chapter 2 before some background information about CBSE and real-time audio is presented in chapter 3 and 4.

Chapter 5 presents the identification of common components in real-time audio applications, and their requirements are discussed. Chapter 6 then presents information about the J2EE standard and the JBoss implementation. Finally in this chapter, there is a discussion of in which degree EJB supports the requirements identified in chapter 5.

Possible designs and implementations of an Internet phone application following the EJB standard is discussed in chapter 7. Chapter 8 presents a design and implementation of an application without the use of EJB. Designs and implementations using EJB and restricted functionality is presented in chapter 9. Both chapter 8 and 9 present some measurements of the performance of the applications.

In chapter 10 possible changes and extensions to the EJB standard are discussed. Finally, chapter 11 presents the conclusions and further work.

Chapter 2

Related work

Contents

This chapter will have a brief look at related work concerning support of real-time multimedia systems and adaptive applications in general, using component middleware platforms. The work presented in the first section is creating a completely new component middleware platform. The work in the two next sections are based on the Microsoft COM/DCOM and CORBA frameworks respectively, and are thus more similar to this thesis work since their platforms are made with changes and extensions to existent platforms. The project in the last section is extending the EJB framework with QoS support for RMI, and is thus quite related to this thesis work.

2.1 QuA

There are several research projects designing new component architectures that are made to better support QoS. One of them is the QoS-Aware Component Architecture (QuA) project at the Simula Research Laboratory [7]. QuA is an adaptive component platform that supports real-time and multimedia systems. It seeks to separate the application requirements and environment dependent implementation decisions, to better

exploit the adaptive components and to make reuse possible for different environments. This is made possible by letting the platform manage the deployment, configuration and dynamic adaptation of an application.

QuA uses *service planners* to discover possible implementations and selecting the best according to a quality specification. The service planner is a deployable component that is connected to dedicated hooks in the platform. An implementation plan can be divided recursively into finer grained implementation plans, and thus also the service planners can be called recursively to find the best sub-implementation plans that finally will form a complete implementation.

QuA is platform (i.e. OS and programming language) independent. Working prototypes are made in Java and Squeak (Smalltalk).

2.2 OpenORB 2

The OpenORB 2 component platform is somewhat older than QuA. Like the QuA project, it aims to offer better support for applications with special requirements that may also change in different domains [8]. Examples of such applications are safety-critical, embedded, and real-time systems. Traditional middleware platforms can not meet the requirements of such applications because of the limitations of the black-box philosophy where the components can not observe any state within the platform.

Open ORB 2's solution to this problem is to offer reflection where the components can use a meta-interface to observe the internal observation and structure of the middleware platform. The components can also perform changes to the platform through the meta-interface. The Open ORB 2 architecture itself consists of components as well. Instances of the platform can be made of a selection of components at build-time, and they can be reconfigured at run-time using reflection.

The Open ORB 2 component model is based on Microsoft's COM platform, and the implementation is based on the Open COM implementation.

2.3 TOAST

The Toolkit for Open Adaptive Streaming Technologies (TOAST) [9] is a CORBA-based multimedia middleware platform that aims to imple-

ment multimedia support in CORBA using plug-and-play mechanisms for components.

Components exchange data through flow- and stream interfaces. A flow interface can be either input or output. Stream interfaces are two or more flow interfaces bundled together to one single interface. They are meant to make connection establishing less complex. Flow and stream interfaces are defined using CORBA Interface Definition Language (IDL) which makes it language and platform independent. Distributed binding objects are used to bind objects over a network.

Adaptation is made possible through reflection: The bindings offer meta interfaces that makes it possible to inspect a component graph that models the TOAST components and their bindings. This graph can be altered to make changes in the application.

2.4 Adding QoS support for EJB

One project has extended the EJB standard with QoS support for RMI [10]. The current main standards provide support for e-commerce and general purpose business applications, and hence, they provide services like persistence, transactions, events and security. QoS support for real-time applications are not supported.

[10] have made a prototype with a new component type called a *QoS-Bean* to support QoS, although the project do not address multi-media streaming applications like real-time audio. The QoSBean supports business components with QoS IP requirements like jitter and latency. The QoSBean has new semantics and configuration attributes for the QoS, just like session beans and entity beans have attributes for e.g. persistence and security.

The introduction of a new type of bean also requires a new container that can handle resource reservations. Clients can request and negotiate with a component the number of times each second it will invoke each of the component's methods. The component will then negotiate the resource reservations with the container on its client's behalf. If the component needs to invoke methods of other components to serve its client's invocations, it must negotiate with these components as well.

Chapter 3

Component technology

Contents

This chapter gives a short introduction to components and Component Based Software Engeneering (CBSE)

3.1 What is a component?

There have been given many different definitions of a component. For this thesis it would be suitable to say that "(...) software components are executable units of independent production, acquisition and deployment that can be composed into a functioning system. To enable composition, a software component adheres to a particular component model and targets a particular component platform" [2]. Software made of such components, are thus called *component software*, and the act of making such software is called *component based software engineering*.

• component

• component-based software engineering

Szyperski lists three characteristic properties of a component [2]:

- It is a unit of independent deployment
- It is a unit of third-party composition
- It has no (externally) observable state

A component should not be dependent on other components in order to be deployed. It could be dependent on other components to *execute* and perform as described, but not to be deployed. In the deployment process it can never be partially deployed, that is, in the deployment perspective, a component is a single atomic unit.

The second characteristic property is its possibility to be part of a composition made by a third-party composer. Before a composer can use a component, he would need to know how to access it. This means that the component's interface must be well-defined and specify which methods and public variables the component does offer, and what the method's signatures are. In addition to the interface, the composer needs a broader understanding of what the component can do, in order to classify a component as usable or not. This is defined in a *contract*. The contract specifies what the clients need to provide, and what the component then can offer (as long as the client fulfills its part of the contract). The contract can cover functional and extra functional aspects (QoS guarantees). The composer should *not* need an insight to the component's implementation in order to use it. It should be of no interest as long as the component's interface and contract offer are well defined. By analyzing and comparing the properties of the component candidates, the composer can choose the best component for the current scenario.

- component interfaces
- contract

Finally, a component has no (externally) observable state. This means that a component can not be distinguished from copies of itself. Szyperski gives an example with a database and a database server [2]. The database server could be a made as a component. This component would be the same in any installation independent of the database(s) it serves. The database would be an instance. If the component should also contain the database as well as the database server, replacement and maintenance of the component would be very difficult.

3.2 Component frameworks, containers and servers

- component framework

A component framework can be described as the "glue" that ties components together. More accurately it is a set of interfaces and rules for how components that are plugged into the framework can interact [2]. The standardized rules and interfaces for interaction between components simplifies component development and assembly, and makes it easier for developers to understand the design of an application [11].

- component container

A container can in general be described as objects that contains other objects [2]. A component container is thus an object that contains in-

stances of components. Features of the components can be *container managed*. An example of this is life-cycle management where the container takes the responsibility for creating and destroying components when it finds it appropriate. Its clients do not have to worry whether a component is instantiated or not, as the container will take care of this when required.

• container management

A component container runs on a component server. The server can host one or more (possibly different) containers.

3.3 Benefits from using component technology

Szypersky [2] gives several reasons why the use of components is less expensive than production from scratch. Below, it is presented a list extracted from Szypersky's discussion. The arguments underlying these reasons are discussed in the following paragraphs.

1. Production from scratch is very expensive.
2. Except for the local areas of expertise, the implementation from scratch will likely be suboptimal.
3. In a world of rapidly changing business requirements, custom made software will often be too late to be productive before it is obsolete.
4. It is likely that components with different qualities will be available at different prices. You get what you pay for.
5. Using standard software the burden of maintenance, product evolution and interoperability is left to the vendor of the standard package.
6. With components, massive upgrade cycles will be put to an end.

1) Creating complex component from scratch, could be very expensive. If it is a common problem, there could be a market for selling components solving the problem, and thus suitable components could be available. The price would depend on factors like how big the market is, the complexity and quality of the component, and the price/availability of components from other companies. Even though a problem could be common, there is no guarantee for the component solution being simple. In some cases it could be quite the opposite, and a standard component should then be considered.

2) A component created and tested by experts, would also release more time to concentrate on local problems where the local expertise most

likely would be. To make sure that all parts are well made, each part should be created by experts in this field. It is very rare for a company or a project to have experts in all required fields, and thus, the parts where the expertise is missing could be of better quality if left to external professionals.

3) With the increased complexity of a problem, the time spent creating a component solving the problem would also increase. Delays in a project could be very expensive, and if a component is too late for the product to be successful, the result is fatal. A bought common-off-the-shelf (COTS) component which is known to work, will shorten the time from planning to finish, as the implementation is already done, and hence unexpected delays are less likely to occur.

4) Another unwelcome surprise to a project is a component that performs worse than expected. A component's performance could be difficult to predict before it is made and tested. Thus, critical parts of the application could therefore be a source for delays in a project if the quality of the component is inadequate. COTS components are tested components, and they could be advertised with known quality, requirements and price attributes. When a quality requirement is specified, the selection of a COTS component with a sufficient quality would give a solution with both a known price and a known performance.

5,6) The larger an application is, the greater is the amount of time likely to be spent on maintenance. Again, using a third party's components should leave more time for maintaining in-house components. Any upgrades should be a matter of independently upgrading components, not a matter of upgrading the whole system. It could be easier to find a bug if the changes to the system is smaller. If an error occurs after upgrading one single component, the maintainer can conclude that the upgraded component has something to do with the error in some way or another. A simultaneous upgrade of several components could make bug tracking more complex.

Chapter 4

Real-time Audio Streaming

Contents

This chapter will present some basic characteristics of real-time audio. It will also address common problems in real-time audio applications, with corresponding strategies for dealing with those problems.

4.1 What is real-time audio?

Before describing the *real-time* aspect of audio, the term *audio* itself needs to be defined. To define audio, one need to know what sound is:

- sound

Sound is rapid pressure variations in the air, and the magnitude of the pressure variations creates the sensation of *loudness*. Sound move through the air according to the rules of wave propagation, and therefore sound pressure variations are often called sound *waves*. The sound waves are analog as the air pressure is continuous [12].

- audio
- sampling

Electronic sound is called *audio* [12]. To transform the analog audio signal to digital audio, it have to be *sampled* . We define sampling as "the process of making a series of values equally spaced in time out of

a continuously varying analog signal. Each sample is the instantaneous value of the analog signal at the time of sampling" [12].

• audio quality

When the human ear is the destination of the sound that is played, audio quality is best defined as the subjective evaluation of the audio. Thus, the best way to determine the audio quality of audio clips, are subjective listening tests [13]. But even though the quality evaluation of the audio is subjective, it is safe to say that in general, the quality of the sound depends on attributes such as the sample frequency, the resolution of each sample, and the encoding schemes (if the audio data is compressed).

• audio stream

Internet phones transports audio data from the sender to the receiver via an audio *stream*. A stream is an entity representing an ordered, finite sequence of entities or values called elements of the stream [14]. Hence an audio stream is an entity representing an ordered finite sequence of audio samples.

• end-to-end delay

It is important that the end-to-end delay (the difference between the time an audio sample is recorded, and the time it is received and played back at the listener's computer) is low, since a long end-to-end delay can cause confusion in the conversation. This can be experienced when speaking in a satellite transmitted phone call which have longer delays than most regular phone calls. In a system where we have short delay constraints,

• real-time audio

we say we have a *real-time system*. Thus, real-time audio is audio in a setting where the end-to-end delay should be low.

With common QoS requirements such as short delays, low packet dropping and low jitter, a real-time audio application would be an example of a QSA.

4.2 End-to-end delay

The end-to-end delay is caused by the network, the operating system, the hardware, and the software itself. Figure 4.1 illustrates the delay from the recording of an audio clip on one computer (A) till the playout of the clip on an other computer (B).

a The start of the recording at computer A.

b The first audio clip is recorded and being sent to computer B

c Computer B starts playing the clip.

d The playing of the clip is done.

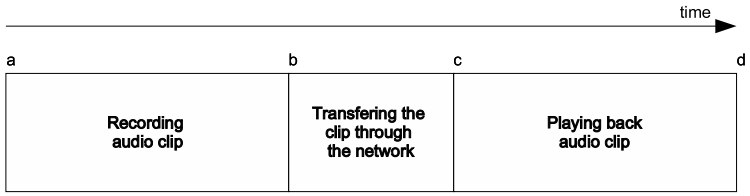


Figure 4.1: An illustration of the delays before playout

As we can see, what user A said at *a* (the beginning of the clip) will not be played back at computer B before *c*. The end-to-end-delay is thus defined as *c* minus *a*.

If the information flow is one-way only (i.e. a radio transmission), the end-to-end delay would not be very important. In a phone or conference scenario the case would be quite different: If the delay is too long, it conversations get difficult. The end-to-end delay should be kept under 400-600ms to avoid confusion and frustration from the users [15]. If the delay is larger, it is likely that both the users will eventually start talking in a silent spot on each side. After the delayed time, they will find themselves talking simultaneously, and both instantly stop before starting to talk again, making it possible to create the incident again.

4.3 Jitter

A phenomenon that is often discussed in a real-time and network environment, is *jitter*. Jitter is the variation of the delay caused by the network. A sudden increase in the delay could cause a silent gap in the stream. A sudden decrease in the delay could cause the packets to arrive too fast, forcing the application to discard the packets arriving too early.

• jitter

To deal with jitter, it is common to buffer the incoming audio stream for a short while before it is played back. If there occurs a gap in the stream before arrival, due to network unreliability, there would be some time to wait for the missing audio packet to arrive and fill the gap. If the packet is lost, a re-transmission of the lost packet could be required if the buffer is big enough for the re-transmitted packet to arrive in time. A programmer have to make a trade-off between dealing with delayed/lost packets by using a jitter buffer, and a big end-to-end delay, as the jitter-buffer increases the overall delay.

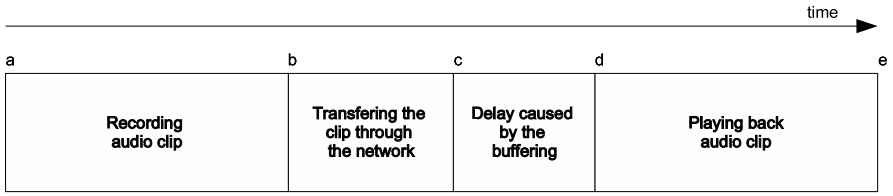


Figure 4.2: An illustration of the delays before playout, including the delay caused by buffering

Figure 4.2 shows the end-to-end delay including the delay caused by the buffering. The end-to-end-delay is here defined as d minus a.

4.4 Packet dropping

• packet dropping

An important challenge in a real-time audio application, is to avoid packet dropping. We define packet dropping as a packet being lost in the network transfer, or a packet that is received too late to be played back and thus discarded.

There are two different approaches to deal with packet dropping: One could either try to avoid the packet loss and jitter, or one could have a strategy to compensate for the problem. The former is handled by introducing a real-time operating system and resource reservation protocols that can guarantee for the packets to be processed within an agreed time [14]. The network would have to provide QoS guarantees (with guaranteed values for throughput, latency and packet drop). As the "phone call" could be routed through a heterogeneous network with many network providers which do not necessarily understand or respect the protocols, this could be a problem. In addition to that, people rarely have a real-time operating system.

The latter approach is the approach most applications use, as it does not require special network protocols, and could be used on the Internet as it is today. It gives no guarantees, but tries to compensate when problems occur. There are several ways to compensate for lost packets, and often more than one way is used in an application: One could use layered encoding (with which a lost packet only reduces the sound quality and not causing a short silence), or one could use forward error correction (where parity data is added to later packets to make it possible to recon-

struct the lost packets) [16]. Both these methods add extra data which is not always feasible. A third approach is to duplicate the former packet, playing it twice, which works satisfactory if the packet size is small and the loss rate is low [15]. A fourth and not so sophisticated approach is to just play white noise or silence in the place of the dropped packet [16]. Many applications also monitors the packet loss, and if the loss is too big it could change its strategy (possibly in agreement with the other user(s)) by changing the size of the jitter-buffer, or it could introduce, replace or reconfigure a compression algorithm.

4.5 Handling bandwidth limitations

The absence of an adequate bandwidth could be a problem for the application. If the bandwidth is insufficient, the packet drop would be great. 8000 kHz, 8-bit, mono, is by many considered to be of sufficient quality for conversations. For uncompressed audio this equals a bandwidth of 8000 bytes per second or 64000 bits per second (bps). Compression codecs will reduce the bandwidth requirements, and the compression ratio will depend on the codec. Speex[1], an open source codec, can reduce the bandwidth down to as low as 2000 bps (a 1:32 ratio) and up to 44000 bps, depending on the quality needed.

To avoid packet dropping, the bandwidth requirements should be adjusted to be within the limits of the bandwidth available. One way to adjust the bandwidth is to reduce the size, and hence also the quality, of the data itself. In the case of audio, it is possible to reduce the sampling frequency or the sample size, or switch from stereo to mono to achieve a reduced bandwidth requirement for the network.

Another approach for dealing with bandwidth problems is the use of compression. Compression is the act of transforming data to a new entity of lesser size, but where the new entity still represents the original data. The compressed data could later be an object of decompression, that is; being transformed back to data representing the original data, and with the same size as the original data. The ratio between the original data and the compressed data is called the *compression ratio*.

- compression
- compression ratio

If the original data is exactly the same as the data produced by decompressing a compressed version of the original, we say we have *lossless compression*. If the decompressed data is not the same as the original data (i.e. there has been a quality reduction), we have a *lossy compression*. Lossless compression could seem to be the obvious choice as the

- lossless and lossy compression

[1] `http://www.speex.org`

goal is to play the audio at the other end as well as possible, but as the lossy compression usually have a much higher compression ratio and still offer a tolerable quality, lossy compression is often preferred.

Chapter 5

Identifying Common Components

Contents

To evaluate the benefits of implementing a phone application using components, the application's components and their requirements need to be known. There is an infinite number of existing designs which could be evaluated, and it would be impossible to evaluate them all. The method used in this work is therefore limited to investigate several existing Internet phone implementations and an information flow architecture, to identify all common components and behaviors of the investigated applications. In addition, experience that helped in the analysis work was gained through the implementation work in chapter 8.

This chapter will take a closer look at this identification of common components. Finally their requirements to the component architecture will be discussed, both individual and as a whole.

In which degree EJB supports the identified requirements, will be discussed in chapter 6.

5.1 Investigating existing architectures

In the mid 90's there were several research projects looking at real-time audio transmissions on the Internet. In this thesis, two applications from two of the more well known projects will be investigated. To cover more of the up to date applications, the H.323 and SIP standards will be investigated as-well, as they are two often used standards in open source projects. Finally, the Infopipes architecture will be briefly covered, as it presents general elements for the building of distributed streaming applications, and as it is somewhat different from the others.

At the end of the section for each application or architecture, there will be listed candidate components for a final list of common components. The criteria for a component being considered as a candidate component, is that the component can solve its task in an other application as-well, and hence is independent of the specific composition or protocol suite. In section 5.2 the final list of common components will be presented.

5.1.1 Free Phone

Free Phone [17] is one of the early Internet phone applications. An overview of its design is depicted in figure 5.1. The design is divided into two parts; one for the application in the role of a *sender*, and one in the role of a *receiver*. In the further study of Free Phone, the transmission of a sentence from it is recorded, and till it is played back at the receiver's computer will be investigated. This will be done by going through the components in the design step by step, and inspect the behavior and responsibilities of the components.

When the user is talking, the audio is recorded by the audio input component. Before it reaches the compression component, the audio packets are filtered through echo cancelation and silence detection. When the audio is played back with a loud speaker, it could happen that the microphone catches the audio that is played back. If this happens, the other user will experience an *echo* of his own voice. The role of the echo cancelation component is to detect and filter out the audio that is recorded from the loud speaker (and not the user's speech). To do that, it needs to analyze and compare the data from the playout buffer and the audio input.

- echo cancelation

- silence detection

The silence detection component decides if an audio packet has any informational value, or if it just contains silence or white noise. Silent audio packets are filtered out and not sent to the receiver. The silence

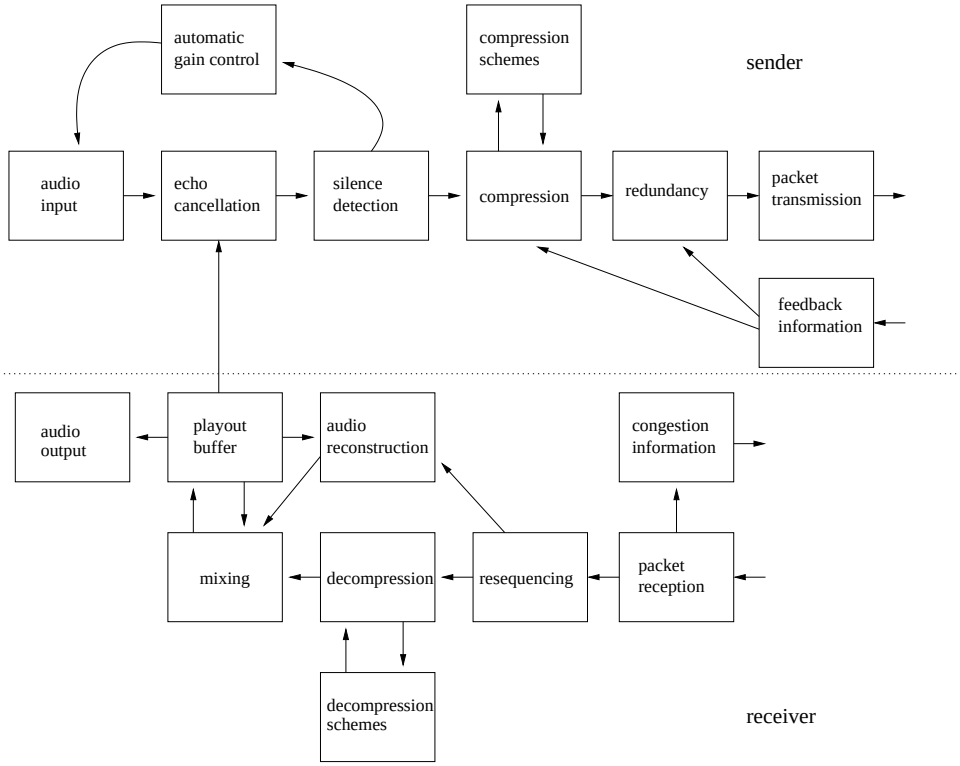


Figure 5.1: The design of Free Phone

detection component also feeds the automatic gain control component with data which adjusts the sensitivity of the microphone to normalize the audio level.

Free Phone is designed to be a robust application, and it aims to handle the unreliability of the Internet and adapt to changing resources. This is achieved using two different techniques:

- Adjustment of the size of the data
- Adding of redundancy information

The size of the data that is sent depends of the bandwidth of the recording (sample rate, bits per sample, and stereo/mono) as well as the compression rate. The compression component uses a compression scheme to compress the audio packets and make them smaller. It could be a lossless compression scheme, or a lossy compression scheme. The feedback information component receives data from the receiver about how well the data is received. If the network is congested and much data is lost, the feed back information component can tell the compression component to change its compression to one with a lower bit rate, and thus reducing the size of the data in each packet.

The other technique that makes the architecture more robust, is covered by the redundancy component. It adds extra data, called parity data, to the audio packets making it possible for the receiver to reconstruct lost data by looking at previous audio packets as well as the parity data. The redundancy data is "piggy-backed", to a later packet. *Piggy backing* is a technique where separate (and often small) data is not sent in its own packet, but added to a packet that is sent through the network and primarily carrying other data.

- piggy backing

At the receiver, the data is arriving at the packet reception component. Information about the received audio packets is fed to the congestion information component, which reports back to the sender about the congestion. The audio packets are put back in the right order by the resequence component (as the UDP packets may not arrive in the right order). The compressed audio data is then decompressed by the decompression component, and lost data recovered by the audio reconstruction component. The two streams are then put together to one continuous stream by the mixing component. The audio is stored in a playout buffer before it is played back at the audio output.

Free Phone also uses the Real-time protocol (RTP), and to each audio packet it is added a timestamp (used to measure end-to-end delays) and a sequence number (used to rearrange the packets and to detect packet loss).

Free Phone supports conferences with multiple participants. It can either send and receive data using several unicast connections, or it can use multicast. Thus, no central server is needed.

The composition of Free Phone (shown in figure 5.1) consists of components implementing common strategies and techniques in Internet phone applications. All of the components can be found in other applications as well. Thus, for Free Phone all components in figure 5.1 are included as candidates for the list of common components.

5.1.2 Speak Freely

Speak Freely[1] was originally written by John Walker in 1991, and is therefore quite an old Internet Phone application. It has many of the same features as Free Phone, such as silence detection, jitter buffer (play-out buffer), RTP and compression. In addition it includes security by supporting encryption standards (like Advanced Encryption Standard (AES), Data Encryption Standard (DES) and Blowfish), and MD5 hashing (for signatures).

Speak Freely does not add any component candidates to the list of common components that Free Phone has not already added. Although encryption could be interesting in itself, it is in fact just an interceptor in the flow path of the audio data, that transforms data. Encryption requires normally only CPU time to perform its task, and is therefore similar to other interceptors like the silence detection component. Encryption components are not added to the list since they do not introduce any new aspects not covered by the other components.

5.1.3 H.323

More recent applications include *GnomeMeeting*[2], *CPhone*[3] and *ohphone* (the first two found at `www.freshmeat.net`). Common for these applications, are that they use the *H.323* protocol standard, which is made by the International Telecommunication Union (ITU)[4]. All three applications were using the libraries from the *OpenH323* project [5]. *ohphone* is a simple application made by the OpenH323 project. There are also other well known (and mostly proprietary) applications that use

[1] `www.speakfreely.org`
[2] `www.gnomemeeting.org`
[3] `cphone.sourceforge.net`
[4] `www.itu.int`
[5] `www.openh323.org`

H.323, like the Windows NetMeeting from Microsoft [6], which were not investigated.

H.323 is an audio/video conferencing standard, but it is in fact a collection of several standards. That includes the H.261 and H.263 video compression standards, and many audio compression standards (prefixed G.) [18]. The audio and video data is sent using RTP/Real-time control protocol (RTCP) over UDP.

- call signaling

The call signaling (like connection requests connection establishing) is covered by the Q.931 standard (used in Integrated Services Digital Network (ISDN)) and the H.225 standard. The user terminals use the call control standard H.245 to negotiate master/slave issues, and for capability exchange (as for example which compression protocols they support). All three protocols are using TCP.

H.323 also covers features to adapt to resources changes. Methods for maintaining the QoS should be handled by the end-points (H.323 terminals and H.323 gateways). For example, bitrate changes should be signaled via the H.245 FlowControl commands. Echo control should be handled at the terminals as well.

As mentioned, all the investigated H.323 applications used OpenH323. There are many parameters in the OpenH323 library that can be set to change the behavior of the application. Many of them in the H323EndPoint class:

- Silence detection can be set with the SetSilenceDetectionMode method
- The jitter buffer delay is set with SetAudioJitterDelay method
- SetAEC sets the Acoustic Echo Cancelation level.

One can also define a priority list of which compression schemes that are preferred.

H.323 supports conferences with multiple participants through its Multipoint Control Unit (MCU) [19]. In the MCU there is a Multipoint Controller (MC) which handles call signaling and conference control. There are two ways the audio data can be exchanged between the participants: The first option is to use multicast to send the audio data to the other participants. The second and centralized option, is to use the optional Multipoint Processor (MP). The MP mixes the incoming audio streams and uses unicast to send the data to all the participants. Some MPs can also convert the audio (i.e. change sampling frequency, audio compression etc) in real-time, and give the audio to each participant in their

[6] www.microsoft.com

preferred format. A conference server is added to the list of component candidates.

5.1.4 SIP

The *Session Initiation Protocol (SIP)* [20], made by the Internet Engeneering Task Force (IETF), is somewhat newer than the H.323 protocol. It has less logical components and is thus simpler [21], but it is still used as a basis for many of the same type of applications H.323 is used for.

Alone, SIP will only initiate, modify and terminate sessions. Thus, to form functional telephony systems, it has to be used together with other protocols, such as the rest of the overall IETF multimedia architectures. Examples are RTP (used for transporting the media streams), Session Announcement Protocol (SAP) (used for announcing multicast sessions) and Session Description Protocol (SDP) [22].

Most implementations of SIP are using SDP to describe the multimedia sessions [22]. SDP describes in a simple textual format which kinds of media streams a client can handle and which it prefers. SDP descriptions are carried as messages within SIP.

As mentioned, SIP is a quite small architecture, and do not offer mechanisms like echo cancelation, compression schemes or any other components actively involved in the audio stream handling. Such components have to be put on top of SIP. Thus, there are no additional component candidates from this architecture.

5.1.5 Infopipes

The *Infopipes* project seeks to simplify the building of distributed streaming applications by defining a set of building block components called *Infopipes* [23]. An application is built by connecting these Infopipes together. They have a common data interface that either pushes data or is available for being pulled.

Some of the Infopipes are:

- *Sources* which provide information to be transfered. A microphone would be an example of a source.
- *Sinks* are the destinations for the stream.
- *Pipes* transmit data. A network connection may be wrapped as a pipe.

- *Filters* perform changes on the data. Compression and encryption are two examples of filters.
- *Buffers*, such as jitter buffers
- *Pumps* are used to solve "push and pull"-incompatibility problems between interfaces. If we have connected a source that expects to be pulled, and a sink that expect to be pushed, nothing will happen. With a pump between them, the pump will pull data from the source, and push it to the sink.
- We also have *Split tees* and *Merge tees* to split and merge data streams.

There are also interfaces to control the behavior of the Infopipes, such as a *slower/faster* interface for the pumps, and *fillLevel* for a buffer. With these, we can build control mechanisms to adapt to changes in the system.

Pumps are not mentioned in the other architectures, and could have its mission, but as other components include the functionality found in pumps, they are not included in the list of component candidates. A further analysis of the need of pumps is present in the next section.

5.2 The common components

This section will present a list of selected components. The selected components contain behavior that is common in all or most of the investigated applications. The purpose of making this list, is to investigate the components to find their requirements for a component architecture. The combined set of these requirements will be considered as the requirements for a component platform in order for it to support common Internet phone applications. From the list of common components, there will be made an abstract design of an Internet phone application. This design is shown in figure 5.2.

The simplest design of an application would be to have a main data flow that uses either pushing or pulling in the same direction all the way, although it is also possible to introduce pumps (as in Infopipes) to combine pull and push interfaces. The advantage of not using pumps is that there is only one type of interface for all the components doing the same task. In a real world, pumps could be needed, but as pumps in fact add no new requirements compared to any of the active components, pumps will be left out in the list of common components to keep it simpler.

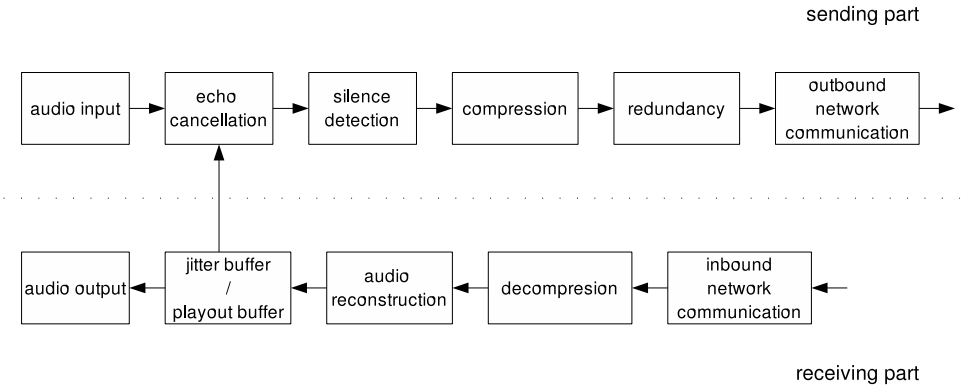


Figure 5.2: The overall abstract architecture

audio input and **audio output**
All Internet phones would need an audio input to function. The output would be a stream of audio clips. Audio output is equally needed, as it receives audio clips for playout.

echo cancelation
As long as a loudspeaker is used, echo cancelation would be needed in a phone application.

silence detection
To avoid streaming data when nobody is talking, silence detection would also be needed.

compression, **decompression** and **redundancy**
Compression is necessary unless the bandwidth is very large. Substitution of compression schemes is a common way of adapting to a network congestion situation. In an unreliable network, many packets could be lost, and if there is no time for re-transmitting the lost packets, the use of redundancy would be a possible solution to avoid gaps in the playout. To make use of the redundancy information, an **audio reconstruction** component and an **audio mixer** component would also be needed. As all these components are common and often required for the applications to perform satisfactory, these components are included in the list.

outbound communication and **inbound communication**
The sender needs to send its packets to the receiver. The communication can be implemented in several ways including the use of a direct connection and publish/subscribe mechanisms. As the identified components should not be tied into only one type of communication, two general components are made for communication; an **outbound communication**

component and an inbound communication component. All mechanisms that can be part of the communication, such as network congestion detection etc, are included into these two components. The outbound communication and inbound communication components combined would have the behavior of an Infopipe *pipe*.

jitter buffer
A jitter buffer is needed in all real-time audio applications to cope with network jitter. The size of the buffer will depend on how big the jitter is, and the maximum acceptable overall delay in the system.

The jitter buffer would also work as a playout buffer for the audio output.

GUI
A GUI component would also be needed in an application, and even though it is not mentioned earlier as a candidate component, it is included in this list. The user interface, would usually vary from application to application, and hence it could be hard to reuse. In a very simple form it *could* be reused, although its properties would have to be quite limited.

interceptors
With a common stream format, it should be possible to introduce intercepting components such as filters and monitors between two other components in the flow path. A resequencing component which assures that the audio clips are played back in the right order, is one example of an interceptor. An audio resampler is another example of an interceptor. The requirements of the interceptors would be very similar to the requirements of already listed components such as the silence detection component. To keep this list simpler, they are not included in this list.

conferencing
For support of conferencing, some additional mechanisms will be needed: As seen, multicast was used by most of the architectures to support conferencing. The use of multicast requires the inbound communication and outbound communication to handle multicast streams. In addition, somewhere in the flow path, the different streams need to be merged. A natural place to do this, would be close to or in the audio output component. All the logic before the audio stream merging should be able to treat multiple parallel audio streams independently.

Some communication systems like H.323 use a conference server for conferencing. The conference server listens to all the clients, and merges the incoming streams into one single stream. The client's inbound communication components receive the single (merged) stream from the server, just as they would from an other client.

5.3 Requirements of the Common Components

In this section, each identified component will be investigated to reveal its requirements. Some overall issues, like the information flow between the components, will be discussed as well. The identification of requirements will also be based on the gained experience from the implementation work in chapter 8.

5.3.1 The information flow

In the phone application composition, there would be two flow paths for the audio data. The flow path will be a chain of components sending data between each other. The first part of the flow path starts with the audio input component, and ends with the outbound communication component. The second part starts with the inbound communication, and ends with the audio output.

Active or passive components

The components can be designed to be either passive or active. The active components form *pressure* to the information flow that drives the audio packets through the system. In an active mode, a component would have to run in its own thread. That is; it should run continuously, only being put to sleep by performing blocking calls. A blocking situation would typically appear when the caller tries to retrieve data that is not yet ready. The method call then blocks (i.e. sleeps) as long as the data it asks for is not ready. When the data finally gets ready, the caller of the method will wake up and get the data.

An example of a method that could cause the caller to block, is a (buffer) read method from the audio API. If the thread is asking for audio data, and the audio data is not yet ready, it blocks until the audio card has sampled enough audio data to deliver the amount of data the caller is asking for. When the audio data is received, the active component would perform some action with the data before it possibly once again would invoke the read method to get more audio data. Thus, the thread would be constantly running in a loop:

1. Invoke an audio read method (and block if the data is not yet available)
2. Sleep until the method unblocks, if it blocked in point 1.
3. Perform action on/with the audio data.

4. Go back to 1.

In a passive mode, the component would depend on getting one of its methods invocated by an external object. When the component has been activated by a method invocation, it could perform any action until it returns from the method. It would then once again depend on a method invocation by an external object to become active. If the component has a deadline for a job to be done, the external object would thus be responsible for the component to be activated in time to respect the deadline.

Both active and passive components could be used, but the flow path need some *pressure* to work, that is, some action to move the data through the flow. This could be provided by at least one component such as an active audio input component that could "push" data through the system, or an active outbound network communication component that could "pull" the data to itself. It could also be just a pump (which we left out in the list of common components). Either way, at least one component would need to run actively in a thread.

Delays

As there is an overall end-to-end delay limit, any component can at most spend the time equal to the maximum delay. As this leaves no time for the other components to perform their actions, the delay should ideally be considerably less. If the end-to-end delay exceeds the maximum limit, audio clips will arrive too late, and the user will experience "gaps" in the playout.

5.3.2 Investigating each component

The audio input and the audio output components

First of all, an audio input component would require access to the audio card. There are a couple of ways this could be done: The first approach is to let the component access the audio card *directly*. The second approach is to let the component access the audio card *indirectly* through an audio *service* within the component framework, if one such exists. Both ways would be sufficient.

From the audio card or the audio service, the audio input component gets sampled audio data. In the audio input, it should also be possible to set the audio format, that is; set the audio frequency and the sample data

size, and choose between mono and stereo. The audio input would also require enough CPU to perform its tasks in time.

The audio output component would have many of the same requirements as the audio input component: Just as the audio input, it requires access to the audio card, directly or indirectly through a service.

The echo cancelation component

The echo cancelation component is a filter in the path of the audio stream. If an echo is detected, the clips containing echo are filtered out, abd not sent to the next component in the path.

The component would require access to the playout buffer of the audio output, either directly, or through an interface, in order to detect the echo: It should be able to access the audio clip currently played and possibly one or more clips played before the currently played clip. The only requirement to the component framework would be to let the two components involved have access to each other's data in one way or another. This should not be a problem as the composition would be impossible to implement at all if the framework did not offer inter-component communication. Thus, this requirement should be assumed provided by all component frameworks.

This filter, as all the other filters, should also have enough CPU time to perform its action within the time limit (of the maximum end-to-end delay).

The silence detection component

The component should get enough CPU time to, within the given time limit, decide whether the audio clips contain valuable information to hand over to the next component in the path, or just silence, which should be stopped.

The compression and decompression components

The compression and decompression components have the same requirements as the silence detection component, but could possibly require more CPU time to avoid exceeding the delay limit.

The inbound and outbound communication components

The requirements of the communication components will depend on the type of communication used. One possible communication model (used by all the investigated applications in 5.1) uses direct communication, i.e. the application establishes a connection to its counter part, and receives connections itself. In that case the application acts as a server for incoming audio packets, and the inbound communication component will need to create network sockets (or access the equivalent functionality offered by the current OS). The component should be able to handle connection requests, establish connections, and receive packets from the network. Thus, the component should always be or become active when a connection request arrives.

In the same case, the outbound communication component will need to establish connections to its counter part. Depending on the communication protocols of the application, it could require the availability of one or more of the UDP, TCP and RTP/RTCP protocols.

An other possible communication model introduces the use of communication services. This means that the application do not send and receive data itself. Instead it will submit data to a service which sends it to one or more receivers on the application's behalf. In that case, the requirements of the components will be the presence of such services, and the possibility to be called back by the service when incoming data arrives.

A configuration of an application will also have QoS requirements for the communication components like maximum end-to-end delay, minimum bandwidth and maximum jitter. If these requirements can not be fulfilled, the application would have to be reconfigured to adapt to the current network performance.

The jitter buffer and resequencing components

The jitter buffer is basically a simple first in - first out (FIFO) queue. Since it only stores the audio clips in advance to even out the variable time difference between the arriving audio clips, the only important requirements would be some mechanisms to avoid race conditions (flaws occurring when parallel processes are simultaneously accessing and updating the same data, causing the result of the operations to depend on the timing of the events).

- race condition

The resequencing component would be equal to the jitter buffer, except that it will not be a FIFO queue as it sorts the incoming audio clips by its time stamp set by the sender.

The GUI component

The GUI component would require some kind of access to the graphic API. It could be done directly, through a graphic server running in the OS (such as the X server in Unix), or via a service within the component server.

The component would also have to run actively to handle user inputs such as button clicks, mouse movements etc., or alternatively; become active when a user input takes place.

5.3.3 Summary of requirements

After investigating each component, this short list of requirements to the component platform could be made:

- Creation of threads should be possible. Alternatively the components should get activated by a timer often enough to perform as necessary.
- Enough CPU time should be given to the components to avoid breaking the maximum delay limit.
- Depending on the communication model, the components could need to create connections and send network packets.
- The components could also need to act as servers, receiving connection requests and incoming network packets.
- One or more of the following network protocols could be needed; UDP, TCP and RTP/RTCP protocols
- The network communication services must offer enough bandwidth, and satisfying values for jitter and end-to-end delay, or else the application must be notified so a reconfiguration of the application can take place
- Access to hardware and lower level services (directly or indirectly) is necessary, such as for the audio card and the desktop (user interaction).
- Data that is accessed by more than one component, should have mechanisms to avoid race conditions.

An analysis of how well EJB supports these requirements will be discussed in the end of chapter 6.

Chapter 6

Java 2 Enterprise Edition and EJB

Contents

First in this chapter, an overview of J2EE and its technologies, including Enterprise Java Beans (EJB), will be presented. There will then be an introduction to JBoss, the EJB implementation used in this work. Finally, the possibility of implementing the common components identified in chapter 5 using EJB, will be discussed. This will be done by investigating the identified requirements as well as the EJB standard [24].

6.1 Java and J2EE

Sun Microsystems[1] is the owner of the Java technology. Java seeks to offer a cross-platform programming environment, where the programmer is not dependent of certain Operating Systems or special hardware. A

[1] `http://www.sun.com`

program written in Java can be compiled to Java Byte Code. The byte code can then be executed in a Java Virtual Machine (JVM). Such JVMs are available for many OSes and hardware systems, and make the OS and hardware differences invisible to the application. Thus, it is possible to run the same compiled program on different platforms without recompiling. In other words, Java is both a programming language and a virtual machine.

The current major version of Java is Java2. Java2 is delivered in three different versions: *Java 2 Standard Edition (J2SE)*, *Java 2 Micro Edition (J2ME)* suited for a mobile/wireless environment, and finally *Java 2 Enterprise Edition (J2EE)*. J2EE is J2SE with an added collection of technologies which is commonly needed in an enterprise environment. These technologies are also standardized by Sun Microsystems.

Some of the added technologies are:

- **Java Database Connectivity (JDBC)** - JDBC performs SQL queries on a relational database, through a standardized API. JDBC supports DBMSes (Database Management Systems) from different vendors, but seeks to hide the differences of the systems to the client.
- **Java Naming and Directory Interface (JNDI)** - JNDI supports different naming and directory providers, such as DNS, and in the same way as JDBC, it seeks to offer the client a standardized way of performing a request on the services, hiding their differences.
- **Java Transaction API (JTA)** - JTA offers local and distributed transaction services for the client.
- **Java Messaging Service (JMS)** - JMS is a standardized interface for access to Message Oriented Middleware servers which supports reliable asynchronous message transfers through publish/subscribe and point-to-point models. We will look more into JMS in section 6.4.
- **Remote Method invocation (RMI)** - RMI provides transparent access to remote objects. RMI is further explained in 6.2.
- **Java Server Pages (JSP) and servlets** - These are standards for providing an environment for container managed components, most commonly used for creating dynamic web pages through HTTP(S).
- **Enterprise Java Beans (EJB)** - EJB components, called enterprise beans, are run in a container which manages the components. We will look more into EJB in section 6.3.

- **Java Management Extensions (JMX)** - With JMX, Sun seeks to offer a standard way of managing resources such as applications, devices, and services [25]. Read more about JMX in 6.5.1 on page 44.

6.2 Remote Method Invocation (RMI)

The intension of Java RMI is to make the introduction of distribution easier. RMI enables local objects to perform method invocations on remote objects as if they were local. The distribution is transparent for both the caller object and the remote object.

RMI can communicate using two different protocols: *Java Remote Method Protocol (JRMP)* is RMI's native protocol. RMI can also use *Internet Inter-Orb Protocol (IIOP)* which makes it possible to access objects running on the Common Object Request Broker Architecture (CORBA) platform.

CORBA is defined by the Object Management Group (OMG)[2]. The basic version does support distribution but not components. For the use of components, OMG defined the CORBA Component Model (CCM) [26].

6.3 Enterprise Java Beans (EJB)

EJB is Java's distributed component model. The components are called Enterprise Java Beans (not to be compared to Java Beans which have no support for distribution).

In EJB the beans are run in a container, which manages components and performs method invocations on behalf of its clients (see figure 6.1). The container has the responsibility for extra functional aspects of its beans, such as the life cycle: When a new bean is needed, the container creates and launches it. In the same way it can destroy the bean, or temporarily cache it to disk to be activated when needed. The container also makes the location of the caller transparent, and the bean does not need to know if the calling object is a local object or a remote object using RMI.

The goal of EJB is to enable the programmer to focus on the component's functional aspects alone, as the business logic is separated from the underlying technologies. Using the technologies listed in section 6.1, Sun claims that the maintainer should be able to replace a resource, e.g.

[2] `http://www.omg.org`

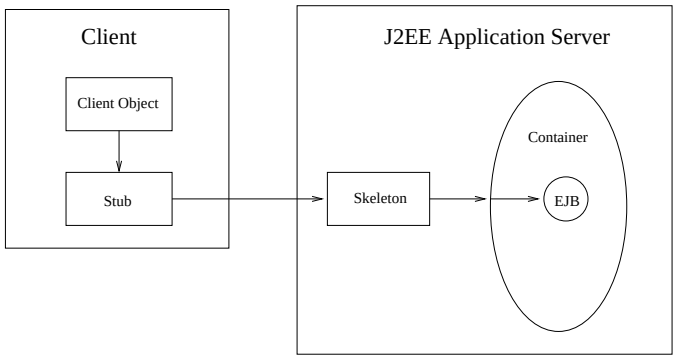


Figure 6.1: An example of a method call on an enterprise bean (EJB)

the computer architecture or a Database Management System (DBMS), without reprogramming the components. Such separation of concerns can be useful as the need for changes in the business logic and changes in underlying software resources not necessarily occur at the same time.

There are four types of enterprise beans:

- Stateless Session Beans
- Stateful Session Beans
- Entity Beans
- Message Driven Beans

6.3.1 Session beans

Session beans are used to implement business logic. They are stored in memory, and thus will not survive a server crash or server shutdown, but for the same reason they are faster to execute than entity beans (see 6.3.2). There are two types of session beans; *Stateful* session beans and *stateless* session beans.

Stateful session beans

When a client calls on a stateful session bean, it gets its own copy of that bean. The values of the variables in the bean is persistent between method calls, and thus a stateful session bean can be used to store data for a short while. As the data is stored in memory, it will be lost in a server shutdown or server crash.

An example of a situation where a stateful session bean is useful is a shopping cart in a web shop; The bean stores all items in the cart until the user submits the order or aborts.

Stateless session beans

Stateless session beans do not store any data between calls, and thus a client does not "own" a particular bean. The server has a pool of stateless beans, and offers any of them to a client that wants to invoke a method on a bean. A method call on a stateless bean can therefore be faster than a call on a stateful bean, as the server has to find a particular (or create a new) bean for an invocation on a stateful bean.

6.3.2 Entity Beans

The entity beans are designed to store persistent data. The server has the responsibility for their persistence, and thus a bean will survive a server shutdown or a server crash. An entity bean can map to any persistent store such as a file, but a relational database through JDBC is probably most common. The client does not have to invoke any methods to store the data on the persistent store, as this is managed by the server.

The persistence can be managed either by the container, Container Managed Persistence (CMP), or the bean itself, Bean Managed Persistence (BMP). In the case of using CMP, the container has the full responsibility of storing the bean in a persistent store. The bean developer do not have to know which kind of store the container uses. The developer can also make his own methods to preserve persistence using BMP. In that case the developer creates the code for retrieving and storing data to the persistent store, but the server still has the responsibility of calling those methods (the user cannot call them himself).

An entity bean can be found by calling a finder method in the *Home interface* (see 6.3.4). The identity of the bean is represented by a (unique) *primary key*.

6.3.3 Message driven beans

A Message Driven Bean (MDB) is a bean that "listens" to a Java Messaging Service (JMS) topic. When a message is published in the topic, the MDB reads the message, and performs an action (as for example invoking method calls on other enterprise beans). MDB's were added to

the EJB standard in version 2.0. One of the implementations described in chapter 9 will use an MDB.

6.3.4 The Remote and Home interfaces

When a client asks for a bean, or performs a method invocation on a bean, this is done by calling on methods defined in the *Home interface* and the *Remote interface*.

The Remote Interface

The remote interface declares the methods in the bean on which the client can perform method invocations. The interface is a Java interface, so the methods are not implemented, and the interface contains only the signatures of the methods. The EJB server uses the remote interface to create the remote skeleton.

The Home Interface

The Home interface is a Java interface declaring the remotely invoked methods for life cycle management, such as *create*. It also declares the finder methods for entity beans. As with the Remote interface, the server creates an implementation of these methods.

Introduction of Local interfaces in EJB 2.0

As method invocations on beans running locally were quite slow using the remote methods, EJB 2.0 introduced interfaces for locally invoked methods. Thus the prior *Home interface* was renamed *Remote Home interface* and a *Local Home interface* was introduced. The Local Home interface is, as the name indicates, used for local method calls from objects within the same JVM, and it is more efficient. In the same way the *Remote interface* got a local equivalent called *Local interface* for method calls from local objects.

Even though local invocations are executed more effectively, there is a drawback using this approach: If a component is made using the local interface of an other component, it cannot be reused in a setting where the other component is a remote component. In other words, this undermines some of Sun's goals with the architecture.

6.3.5 Deployment

An enterprise bean ready for deployment is stored in a .JAR file. A JAR file is actually a ZIP file (a commonly used archive file type storing compressed files). The "JAR" file extension indicates that the zip file contains Java class files, and is a native Java archive file type for bundling files.

The JAR file used for deploying an enterprise bean contains the class files for the bean itself, and class files for the remote interface, the home interface and (possibly) the local interfaces. In addition to the class files, the JAR file contains a directory named *META-INF* which contains the Extensible Markup Language (XML) deployment descriptor file named *ejb-jar.xml*. The *ejb-jar.xml* is part of the EJB standard and has to be present.

In addition to the ejb-jar.xml file, the META-INF directory can contain server implementation specific deployment information. In the case of JBoss this file is named *jboss.xml*. The introduction of server specific deployment information may reduce portability.

How to actually deploy the JAR file on the server is server specific, and not a part of the J2EE standard. See chapter 6.5 for information on deployment of enterprise beans in JBoss.

6.4 Java Message Service (JMS)

JMS (Java Message Service) has been developed by Sun[3]. In this thesis work it is used for distribution of audio clips between the Internet phone clients. JMS provides two messaging models:

- Point-to-point with *queues*
- Publish/subscribe with *topics*

A *Queue* can have several producers and consumers, but a message cannot be received by more than one receiver. A JMS queue is, in other words, a first in - first out (FIFO) queue where several producers can put messages on top of the queue, and several consumers can remove messages from the bottom of the queue.

A *Topic* can, just like a queue, have several producers and consumers, but the messages are received by *all* the consumers: When a producer publishes a message to the topic, JMS distributes that message to all the consumers subscribing to the topic.

[3] `http://java.sun.com/products/jms/`

The consumer must implement the interface MessageListener. Its method onMessage will be invocated with the message as parameter each time a new message arrives.

6.5 An introduction to JBoss

JBoss is an open source J2EE application server made by the JBoss Group[4]. The server consists of containers for enterprise beans and web servlets, and services like JMS and JNDI (see the next section). The JBoss Group consists of developers from all over the world, and is headed by Marc Fleury.

The support of JBoss is handled by JBoss Inc, which is a worldwide organization, founded by the core developers of JBoss[5]. They make money on support, and by selling documentation.

JBoss now supports most of the J2EE standard, and after an agreement with Sun Microsystems, the JBoss group is currently working on certifying JBoss.

6.5.1 The JBoss core

Many EJB servers are monolithic and have all the services as part of their core servers. JBoss is different in that respect; One of Sun's ideas with J2EE is to make it possible to create applications in a modular way where replacement of any component in the application should be simple. In the same way, JBoss is made with a small core (a so-called micro kernel), and the rest of the services are implemented as components connected to the core (see figure 6.2). The core is named the *JBoss server spine* [27] and is built around an implementation of the Java Management Extensions (JMX) standard.

Sun's goal with JMX was to provide a simple, standard way of managing resources, such as applications, devices, and services [25]. A resource is managed by a Managed Bean (MBean) which is registered in a MBean server. The MBean server acts as a management agent, and thus completely controls the MBeans. As it supports distribution, the resources can be discovered and controlled by remote objects through the MBean server. In JBoss, the MBeans that are registered at the JMX

[4] http://www.jboss.org
[5] http://www.jboss.com

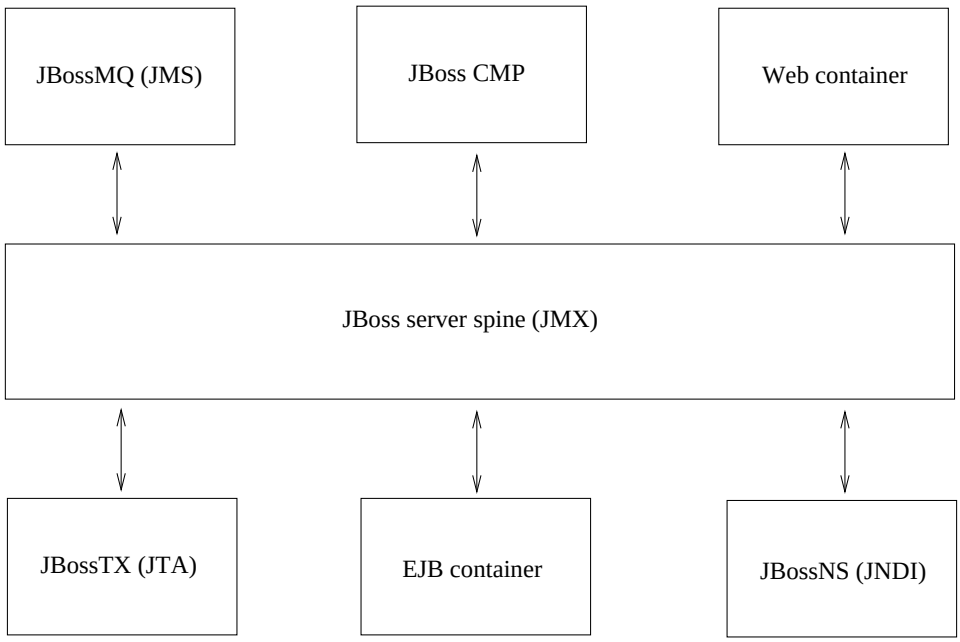


Figure 6.2: JBoss server spine with some services

server can be seen by looking up the web page `http://localhost:8080/jmx-console`

All the services in JBoss (such as JMS, JNDI and the EJB container) is made as MBeans which is managed by the JBoss JMX server. Any service can supposedly be replaced by an equal service from another vendor by replacing the MBean. This design also makes it possible to add new services to the JBoss server and is thus relevant for the analysis and suggested changes to the J2EE standard in chapter 10.

6.5.2 Deployment

To deploy a component, the file (i.e. the JAR file) must be copied to the directory *server/default/deploy* in the JBoss root directory, and JBoss should discover the new file within few seconds. As JBoss deploys the component, it prints system information about the deployment. The result of the deployment (if it was successful or not) will also be printed to the console. Any component or service can be deployed while the applications are running, and without re-starting the server. JBoss will keep track of the dependencies between services [27].

6.5.3 Creating a JMS topic in JBoss

To create a new JMS topic in JBoss, the name of the topic and its attributes must be defined in the file *jbossmq-destinations-service.xml* in the *server/default/deploy/jms* directory. This is how the topic *conferenceTopic* used in the implementations in chapter 8 and 9 is made. When the XML file is saved, JBoss detects the file's changed time stamp, and deploys the file, and thus creates and activates the new topic.

6.6 EJB support for the requirements

In this section, the possibility of implementing the common components identified in chapter 5 using EJB will be discussed. This will be done by investigating the requirements identified in the same chapter, as well as investigating the EJB specification [24].

The following sections will address each requirement separately. The requirements will be repeated in the beginning of each section.

6.6.1 Creating threads

> *Creation of threads should be possible. Alternatively the components should get activated by a timer often enough to perform as necessary.*

The EJB 2.0 specification [24] (hereby called the *specification*) forbids the use of threads:

> *The enterprise bean must not attempt to manage threads. The enterprise bean must not attempt to start, stop, suspend, or resume a thread, or to change a thread's priority or name. The enterprise bean must not attempt to manage thread groups.*

The reason given for this restriction is that if an enterprise bean was allowed to spawn a thread, the container would no longer be in full control of the resources within the JVM, and it can no longer manage the life cycle of its beans properly. In the following sections, the different kinds of requirements for continuous execution will be identified. Finally, there will be discussed if alternative functionality can be used in the place of threads.

There are different needs for continuous execution of the audio application components, and the components can be defined into two different groups: The first type requires the component to give immediate response when a given situation occurs. The only component that have such requirements is the GUI which manages this without threads in Java. The GUI is implemented using the Advanced Windowing Toolkit (AWT) API which uses events for handling the response from the user. Every time a user moves his mouse, pushes a button or writes some text, an event is produced. Other objects can "subscribe" to those events by extending an event interface and implement an event handler. Each time the user pushes a button or does anything else that the subscribing object needs to know about, the event handler is executed. Thus, no components in this category in the application needs threads.

The other type of component has to perform certain tasks periodically and cannot be delayed. Examples of this would be the audio output or the audio input components. When a new buffer is ready and full of sampled audio for the audio input, the data would have to be handled very quickly and within a certain time. When a component is running in its own thread, the solution is rather trivial (and discussed in section 5.3.1). The component does not have to run continuously though, but it *does* have to be running regularly to perform its task. Thus if the component got activated (that is, put in a running state) by another object when needed, the component could still perform as required.

There are two ways a component could get activated (except for running in its own thread and set in a running state by the CPU scheduler):

1. The activation could be done by an other object performing a method call on the component.

2. The component could get activated by a timer

The first option requires that the object activating the component is not running in the EJB container. If it was running in the container, it would also have the same requirement for running actively in a thread as the component it should activate. Thus, the component would have to depend on an object running outside the container, which means that the whole application cannot be made in EJB.

The EJB container offers a service called the *EJB Timer Service*. The Timer Service can be used to perform callbacks to the enterprise beans. This can be scheduled for a given time, or at specified intervals. The timers can be used for entity beans, stateless session beans and message-driven beans through the EJBContext interface. Note that stateful session beans can not make use of this service. When the specified time occurs,

the timer performs a method call on the ejbTimeout method defined by the javax.ejb.TimedObject.

The intended use of the EJB Timer Service is not real-time use, though. The time delays for this service should be measured in hours or even days according to the specification. Using the EJB Timer Service to activate the audio input would hence be quite risky, since the required maximum deviation to the specified callback time should not be more than a dozen of milliseconds. There is absolutely no guarantee that the ejbTimeout method will be invoked within the given deviation limit. A one second deviation to the specified callback time for a "once per day" event is negligible. A one second deviation for a "once every 250 milliseconds" event is crucial.

6.6.2 CPU power

> *Enough CPU time should be given to the components to avoid breaking the maximum delay limit.*

Managing threads is, as previously stated, not allowed. Hence, changing a thread's priority is prohibited. Since the EJB API does not offer any other way of giving priority to certain tasks, there will be no way to make certain that the required CPU priority will be offered.

A real-time client (i.e. a client running in a real-time environment) could guarantee that a method invocation takes place within a given time. The enterprise bean might still run in a non real-time container, though. Since the underlying OS and VM are responsible for the scheduling of the VM and the container respectively, the response time cannot be guaranteed unless the OS and the VM give real-time guarantees. In addition, the container is free to manage all the threads running inside it. The container may therefore offer guarantees as well, otherwise the enterprise bean cannot offer any. During the work of this thesis, neither the OS, VM nor the container gave such guarantees.

6.6.3 Creating and receiving network connections

> *Depending on the communication model, the components could need to create connections and send network packets. The components could also need to act as servers, receiving connection requests and incoming network packets.*

Creating network socket connections is allowed by the specification. This means that a client can use any supported protocol to send information to a receiver. Receiving network connections on the other hand, is not allowed. The specification states:

> *An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast.*

The specification prohibits this since the idea of EJB is to serve EJB clients. This means that there are two possible ways of receiving the audio packets if a communication model with direct communication (i.e. not using communication services) is used:

1. The enterprise bean can get the audio packets by receiving regular method calls through the remote interface (using Remote Method invocation (RMI) over the network).
2. The enterprise bean can be polling the sender. If both the sender and receiver are enterprise beans, neither the sender can receive network connections. That means that polling is only an option if the sender is not an enterprise bean.

Using regular method calls for this task means that the programmer cannot choose which network protocol to use. He has to be satisfied with the underlying protocols of RMI.

If the application, instead of managing its own communication, relies on a service to perform the communication on its behalf, there would be no such restriction problems. The JMS could be such a service. A client can post audio packets (or actually audio messages using JMS terminology) on a JMS topic, and subscribe to the same topic to receive all messages posted on the topic.

6.6.4 Supported protocols

> *One or more of the following network protocols could be needed: UDP, TCP and RTP/RTCP protocols*

All the protocols to be used must either be bundled with Java, or if any protocols are not a part of standard Java, they must be made in pure Java code, i.e. not using a native library. Method calls to native libraries are not allowed by the specification as they would cause a huge security risk.

The package java.net, which is a part of standard Java, supports TCP, UDP and multicast, but there are also other protocols available that do

not use native libraries. An example is an RTP implementation made by Columbia University, New York [28].

6.6.5 QoS requirements for network services

> *The network communication services must offer enough bandwidth, and satisfying values for jitter and end-to-end delay, or else the application must be notified so a reconfiguration of the application can take place*

A configuration of the application would need a certain (defined) quality of the communication (network) service in order to function adequately. This could be supported in two ways: Network protocols with guarantees like DiffServ and Intserv could be used for the all network communication, like e.g. RMI and JMS in the case of EJB. This is what is done in [10]. Alternatively, the application would have to be re-configured to adapt to changes in the network performance when the quality drops under the given limit. A monitoring service of the QoS status would thus be required.

6.6.6 Race conditions

> *Data that is accessed by more than one component, should have mechanisms to avoid race conditions.*

Both the audio output and the packet reception (possibly with some components like decompression and resequencing in between) will access the buffer with the audio clips. When two or more components are working on the same data, race conditions could occur.

In a non EJB setting, it would be natural to store the audio buffer as a local array in a component. In EJB this cannot be done, as only stateful session beans can store data, and there is supposed to be a one-to-one mapping between a component and an instance of a stateful session bean. Thus, data that has to be accessed by more than one component, has to be stored elsewhere. In an EJB setting, *entity beans* would be the solution. As the EJB server has the responsibility to avoid concurrent access by two components, the requirement for this application is thus satisfied. The only question is whether this solution will be quick enough, or not. The entity beans are often stored in a SQL database through Container Managed Persistence (CMP) (see 6.3.2). This could

be too slow for an real-time audio application. The use of Bean Managed Persistence (BMP) with empty persistence methods can possibly solve this time consumption problem.

6.6.7 Accessing hardware and lower level services

> *Access to hardware and lower level services (directly or indirectly) is necessary, such as for the audio card and the desktop (user interaction).*

As stated in chapter 5, there are two categories of hardware / lower level services that have to be accessed (that is, in addition to the network card), namely the audio card and the services for user interaction.

Graphics display, keyboard and mouse

In Linux and other Unix flavors, the user interaction is usually communicated through the X server. The software connects to the X server as a client, and the X server displays all the graphic on behalf of the clients using the graphic card. X also forwards information from the keyboard and the mouse back to the client software.

In Java, software with a Graphical User Interface (GUI) creates its windows and other sub elements in the GUI using AWT, or AWT's successor, Swing. But the EJB specification do not allow this:

> *An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard.*

The specification explains this restriction by saying that most servers do not allow user interaction to happen using the server's keyboard or display.

As accessing the AWT (or Swing) API is not allowed, and as there are no service within the container offering access to the hardware, the user interaction has to be managed and take place outside the EJB container.

Audio card

Access to the audio card is not mentioned in the EJB specification at all. The reason is probably that the intended type of applications to be made in EJB is distributed business applications, as proclaimed at the first page of the EJB specification.

Actually, the specification do not mention hardware access or access to lower level services in general anywhere in its list of restrictions. Only specific restrictions as creating network sockets are mentioned. Another such example of a specific restriction for access to a lower level service is the use of the java.io API.

> *An enterprise bean must not use the java.io package to attempt to access files and directories in the file system.*

The reason is rather obvious: Storing data using files in a multi container environment could cause race conditions and might also be inefficient. The use of JDBC or other similar resource managers is necessary.

Direct access to hardware or lower level services would make it impossible to replace one server with a server cluster. If audio sampling is done in a cluster, one cannot be sure of which audio card on which computer will be used for the sampling. One EJB container could also run on more than one JVM which could also produce some race condition problems when accessing the hardware/service.

EJB is designed for scalability and the hardware/service access in a scalable environment would require the presence of a resource manager for the hardware/service. Thus, for access to an audio card, an audio card manager would have to be present so all components regardless of its location can access that specific audio card.

Even though the EJB specification does not mention hardware or lower level service access as a whole or audio card access in specific, it is reasonable to believe that such access is not meant to be legal and would be stated illegal if the designers had an impression that many programmers would try this.

Chapter 7

Designs following the EJB specification

Contents

This chapter will look at possible designs of an Internet phone system. The designs should follow the EJB specification and not violate any of the programmer's restrictions. The possibility of implementing only some of the identified components as enterprise beans, and the possibility of splitting a component to make only parts of it as an enterprise bean, will be investigated. The motivation for this work is to learn more about the limitations of EJB, and to what extent it is possible to utilize EJB for an Internet phone application without violating the standard.

7.1 Implementing all components as enterprise beans

The discoveries in section 6.6, shows that implementing all parts of the client using EJB components would be impossible.

- The GUI can not be implemented in EJB, as using AWT is not allowed.
- The audio output and audio input that want to access the audio API in the OS are not supposed to do so.
- The inbound network communication is not allowed to create and listen to sockets, so its only option is to receive packets through method calls and JMS
- Many of the components, such as the audio output and the audio input require either to be able to create threads (which is not allowed) or being made active by a timer or an other object. The container's EJB Timer Service is not intended for real time use, and being called periodically by a client violates with the thought of implementing all parts using EJB.

In addition, EJB do not have any resource management for CPU scheduling and network communication. The lack of resource management in general will not make it impossible to create an Internet phone application, but the application could experience performance problems.

In other words, an implementation of the client using enterprise beans would be practically impossible, but there are still possibilities for implementing components that are not affected by the restrictions. An application using enterprise beans for parts of its design would thus be possible. This approach will be discussed in the next section.

7.2 A design partially made using EJB

In a design that is partially using EJB, the design is divided into two parts: One part is running within an EJB container as one or more enterprise beans. The other part runs as a "regular application", that is; an application that runs outside the EJB container. These two parts communicate with each other, and together they form a fully functioning application

All the components with requirements that are not in conflict with the EJB specification, are placed in the EJB container as enterprise beans. The ones that *are* in conflict with the specification are implemented as objects within the part outside the container. This part running outside the container will from here on be referred to as *the main element*. The main element will have the responsibility of initializing the application and establish connections to the enterprise beans.

This separation of the design makes it possible to utilize the advantages

of EJB for the components running within the container. The ones running in the main element are run outside the container, and thus the EJB specification is not violated.

This section will look at this design solution and discuss its pros and cons.

7.2.1 Which components can be made as enterprise beans?

As seen in figure 7.1, the audio input, the audio output, and the GUI can not be made as enterprise beans without violating the EJB standard. Whether the network communication components can be made as an enterprise bean, depends on which techniques that are used for the communication. The identified components that can not be implemented as enterprise beans must therefore be implemented in the main element.

The rest of the components are considered possible to implement as enterprise beans in this scenario (the discussion of why this is so follows below):

- silence detection
- echo cancelation
- compression
- redundancy
- jitter buffer
- outbound network communication (depending on its design)
- inbound network communication (depending on its design)

The silence detection determines whether one or more audio clips contain audio or just silence. This is done as a calculation on the audio data, which then is sent as a parameter to the method that is invoked. The enterprise bean does not require access to other resources to perform this job.

The echo cancelation component has similar functionality as the silence detection component, but it needs access to previously played audio clips as well as the present audio clips. This can be solved in many ways: The main element can send the previously played audio clips as a second parameter. The enterprise bean could also access the audio output buffer (this can be done if the buffer is implemented using entity beans; see below). A third solution would be to implement the echo cancelation as a stateful session bean and let the audio output invoke a method in the

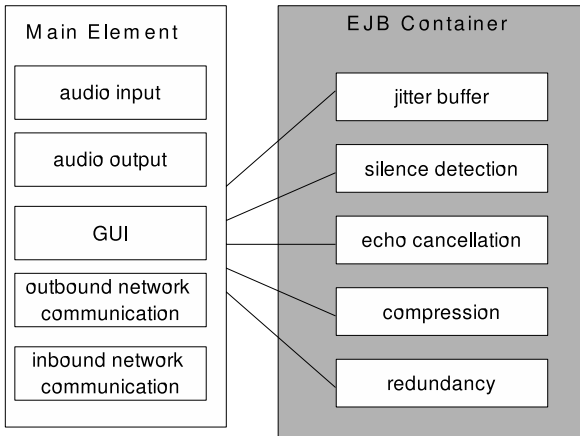


Figure 7.1: An overview of the application. The lines represent communication between the enterprise beans and the main element

bean each time it receives a new audio clip. All three methods would work, but the design of the main element must be compatible with the method used.

The compression component reads audio clips as input and returns compressed audio clips. Depending on the compression scheme in use, it could compress each audio clip independently, or it could need to work on several clips together. Its design will thus be similar to the designs of the silence detection and echo cancelation components.

The design of the redundancy component would be similar to the designs of the other components (above). However, in contrast to the other components, the redundancy component will *always* need to access several audio clips simultaneously; because additional information is added to an audio clip to make it possible to recreate information about other lost audio clips.

The jitter buffer could either be implemented as a stateful session bean, or as an entity bean. As previously mentioned, stateful session beans have a one-to-one mapping with their clients, which makes it impossible for multiple clients to access data in the same instance of a session bean. This means that if the jitter buffer is made as a session bean, only the main element should access the bean; all other parts, i.e. the other enterprise beans, should get access to the buffer through the main element.

The jitter buffer could also be made as an entity bean. With entity beans, it is possible for multiple clients to access the buffer, and hence, in addition to the main element, also the other enterprise beans could access

the buffer directly. The entity bean could either be made with Container Managed Persistence (CMP), or Bean Managed Persistence (BMP). In the case of CMP, the performance would depend on the way the container handles the persistence. It is thus unknown to the designer how efficient this would be as it would vary between different container setups. In the case of BMP, the programmer could take care of the persistence himself. It would be possible to create empty BMP methods that would cause the enterprise bean to not survive a server shutdown or its removal from memory (which happens if the bean is not used for some time). This could thus make entity beans faster.

The outbound network communication can be made without concerns for restrictions, as long as it only sends data. Depending on the communication type, the component could also include congestion feedback logic, which would involve logic for receiving data as well. The enterprise beans are not allowed to create sockets and act as servers, so receiving data can only be done if they use regular EJB method calls, or if a bean uses services like JMS to receive data.

Whether the inbound network communication can be made as an enterprise bean or not, depends on the mechanisms used for the communication. As with the congestion feedback logic in the outbound network communication component, discussed above, this component can only be made as an enterprise bean if it does not have to create sockets and to act as a server. This means that the incoming data must be received using regular EJB method calls, or by using services like JMS.

7.2.2 Discussion about the solution

An important disadvantage of this design would be the inflexibility compared to a design where all the components are running on a component server. The identified components implemented as enterprise beans can be replaced, either by a reconfiguration of the application, or by a physical replacement of the bean. The other identified components are implemented as objects in the main element and can not be replaced without altering the source code, and hence the main element would have to be recompiled. This makes it hard to reuse different parts of the application.

Another problem is that the enterprise beans can not access the main element in the same way as they access other enterprise beans, unless the main element is extended with RMI (see the next paragraph). They are all invoked by the main element, and the data they need from the objects within the main element will have to be provided as a parameter to the invoked method.

Access to the main element could be made possible though, if the main element is implemented with RMI. If the main element also registers itself using JNDI, the enterprise beans can then look up the main element more or less in the same way as they look up other enterprise beans. This is possible without breaking any restrictions since acting as a network client is allowed by the EJB specification.

A third concern is the network traffic the method invocations cause. In the interaction between enterprise beans running in the same container, local interfaces can be used, which removes the network protocol overhead. Between a client outside the EJB container and an enterprise bean, RMI has to be used, and even though the client and the EJB container run on the same computer, the communication is going through the protocol stack, which gives an overhead compared to the use of local interfaces. The reduction of performance using RMI was the reason why Sun introduced the local interfaces in the EJB 2.0 specification.

7.3 Splitting the identified components

The previous section investigated whether the identified components could be entirely implemented as enterprise beans or not, and divided them thereafter. A second way of designing an application partially made of enterprise beans, is to split an identified component into two parts: The logic that is not in conflict with the EJB standard can be made as enterprise beans. The rest of the logic will be placed in the main element. This is interesting since it makes it possible to create larger parts of the application as enterprise beans. This section will investigate this design alternative and how it can be implemented. Finally the solution will be discussed.

7.3.1 The design principle

When a component is split, the two new elements will work in a pair, that is, the two new elements will together perform the task of the old component. One part is implemented as an enterprise bean running in an EJB container. The other part can be created inside the main element as an object. An example with a split inbound network communication component is shown in figure 7.2.

In this example the inbound network communication will use a protocol that makes it necessary to create network sockets. This will be done in

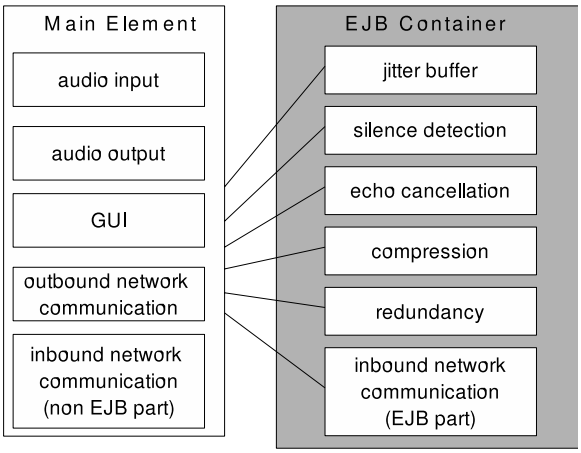


Figure 7.2: An overview of the application where the **inbound network communication** *component is split*

the main element since creating sockets can not be done within an enterprise bean. A second task of the component is to resequence the audio packets after the packet receptions. This requires only access to the jitter buffer, and could thus be done by an enterprise bean if the jitter buffer can be accessed from within the container. If the jitter buffer is created using entity beans, it is will be reachable from within the EJB container. The jitter buffer can also be implemented as an object within the main element, but it can still be accessed by the enterprise bean if the main element has an RMI interface for the jitter buffer. A third task for the **inbound network communication** component is to detect network congestion and to inform the sender about this. This can also be done within an enterprise bean. The **inbound network component** is thus divided into two parts (see figure 7.3):

The first part is implemented within the main element and will receive the audio packets. The second part is implemented as an enterprise bean and will resequence the packets and detect network congestion. The two new elements of the old component must work together to solve the task of the old component.

7.3.2 Splitting the audio and GUI components

The **audio input** and **audio output** components have two requirements that are in conflict with the EJB restrictions: They access lower level services (the audio API), and they requires threads or other mechanisms

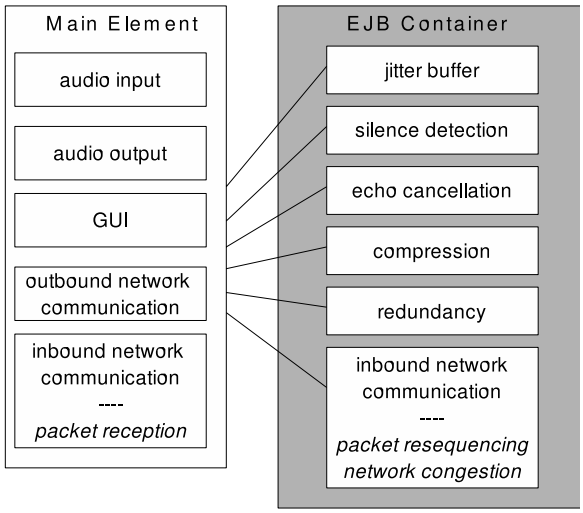


Figure 7.3: The inbound network communication component is split, and the tasks of the two parts are shown in each part

to execute code regularly. Also the GUI requires access to a lower level service.

The use of threads in the enterprise bean can be avoided by letting the main element invoke a method in the enterprise bean regularly. The responsibility for the enterprise beans timeliness would thus be in the main element. The enterprise bean could either be invoked by a corresponding component running in its own thread in the main element (i.e. by splitting the identified component as previously described), or by a dedicated *EJB method invoker*. Such an EJB invoker could be created with a list of stubs to all the enterprise beans to be invoked, and in a round robin fashion it can invoke each enterprise bean (see figure 7.4). When all the enterprise beans in the list have been invoked, the first enterprise bean in the list is once again invoked.

The timeliness of an enterprise bean method invoker will depend on the enterprise bean's time usage. The enterprise bean should thus not exceed a defined maximum time frame, and it would hence be unwise to perform blocking method calls in the enterprise bean, unless it is known that the method would unblock within the tolerated time frame. The enterprise bean is also depending on the EJB method invoker to invoke it in time to fulfill its guarantees. There would thus be necessary to define a minimum invocation rate and a maximum invocation time frame for each enterprise bean. In that sense, there could be a need for other scheduling algorithms than the round robin algorithm.

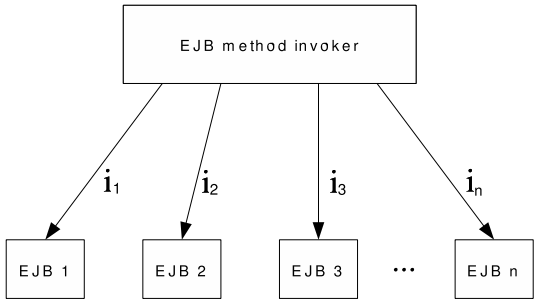


Figure 7.4: Enterprise beans invoked by an EJB invoker in the main element. The arrows represents the invocations (numbered from i_1 to i_n)

Accessing hardware / lower level services is not allowed within enterprise beans either, so all hardware/service access should be moved into the main element. If the GUI component is split, and all hardware access moved to the main element, it will not be much left for the enterprise bean if anything at all. This will also be the case for the audio components: Both the thread (or the equal functionality) and the access to the audio API must be moved into the main element. Unless there is performed some other action on the audio clips in the audio components as well (like up-/downsampling or some calculations on the audio clips), there will not be any functionality left to implement as an enterprise bean.

7.3.3 Discussion about split components

By splitting components, more of the logic is created as enterprise beans. This again results in more logic that utilize the advantages of EJB and component technology in general. But the design after a split could be more complicated to understand. If the functionality of the two split parts are weaved into each other to such a degree that the two parts still have to be inspected as one unit to understand their tasks, the design can be harder to read. If the splitting is more a matter of separating (independent) concerns, the design would not become much less comprehensible. A part of a component with simpler dependencies to its counterpart could also be more relevant for reuse than one with a complex dependency.

The success of a component splitting will depend on the enterprise bean candidate and not the candidate for the main element. Is the enterprise

bean candidate likely to be reused? Is it likely to be upgraded independently of its counterpart? Are there already existing enterprise beans that can be used? These are important questions that must be considered during the evaluation of this design method.

7.4 Summary

By introducing the possibility of moving parts of an application outside the component framework, applications with requirements that can not be offered by the EJB specification can still utilize the advantages of EJB for some of its elements. This is not in any way a new method, though, and e.g in the Windows platform many applications make use of Microsoft COM and DCOM components for some of its functionality. The performance of the communication between the enterprise beans and the main element will have to be considered, since the communication (using RMI) would have to pass through the network layers which is slower than using local bindings.

Splitting of components can be a useful method for moving more of the design to the EJB container, and thus increase the utilization of the advantages of EJB. The success of a split will depend on factors like how complex the dependency between the split parts will be, and thus how likely the EJB part is to be reused and independently replaced. It will also depend on how likely it is to find a ready component from other vendors. Splitting should be considered when it is possible to separate two quite independent concerns.

Chapter 8

Design and implementation without EJB

Contents

Chapter 7 discussed whether all the identified components in the generic design can be implemented as enterprise beans without violating the restrictions of the EJB specification. The discussion revealed that some of the components have requirements that the EJB platform can not satisfy. In chapter 9 the design and implementation of an application using restricted functionality will be presented.

This chapter will present the design and implementation of a client application, called *SimulaPhone*, where EJB is not used. Only an audio distribution server will be implemented using EJB and JMS. The process of designing and implementing this application was meant to aid the identification of requirements of the common components (presented in 5.3). In addition, this application works as a reference for the performance testing of the implementations using restricted functionality.

8.1 The design of the client

The design of the client, called *SimulaPhone*, is based on the abstract design made in chapter 5 (see figure 5.2 on page 29). To simplify the application, the redundancy, compression and echo cancelation components are left out of the implementation, and the application has no adaption mechanisms to handle resource changes. In addition, some of the components are included into others: The jitter buffer is implemented into the audio output, and the silence detection is considered as a part of the audio input. As the SimulaPhone was designed at a very early stage of this work, some of the component's names are slightly different.

SimulaPhone consists of six main classes (see figure 8.1):

- The PhoneManager is the core class, and it coordinates the network communication and the incoming audio clips from the server and from the local SoundRecorder.
- EJBManager is used by PhoneManager to manage the communication to the conference server, such as sending data using RMI, and to establish a subscription to the JMS topic (used to send the audio data from the server). Thus, the EJBManager includes the functionality of the outbound communication component. It also implements the communication setup of the inbound communication component, while PhoneManager implements the actual packet reception mechanisms of the inbound communication.
- The SoundPlayer queues (that is, incorporates the audio/jitter buffer) and plays audio clips.
- The SoundRecorder samples audio and sends it to the remote phone through the PhoneManager.
- The GUI presents the textual data to the user and handles all user interaction.
- Finally, the StarterClass (within SimulaPhone.java) initializes the GUI and the PhoneManager. It also handles the events produced by the GUI and the PhoneManager. As a part of that, it sends data between the GUI and the PhoneManager (and hence, the GUI and the PhoneManager are invisible to each other).

8.1.1 The PhoneManager

The PhoneManager has the responsibility for communication between the application and the conference server. It uses the EJBManager to

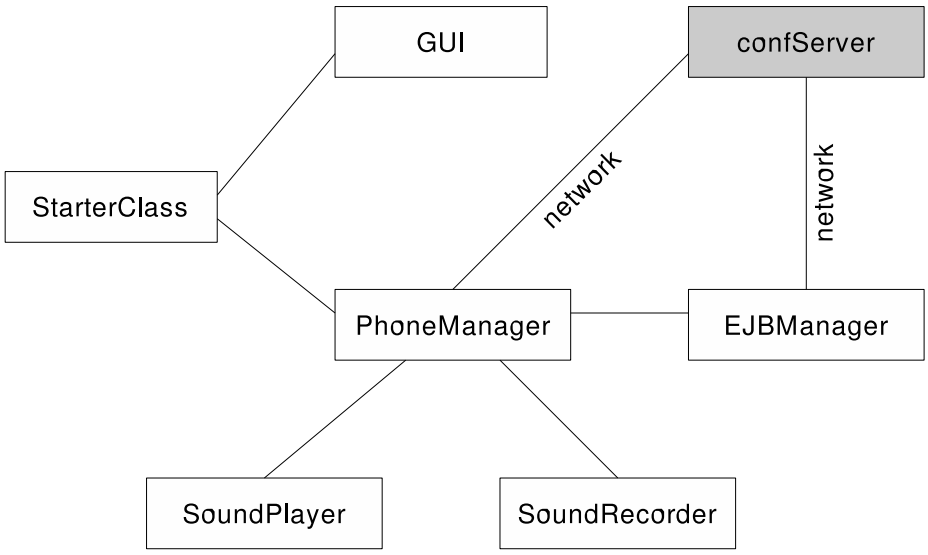


Figure 8.1: The figure shows the design of SimulaPhone. The lines represent communication between the objects. ***confServer*** *is not a part of the client, but is included to show the communication with the server.*

send data to the server. The **EJBManager** also subscribes to the JMS topic on behalf of the **PhoneManager** using a reference to the **PhoneManager** object.

To receive data, the **PhoneManager** implements the **MessageListener** interface, and its **onMessage** method is invoked every time data is received.

All data sent through the network is wrapped in an object of the **PhonePacket** class. The marshalling is done with Java's serialization interface. The interface converts any object of the serializable class from the internal data representation to a byte array. This makes the marshalling issue easier. All the different packet types are made as subclasses of the **PhonePacket** class. When a packet arrives, the **PhoneManager** sends it to the proper method according to its subclass type. Extending the protocol is thus done by creating a new subclass of the **PhonePacket** or by extending an existing subclass.

The subclasses of **PhonePacket** are:

- **MessagePacket** - Contains a written message from the user.
- **SoundClipPacket** - Contains an audio clip (part of an audio stream).

• audio clip series

- EndOfClipSeriesPacket - Tells the remote phones that this was the last clip in the series (where a *series* is defined as a continuous sequence of non-silent audio clips).

The PhoneManager has the responsibility for feeding arriving audio clips from the distribution server to the SoundPlayer for playing. When a text message arrives or a connection is established, a PhoneEvent (an event for notification to the GUI) is produced so the StarterClass object can make the proper feedback to the user.

The PhoneManager also handles the output from the SoundRecorder, and sends all audio clips to the distribution server (through EJBManager).

8.1.2 The SoundPlayer

The SoundPlayer uses Java's standard audio player features by importing the modules javax.sound.sampled.*. To play audio, a data line has to be opened. A parameter defines the audio properties; sample rate, bits per sample and mono/stereo.

To play audio, the audio data must be written to the data line. The SoundPlayer has its own thread that manages the buffer of audio clips, and writes them to the data line.

8.1.3 The SoundRecorder

For the audio access, SimulaPhone uses Java's built in audio classes. The SoundRecorder continuously records audio clips, as long as the microphone is switched on in the GUI, and it uses the SoundFilter (a silence detector) to classify whether each clip contains "audio" or just "silence". Series of audio clips are sent to the receiver through PhoneManager.

For SimulaPhone, a sample rate of 8 kHz (mono) with 8 bit per sample was initially chosen. This is by many considered to be of sufficient quality for speech. Some of the computers that were used for the implementation and for the performance measurements, did not support other audio formats than CD quality (44100 Hz, 16-bit, stereo) for sampling. The solution was to down-sample the audio with software to get a suitable format. A simple conversion with low CPU requirements down-sampled the audio to 11025 Hz, 8-bit, mono using *shift*, *and* and *or* operators. Still, for the performance measurements that were executed, a microphone emulator was used with a sample rate of 8000 Hz (see 8.4).

The programmer can define how large Java's internal recording buffer should be. When the buffer is filled with data, the recording library just re-starts from the beginning of the buffer again in a "round robin" fashion. To copy/read the samples from the internal recording buffer, the read method of the recorder's data line is called, with the destination buffer as a parameter. It is a good idea to read fewer bytes than the size of Java's internal recording buffer. If the destination buffer has the same size as the internal record buffer, Java will have to record over again what is about to be copied. If, on the other hand, fewer bytes are copied than the number of bytes in the internal buffer, Java will have enough time to copy the data before the recorder needs to use that part of the buffer again. In SimulaPhone, the size of Java's internal recording buffer is set to twice the size of a sampled audio clip.

The audio from each interval is classified as either audio or silence by the silence detector (that is, the SoundFilter). This is done with a method that finds the percentage of samples with an absolute value that is over a certain limit. If audio is detected, the audio clip is sent to the server through the PhoneManager. To allow short pauses in the speaking without "cutting off" the stream, a certain amount of silent clips is allowed before the SoundPlayer judges the speaking as over. This is set with the variable numMinClipSeries in the SoundRecorder.

When the talking has ended (that is there have been some audio clips with silence), the SoundRecorder tells the PhoneManager to send the endOfSoundClipSeries packet, which tells the receiver that there is no more audio clips in this series to wait for.

8.1.4 The GUI

The GUI (figure 8.2) is made using Swing, which has been the default GUI API in Java since version 1.3.

The GUI produces an event each time a button is clicked. The only subscriber to these events is the object StarterClass, which also presents all textual information to the user through the GUI.

8.2 The conference/distribution server

The application's server is called a conference server, but is in fact only a distribution server as it does not support merging of simultaneous streams. This means that only one user can talk at each time to avoid mixing of audio clips.

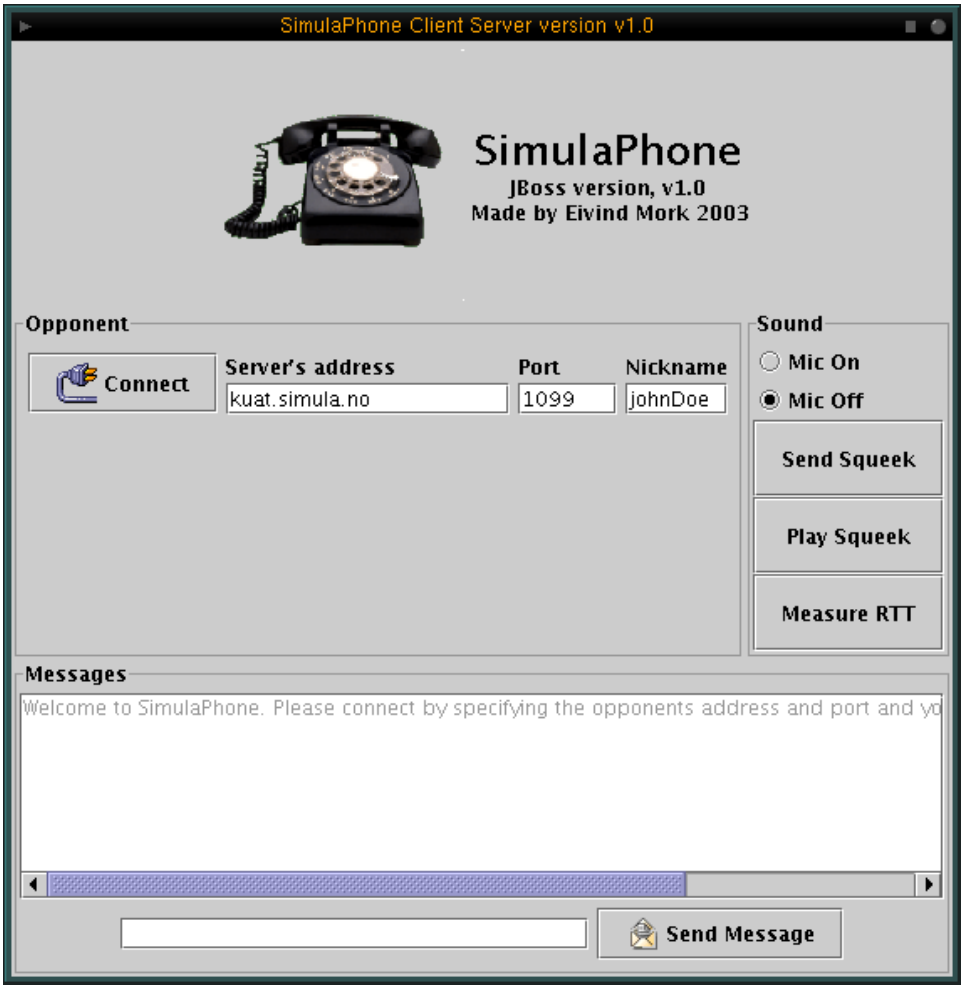


Figure 8.2: A screenshot of SimulaPhone

In this implementation the server is represented by a session bean called confServer which provides the following methods:

- public void joinConference(String nickName)
- public void publishTextMessage(String name, String msg)
- public void publishBytesMessage(String name, byte[] arr)

The clients connect to the conference by invoking the joinConference method. They also subscribe to the topic *topic/conferenceTopic*.

All data for the clients is sent through JMS. The clients do not publish messages directly to the topic, but call on the confServer's methods publishTextMessage (for text messages) and publishBytesMessage (for binary data). confServer then publishes messages to the topic on the sending client's behalf. All the clients (including the sender) then receive the messages from the topic.

Figure 8.3 on the following page shows an overview of the application model.

In the choice between *stateful* and *stateless* session beans, stateful was chosen since data had to be stored in the bean. The application uses JMS, and with the stateful bean, the connection to the JMS service could be kept open. With the stateless bean, the bean would have to recreate the connection at each invocation. Establishing a connection to JMS takes too much time to be done each time.

8.3 An example of a session

This section will discuss an example of a session to ease the understanding of how the application works. In the example there are two users, user *A* and user *B*, who want to talk to each other. At this stage both users have started SimulaPhone which is up and running.

First, both users enter the host name and port number of the conference server in the *Server's address* and *Port* fields respectively. Next, each user selects his nickname in the *Nickname* field and presses the *Connect* button which produces an event. The event is then consumed by the StarterClass object which invokes the PhoneManager's connectToHost method.

The connectToHost method then creates a new EJBManager object which connects to the JBoss server in its constructor method and creates a stub for the confServer enterprise bean. Finally, the PhoneManager invokes

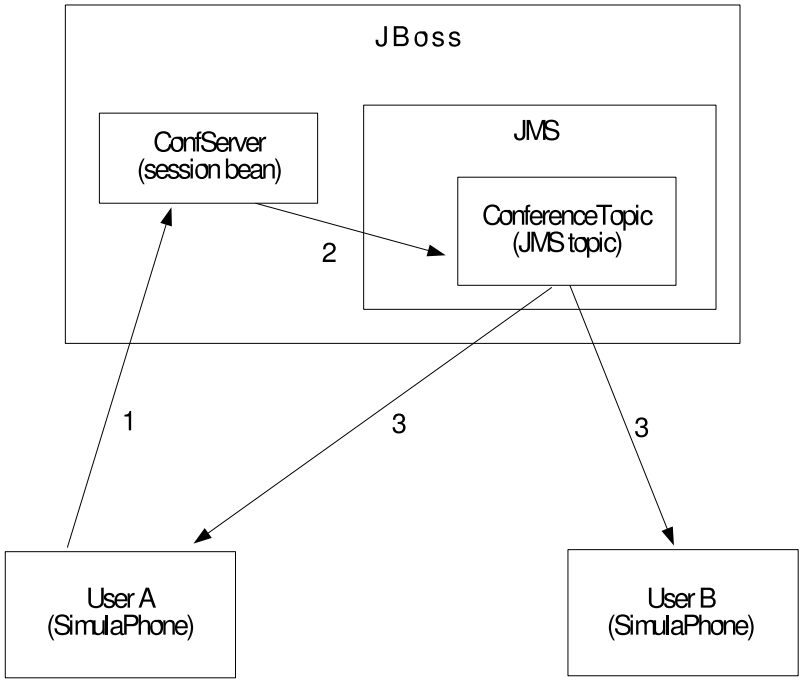


Figure 8.3: Sending audio with JBoss and JMS

EJBManager's initJMS method that subscribes to the conference topic. When this has happened at both client's SimulaPhone, the system is ready for communication.

User A now turns his microphone on (via the radio button switch), which causes the SoundRecorder thread to run. He waits a few seconds before he says something, thus all the starting audio clips are classified as silence by the SoundFilter and thus thrown away. User A then decides to say *"Hello, B"*, which causes the next few audio clips to be classified as audio (that is; not silence).

The SoundRecorder sends each audio clip to the PhoneManager, which wraps the clip in a SoundClipPacket object, serializes the audio clip and sends it to the conference server using EJBManager's sendBytesArray. The conference bean, confServer, publishes the incoming audio clip to the JMS topic.

The JMS then sends the audio clip data to all participants at the topic. This results in the onMessage method being invoked at each client.

When the PhoneManager at each client receives the message, it recognizes the PhonePacket as a SoundClipPacket, takes the audio clip out of the SoundClipPacket, and passes it to the SoundPlayer. The SoundPlayer will then continue to play until the queue is empty. Note that the audio clip will be played at *both* user A's and user B's loudspeakers. If the audio packets had been tagged with the sender's ID, it would be possible to avoid the playback at A's computer. But in a scenario where the server merges the streams, it would be impossible to leave out one's own audio anyway.

Note that the SoundPlayer starts to play as soon as the first audio clip arrives. The whole series of audio clips making the *"Hello, B"* sequence does not have to be present for the SoundPlayer to start playing. For the same reason, user A's SoundRecorder will send the first audio clip in the *"Hello, B"* series as soon as it is classified as audio, and not wait for the rest of the clips to be recorded before sending. This ensures that the audio clip will play at user B's loud speaker as soon as possible.

8.4 Microphone simulator

During the work of this thesis, the need for a microphone simulator started to reveal, and there were two reasons for this: First, it is inconvenient to talki loud into the microphone at a computer lab. And second, it is important to be able to test the exact same recording scenario in different

application scenarios. Talking in the exact same way twice is difficult, if not impossible. In the SimulaPhone application the microphone simulator is activated by pressing the *Send Squeek* button.

If a simulator was to be used, it would have to be certain that the simulator would produce data in the same way as the real audio recorder would, and that the simulated recording situation would be as close to a real recording situation as possible. The microphone simulator uses a sample rate of 8kHz in mono 8-bit, and sends audio clips with data from an audio file. The microphone simulator was used with audio clip sizes of 2000 bytes, which is equal to 250 ms of audio in this audio format.

A test application was made to measure how accurately the microphone simulator produced data. A similar test application was also made to test the real audio recorder. Although the hardware might be quite accurate, it could be small delays caused by the audio drivers etc. in the OS. For an audio clip size of 2000 bytes, both the real audio recorder and the microphone simulator should produce data every 250th ms.

To test the real recorder, 50 audio clips were sampled. This was done by calling the `read` method in the recorder's data line 50 times. After each invocation of the `read` method, the current time was stored. For each sample (except the first) it was then possible to calculate the time difference between the current time and the time for the previous sample.

In the same way, the simulator should "produce data" every 250th ms for 50 clips. Also for the microphone simulator the delay between every clip was measured. Instead of waiting (and blocking) for the data, the thread sleeps for a calculated equal time.

Ideally there should be exactly 250 milliseconds between all audio clip "productions" for both the simulator and the real recorder, but there are some small variations. For the real recorder, the largest deviation was 16 ms. The simulator's largest deviation was 11 ms. The complete result of the test is shown in figure 8.4 on the next page.

8.5 Measurements

To determine whether the JBoss server and the JMS service were efficient and reliable enough to handle the distribution of real-time audio, it was necessary to measure what the end-to-end network delay would be. The variation in delay (jitter) and possible packet loss would be looked into in this test. JBoss does not offer any real-time guarantees, and it is important to notice that all results from this test only apply to the cur-

rent version of JBoss (4.0.0 RC1). It is still believed that the results can provide some idea of JBoss's timeliness for such tasks. It is important to notice that the server only works as a relay, and if it had worked as a conference server where the concurrent streams were merged, the processing time of the merging would have to be added to the end-to-end delay.

To measure the end-to-end network delay, the time when the audio clip leaves the sender and the time when the receiver(s) receives the audio clip would have to be recorded. For the calculated end-to-end network delay to be exact, the internal clocks on the different computers would have to run synchronously. There are tools available to synchronize computer clocks so they can run with tolerable differences, but since public computers where used for this experiment and regular users do not have permission to run such system tool, they could not be used. The solution was to utilize a side effect of the conference server's design: Since the server distributes any arriving packet to all the subscribers of its topic, also the sender will receive its own packets (see figure 8.3 on page 70). The time used for this "loop" (from the sender, to the server and back again) is called the Round Trip Time (RTT). Since the sender is a sub-

• round trip time

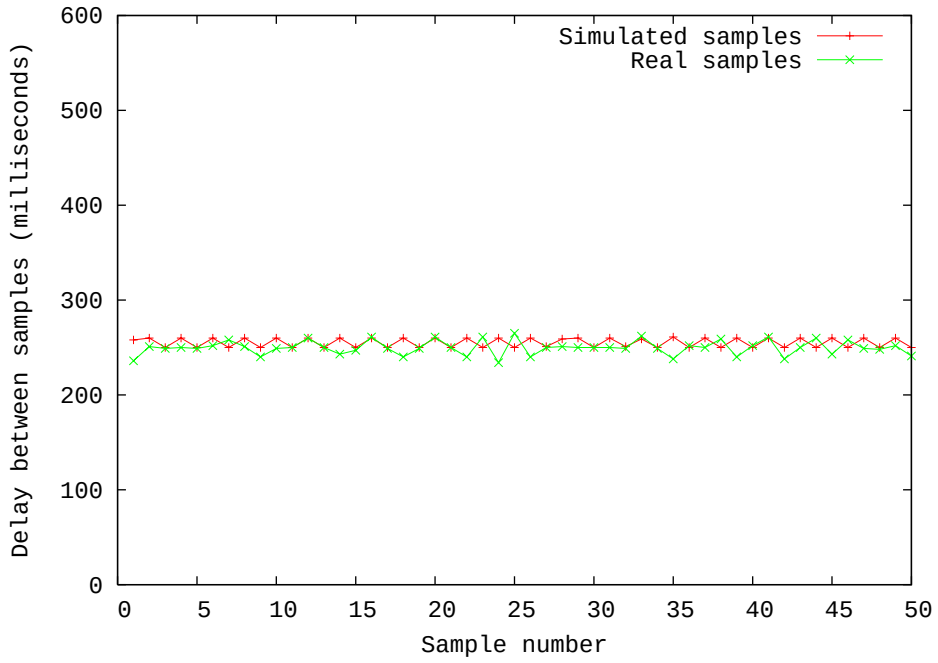


Figure 8.4: Time variations during sampling: The real recorder vs. the simulator

scriber and thus also a client, this RTT to the server would be representative for the end-to-end sending delays to the other clients.

8.5.1 How the Round Trip Time (RTT) was measured

To measure the RTTs, a new subclass of SoundClipPacket called RTTMeasurePacket was made. This class contains a variable long stamp where a time stamp with the time of departure from the sender is stored.

A class named MeasureRTT was made to perform the actual audio clip sending. It is an extended version of the microphone simulator and it sends packets with real audio samples of 250 at 8 kHz mono 8-bit. The only difference is that the microphone simulator creates SoundClipPackets, and the MeasureRTT creates RTTMeasurePackets. Real audio data had to be used for the RTT measurements (and not "empty data"), as compression could be used on the packets on their way through the network, and hence "empty data" could have changed the network speed. Before sending the RTTMeasurePackets through PhoneManager, MeasureRTT saves the time stamp in the packets.

In PhoneManager a new method named handleRTTMeasurePacket was made to handle all incoming RTTMeasurePackets. It calculates the time difference between the current time and the packet's time stamp, and logs the result.

8.5.2 Sources of error

Since the application is not running in a real-time environment, it is possible that it could be delayed by the system. If it is delayed when it is supposed to receive a returning packet, this will affect the result of that RTT. There was one known big source of errors for this test setup: The Networking File System (NFS) dæmon could sometimes block the computer for up to several seconds. This can be experienced by the user by the lack of smoothness when moving the mouse. The process of moving the mouse pointer can sometimes get so little time that the pointer completely freezes for some seconds. One way to manually cause this situation to happen, is to copy a large file to a local drive. For the initial measurements, the results of the RTTs were stored in a file on a NFS mounted drive. There were then several RTTs with a value of more than 1000 ms. When this log file was stored on a local disk instead, almost all of the high valued RTTs were gone. When performing the test presented next in this chapter, the applications were run on a computer where as few as possible other applications were run, and all records were stored

to a local disk. It is still possible though that system processes could read or write to files stored on NFS and thus cause extreme values for some of the RTT samples..

8.5.3 The test setup

Two tests were performed; one with two clients (the sender and one additional receiver) and a second with four clients (the sender and three additional receivers).

The tests were run on a local 100 Mbit network. The ping time to the server from the clients was about 0.2 ms. Five computers were involved during the tests; one for the conference server, and one for each client. They all had the same configuration:

- CPU: Intel Pentium 4, 2.8 GHz, 512 KB Cache, Hyper Threading
- RAM: 1 GB
- OS: Red Hat Enterprise Linux WS, release 3 (Taroon Update 4)

8.5.4 The results of the RTT measurements

The results of the tests are presented in figure 8.5. Each value in the graph represents the mean value of the RTTs in one sending. One sending consists of 35 audio clips (of 250 ms) from a recorded spoken sentence. In each test there were 50 sendings. There was a five seconds delay between each sending in the tests.

As stated in 4.2, the end to-end-delay should be kept under 400-600 ms. The RTTs to the server (which represents the end-to-end network delay shown as the time between *b* and *c* in figure 4.2 on page 18) together with the sample length and the jitter buffer should thus be kept under 400-600 ms. Even for the highest mean RTTs in the two tests, which are around 40 ms, this leaves a time for additional delay in a jitter buffer for up to 300 ms. This means that one audio clip can be delayed, but not two before the jitter buffer runs empty.

There were no packet loss for both the tests. When it comes to variations in delay, most audio clips arrived within 5 milliseconds. Figure 8.6 on page 77 shows histograms with the delays of the two tests. Each bar represents the number of audio clips arriving within its five millisecond interval. For the test with two clients, almost all but 14 packets arrived within 20 ms, and about one third of those within 6-10, and two thirds within 1-5 ms. For the test with four clients, all but 28 audio clips arrived

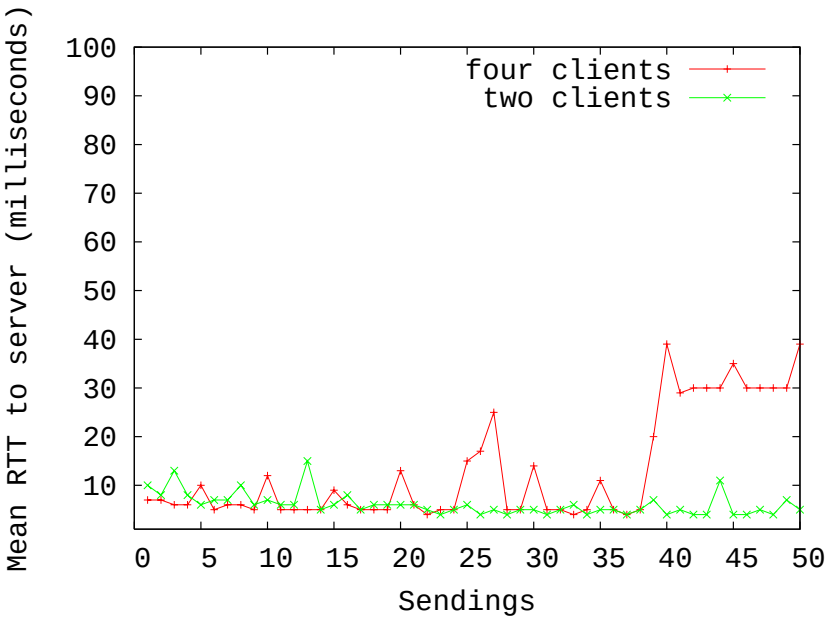
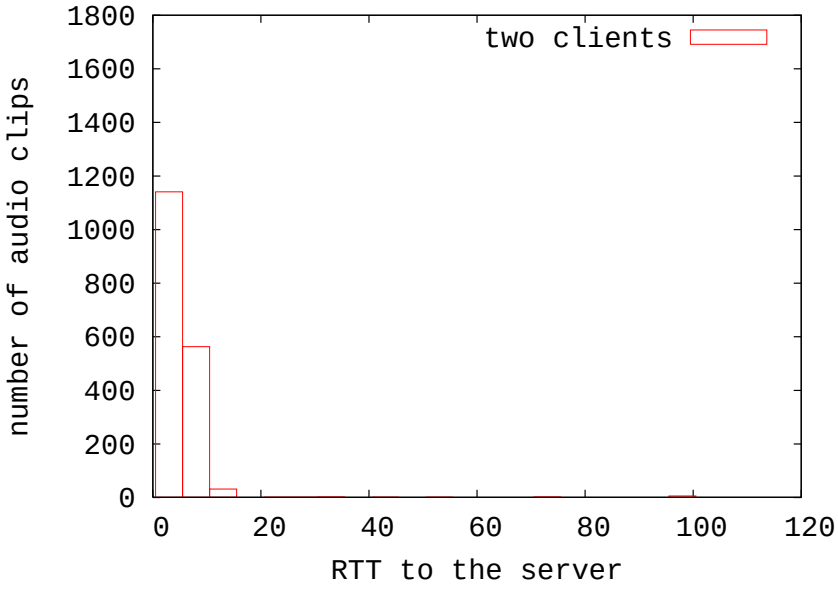


Figure 8.5: Mean RTTs to the server for two and four clients

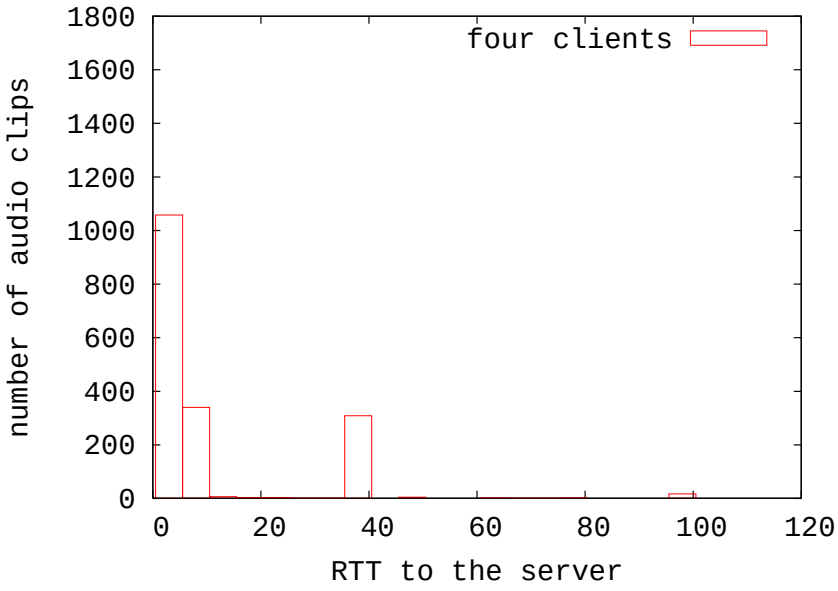
within 40 ms. Still most audio clips arrived within 10 ms, but there were about 300 that arrived between 36 and 40 ms. This is considered as a satisfying low variation in delay.

The tests showed that the JBoss server performed satisfactory for those two tests. Again, this can only apply directly to the current version of JBoss, although it can provide some idea of JBoss's timeliness for such tasks in general.

It can be discussed though, how representative this test is for other QoS sensitive applications. Audio is not in a very bandwidth demanding domain, and the conference server did not perform any action on the data. It only tests one remote method invocation on a session bean, and one distribution through JMS for each audio clip. It is thus fair to say that the results is representative for this small domain of applications only.



(a) Two clients



(b) Four clients

Figure 8.6: Histogram with the RTTs distributed on 5 ms intervals. RTTs beyond 100 is stored in the 96-100 interval

Chapter 9

A design using restricted functionality

Contents

Chapter 8 discussed an implementation of an Internet phone where only the conference (distribution) server was implemented using EJB. This chapter will investigate cases where the restrictions of the EJB standard are not respected. The design and implementation of a working application made with EJB and restricted functionality will be discussed. The purpose of this work was to get experience and ideas for extensions to the EJB platform to improve the support of real-time audio applications in EJB.

9.1 Using restricted functionality

Before designing the new application, there was a need to know in advance if the restricted functionality would work or not in the planned scenario. The easiest way to investigate this was to create some simple test beans, to see if it was possible to use these methods or not. It is important to notice that the results of the tests only can tell if the use of

restricted functionality will work for this version of JBoss (4.0.0 RC1). The tests do not say if other versions of JBoss or other platforms would give the same results.

The result of the test for the use of threads, which are used in the audio output and the audio input, showed that they could be created and activated within the EJB server. This means that the container will lose control of the resources, since an enterprise bean running in a user made thread will continue to run even though the container might want to passivate it. The user could implement functionality in the ejbPassivate method within the enterprise bean to passivate the bean, though.

The tests also showed that it was possible to create a GUI with AWT/Swing, and its event listener worked as-well. Both the audio input and audio output could access the audio cards through Java's audio interfaces as needed.

The need to share data between enterprise beans could have been solved using entity beans with BMP and empty persistence methods, but since the purpose of this implementation was to test if the restricted (but wanted) functionality worked, an other solution was chosen: Since stateful session beans have an one-to-one mapping between bean and client, two components can not access the same instance of a third component. They would get one instance each of that third component, and thus they could not share any data using stateful session beans. By creating a new class that could be instantiated as a singleton (an object that is the only instantiation of its class) this functionality could be obtained.

- singleton

A singleton in EJB can be created by defining a variable of a class as static, inside the stateful session bean. The first instance of the bean that is initialized, creates this singleton, and then they can all access the same data. Figure 9.1 shows an example of this. The enterprise beans A and B want to access the data in a stateful session bean C. When they look up bean C, they get an instance of C each. But as both the instances of C access the singleton, both A and B can access the same data through C.

The use of static variables is restricted, unless they are defined as finals (constants). Since the value of the singleton might change during runtime, the static variable can not be defined as final, and hence, this is restricted functionality.

These tests indicated that it was possible to create the application by using the restricted functionality in JBoss version 4.0.0 RC1. As the functionality is restricted, it is hard to say how well it will work on other EJB server impllementations.

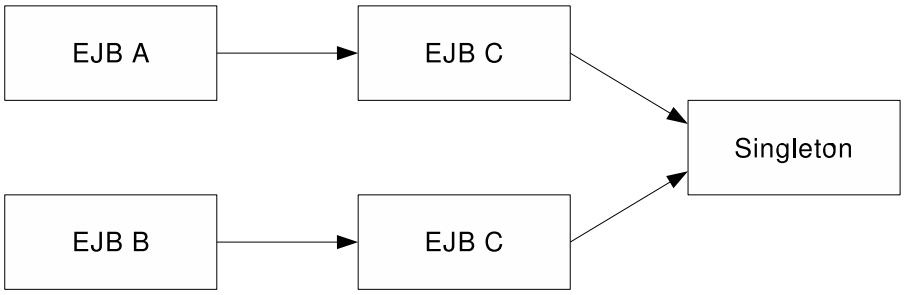


Figure 9.1: Two components access the same data in a singleton. They do this through their respective instances of a third component

9.2 The design of the application

The design of the application is very similar to the one made without enterprise beans in chapter 8 on page 63 (see figure 9.2 on page 83), but there is a couple of important differences:

- All the components must be looked up using JNDI before any connection and access can be made to the components.
- All components (except the packet reception) are made as an enterprise beans with corresponding singletons. In the case of the PhoneManagerSingleton this is the only way to do it, since both StarterClass and PacketReceptionComp have to look up PhoneManagerComp and access the same data. The other singletons are created to make a more flexible design, where all the singletons can handle multiple access in a different composition later.

In addition, some new interfaces had to be made to avoid circular dependencies between the components. For example, when compiling the PhoneManagerSingleton, the compiler (javac) needs access to the compiled classes of the SoundRecorderComp. This means that the SoundRecorderComp must be compiled before the PhoneManagerSingleton. The problem with the old model occurs when the SoundRecorder also needs the definition of the PhoneManager classes which causes a circular dependency. A circular dependency is not a problem when all the classes are compiled together, but causes a problem when classes are compiled separately, and this is the case in a component setting. To avoid this problem, an interface of an *audio packet receiver* was created in the SoundRecorder. When the SoundRecorderSingleton is created, it saves a

reference to such an audio packet receiver. If the PhoneManagerSingleton implements this interface and passes its own reference to the SoundRecorderSingleton, the problem with circular dependencies is solved.

All the components are made as session beans with one exception. The packetReceptionComp is a Message Driven Bean (MDB). Its onMessage method gets invoked by the JBoss server each time a message is sent in the topic the MDB is subscribing. The MDB's variables are not preserved between the invocations, so the PacketReceptionComp has to look up the PhoneManagerComp through the JNDI server each time a packet arrives.

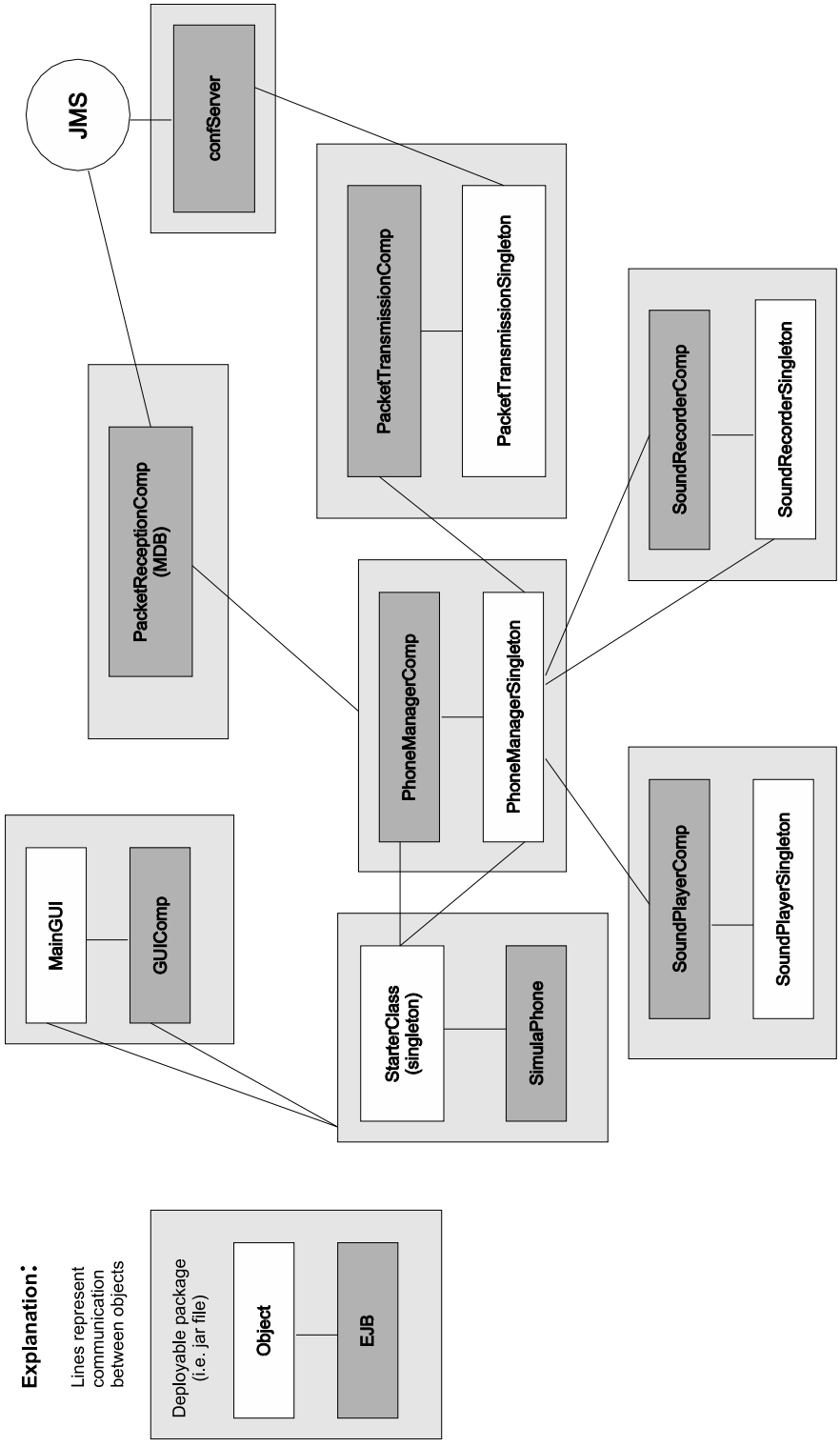


Figure 9.2: The design of the application made with EJB (MDB version)

9.2.1 Instantiation of the components

To start the application, a client has to look up the enterprise bean SimulaPhoneComp, and invoke its startSimulaPhone method. SimulaPhoneComp then creates a StarterClass object, which again looks up and gets instances of the GUIComp and PhoneManagerComp enterprise beans. The StarterClass is registered as an event listener at both the MainGUI and the PhoneManagerSingleton objects.

In the PhoneManagerComp's init method, the PhoneManagerSingleton is created, which again looks up and the SoundPlayerComp, SoundRecorderComp and PacketTransmissionComp enterprise beans, and invokes their init methods. They again creates their respective singletons.

The SoundRecorderSingleton gets a reference to the PhoneMangerSingleton as a parameter to its constructor.

When all this is done, the application is up and running.

9.3 RTT measurements

The test setup for this version of the application was the same as for the version without EJB in chapter 8. The tests were performed with two clients. It was expected that this version would have approximately the same performance as the EJB-less version, since the use of local interfaces basically sets up direct object references without the use of stubs and skeletons.

The mean values of the RTTs (figure 9.3) shows about the same results as for the version without EJB, except for one extreme mean value where one of the underlying RTTs was over 2000 ms. The histogram in figure 9.4 shows that all but 45 packets arrived within the 6-10 ms interval. In the version without EJB, about two thirds of the packets arrived within the 0-5 ms interval. The relative difference is therefore significant, but in a matter of milliseconds, the difference is small.

9.4 A version without MDBs

Except for the initialization of the EJB version of the application, the differences are small between the old and the new version. When the applications are up and running, they should be working fairly the same, since the local home interface, and not the remote home interface, was

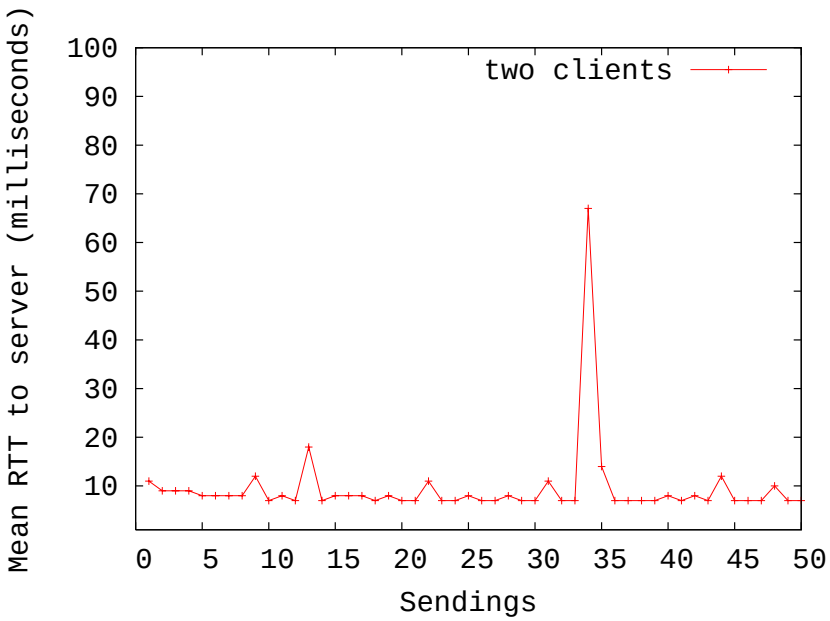


Figure 9.3: Mean RTTs to the server with two clients (with MDB) for 50 sendings

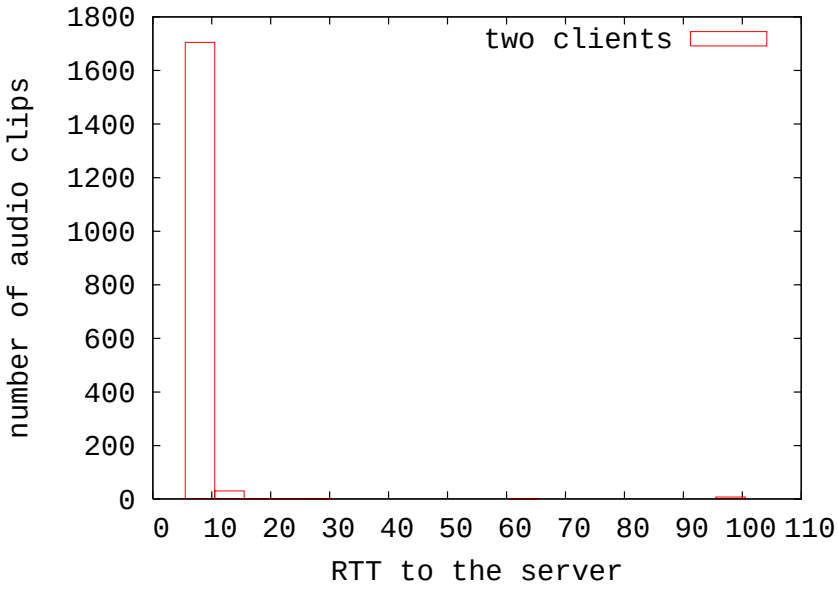


Figure 9.4: Histogram with the RTTs distributed on 5 ms intervals. RTTs beyond 100 is stored in the 96-100 interval

used for the new version. There is one exception, though: The PacketReceptionComp in the EJB version has to make a JNDI lookup for the PhoneMangerComp each time a packet arrives through the JMS topic. This is because MDBs can not store any variables (including the result of a JNDI lookup) between invocations. It was not known how effective this was, and therefore a third version of the application was made. In this version, the PacketReceptionComp (that so far had been made as a MDB) was substituted with a session bean with a corresponding singleton. The singleton establishes a connection to the JMS service and subscribes for the current topic. It also has a reference to the PhoneManagerSingleton, so no JNDI lookup is needed each time a packet arrives.

The results of this test is shown in figure 9.5 and 9.6. It is a little surprising that this version performs a bit worse than the version without a MDB for packet reception. Except for the elimination of the repeated JNDI lookups (used in the MDB in the previous version), the two EJB implementations should be about the same. In the MDB-less version, more of the packets arrived in the 36-40 ms interval. No good explanation for this was found, but it is possible that the drop in performance could have been caused by, or at least been under the influence of, larger CPU consumption of other background services. It is still fair to say, though, that the current version of JBoss performed satisfactory for all three versions of the Internet phone application.

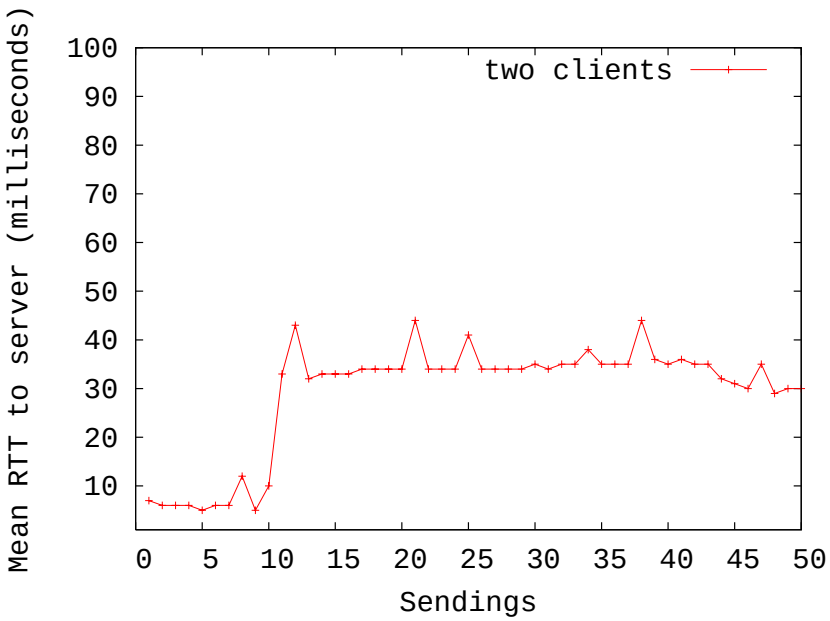


Figure 9.5: Mean RTTs to the server with two clients (without MDB) for 50 sendings

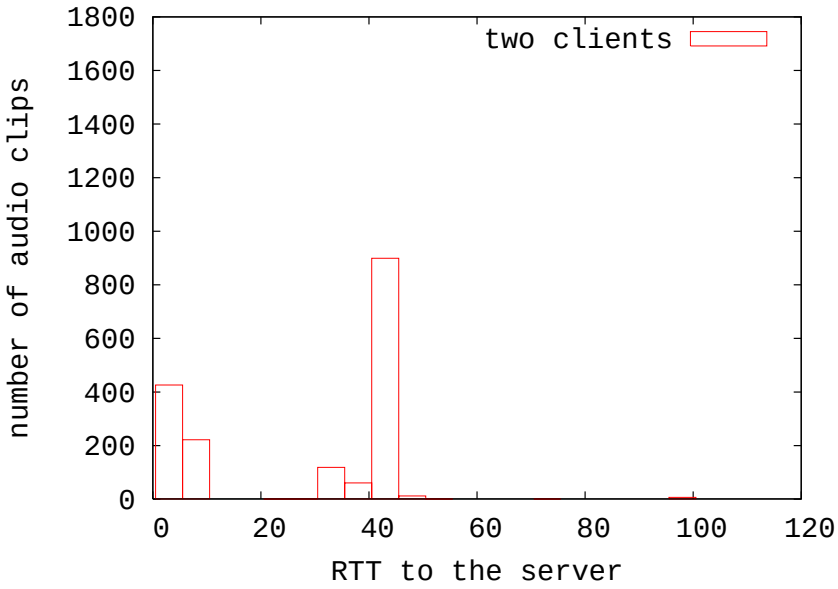


Figure 9.6: Histogram with the RTTs distributed on 5 ms intervals. RTTs beyond 100 is stored in the 96-100 interval

Chapter 10

Extensions to the EJB specification

Contents

First in this chapter, there will be a discussion about the required functionality that is missing in the current EJB specification. Next, and based on the results from the previous discussion, there will be suggested changes and extensions to the specification that will improve the support of QSAs. The suggestions are divided into two sections: The first section will discuss general extensions that are needed in order to create QSAs using EJB. These are basic extensions that are required for an Internet phone application to run at all. The next section will discuss QoS functionality that will improve the performance of such applications.

10.1 Discussion of required functionality

The discussion is separated into two parts: First the results from the discussion about creating applications without violating the EJB specification will be addressed. Next, the results from the implementations using restricted functionality will be discussed.

10.1.1 Creating applications following the EJB standard

The discoveries in 6.6 and the discussion in 7.1 show that an Internet phone application can not be entirely built using enterprise beans. Some simple components that do not run actively nor are accessing hardware or lower level services can be built (like the silence detection component, and components for compression), but there are others that can not.

The missing ability to create and manage threads makes it difficult to create components like the audio output and the audio input. The EJB Timer Service could in some cases replace the need for threads in an application, but since the service is not supposed to work with small intervals (rather minutes, hours, or even days), it is not suitable to replace the use of threads in a real-time audio application.

Accessing hardware or lower level services from an enterprise bean is a second and major problem. Some types of access to hardware / lower level service are made possible through dedicated services within the J2EE framework. This is for example the case for databases (using JDBC) and some kinds of network access (using RMI/IIOP and JMS). Other types of hardware/service access such as the ones needed for user interaction, like keyboard and mouse and GUI, can not be accessed. Neither file access nor the creation of network sockets is allowed by the EJB standard. There are no dedicated services that could compensate for this missing functionality either, and hence; components with user interaction or components accessing files can not be created as enterprise beans. Other types of hardware/service access is not mentioned directly, but since the arguments against GUI and file access are valid for most types of hardware/service access, it is reasonable to believe that other kinds of hardware/service access would not be allowed if Sun Microsystems knew that developers planned to do so. Hence, access to hardware and lower level services in general could be considered illegal.

10.1.2 Violating the EJB standard

The implementation using restricted functionality (in chapter 9) shows that it is possible to create an Internet phone application with the current version of JBoss (4.4.0 RC1), when the restrictions are not respected. All components that could not be made following the specification were implemented using restricted functionality, and the application worked as a whole. Violating the specification could have negative effects for the server, and the application could fail, though. This will be discussed for each used restriction in the next paragraphs.

In the application, two enterprise beans had to access data in the same instance of an other enterprise bean. The EJB standard does not support this, since stateful session beans have a one-to-one relationship to their clients. The use of singletons that were accessed using a static variable in the session beans (another restricted functionality) solved this problem. Proper use of the *synchronized* statement would eliminate race conditions in the singletons. Use of static variables in a multi JVM (i.e. cluster) environment could cause the application to fail, though. The result of an access to a static variable would depend on which JVM the enterprise bean was executed in. There would be one static variable for each JVM, not one for all JVMs, which would be needed in order for the application to run safely.

Access to hardware and lower level services (like the audio card) was also needed. Like the static variables, this would be a problem in a multi JVM environment where it would not be known in advance which computer the enterprise bean was executed in. The use of a cluster is not a probable environment for an Internet phone application, though, and in a scenario where all enterprise beans run in a single JVM, both access to static variables and hardware / lower level services should be safe. With the introduction of local interfaces, Sun Microsystems has made it possible to create applications following the EJB specification that can not run in a cluster environment. The arguments for that e.g. static variables will not work in a multi JVM environment, would also apply for the use of local interfaces. As long as there is a premise that all enterprise beans should run in the same JVM, the use of static variables and access to hardware / lower level services should not be a problem.

Application level management of threads makes it impossible for the EJB server to be in control of the CPU resources. It can no longer set the priority to all threads and decide how many that should run at a given time. This can affect the response time of its services. The server can neither manage load balancing between the JVMs properly in a multi JVM environment if user-made threads were spawned.

If applications that needs functionality like threads should be made using enterprise beans, the lost control of CPU resources by the server would have to be accepted. If not, alternative functionality that could solve this problem would have to be made. An example of such functionality is a timer service in the container that is calling registered methods periodically. But in which degree the server can control the CPU resources in any case can be discussed, though. The JVM runs as a regular application in the OS and shares the CPU time with other processes. Unless both the OS and the JVM have real-time functionality, the EJB server can never offer any hard guarantees anyway, with or without control of the threads

within the JVM.

10.2 General extensions

Application level management of threads makes it impossible for the EJB server to be in control of the resources. If an Internet phone application should be made using EJB, this would have to be accepted, unless sufficient alternative functionality is added to the standard. A timer service could be such an alternative. The EJB standard has a timer service; the EJB Timer Service, but it does not offer a service with small intervals and high precision, and can therefore not be used. By changing the quality of this service so it can offer short intervals with better precision, and access to the CPU as needed by the clients, it could eliminate the need for threads.

As seen in 10.1.1, hardware access or access to lower level services is another problem. As with local interfaces, a solution could be to accept direct access to hardware or lower level services in a single JVM environment. The alternative would be to create new services. Just as JDBC gives access to databases, there new services could be made for access to other kinds of services, like the audio card interface, the network interface (to create sockets) and the X server (or equivalent display access interfaces). In JBoss, new services can be made using JMX, and the service could be made as a MBean and get plugged into the MBean server in JBoss. Such a service can be deployed into any JBoss installation.

During the design phase of this thesis work, the need for two or more enterprise beans being able to access the same (third) enterprise bean was revealed. This was solved using a static variable in the session beans to create a singleton. Although this would work in many cases, it would be a better solution to make it possible to create the enterprise beans themselves as singletons. A third (and probably the best) solution would be to make it possible to look up one particular instance of an enterprise bean, so it can be accessed by more than one client. The enterprise bean could be given a name as a parameter upon its creation, which later could be looked up by others, using JNDI. This solution would also work in a cluster scenario, even though the server no longer could manage load balancing in the same degree.

The lookup of a component is performed using JNDI. The parameter for this lookup is the name of the component. In the implementations in this thesis, the name and code for this lookup have to be decided at compile time. A much better approach would be if the EJB specification suppor-

ted attribute-based programming, where the composer of an application could define the name of the wanted component at deployment time and not compile time. This will greatly improve the portability of a component [10]. In EJB, such attributes could be defined in an XML file like the *ejb-jar.xml*.

10.3 Extensions for QoS support

As stated in the previous section, some components require periodic execution. This could be solved using application level managed threads or an EJB Timer service. But with both these solutions, the component can not be certain that it will get enough CPU time, and that it will get it in time, since the JVM also serves other services, and the resource is limited. In other words, there should be a way for the component to reserve execution time, with assurance of getting it within a defined time frame. A natural place to negotiate for CPU reservation would be at the container since the container, controls the component.

An implementation/configuration of an Internet phone application would have requirements for the network communication, like end-to-end delay, jitter and bandwidth. If underlying network services with QoS support (like Diffserv and Intserv) are used, the network communication components would have to negotiate resource reservation with these services. In different environments, the underlying services (and resource negotiation services) may be different, and this can make the portability of the components difficult and complex. If the container, on the other hand, can act as a resource reservation broker on behalf of its components, the process of creating portable components will be much simpler [10]. In other words, in addition to offering resource reservation to server controlled mechanisms (like CPU reservation), the EJB server should offer a standardized reservation mechanism for external services as well, like in the case of network related reservations.

In the case where there are no underlying network services with QoS support, a configuration of an application can cease to function adequately, as a response to sudden drops in network performance. In this case, the application would have to reconfigure to perform as required. Examples of such reconfigurations would be a replacement of the audio compression codec, or changes to the audio format. Since monitoring of network performance is a regular task for QSAs, it would be natural to include this functionality into the EJB server.

Reconfiguration of an application as a response to an under-performing

situation could be managed by the application itself (after notification of the QoS monitor), or by the component framework. A framework managed reconfiguration would simplify the programming of QSAs, but this is a very demanding task since different types of applications can have very different requirements. The QuA project solves this by introducing service planners that can be custom built for each type of application [7]. The task of assembling the application and adapting to changes in the present resources, will be executed by the service planner. The introduction of service planners to the EJB specification could greatly improve the support for QSAs, but would require a major re-write of the specification.

To summarize the discussion in this section: In order to support QSAs, EJB needs to support resource management. That is; reservation and access control must be present for resources like CPU, memory and network. A fine granulated timer service will not solve the CPU access problem alone, since it can not guarantee access to the CPU when the component actually needs it. In order to offer a timer service with guarantees, services for CPU reservation and CPU access control must be offered as well.

Chapter 11

Conclusions and further work

Contents

11.1 Conclusions

This thesis work has looked at how QSAs, with an Internet phone as a case, can be implemented using EJB. Common components for an abstract architecture of an Internet phone application have been identified by inspecting existing applications. The result of the inspection of these common components shows that this type of application can not be fully implemented using enterprise beans, if the restrictions of the EJB specification are to be respected. Only some of the components can be made as enterprise beans without violating the specification.

In particular, three things were causing problems:

- The need for access to hardware and lower level services
- The need for threads or a reliable timer service
- The need for several clients to access one instance of an enterprise bean

In the case where the restrictions of the standard were not respected, a fully functional Internet phone application was made. To enable the general use of the EJB platform for such applications, the restrictions

made to avoid problems in a multi JVM environment should not be valid in a single JVM environment, just as local interfaces only work in a single JVM.

Some issues could also have been solved if new services were introduced. Services for access to audio cards and the X server (or equivalent display access interfaces) would eliminate most of the hardware and low level service issues for this application. An EJB Timer Service with better guarantees for quality of its service could have eliminated most needs for using threads.

The possibility to lookup particular instances of an enterprise bean would eliminate the need for static variables in enterprise beans and singletons. This solution would also work in a multi JVM environment, but would cause the task of load balancing between the JVMs to become more difficult.

In general, the developers of the EJB architecture seem very focused on ensuring that the server will always work in a cluster environment. A cluster is not in the present time the most likely runtime environment for an Internet phone application. If the restrictions were differentiated depending on whether the enterprise bean was meant to run on a cluster or not, many of the problems would have been solved.

In addition to lack basic mechanisms as described in this section, the common components had QoS requirements like CPU time and network qualities (like bandwidth, end-to-end delays and jitter). The EJB server would better support QSAs if general mechanisms for resource reservations (that hide the underlying protocols or mechanisms), and performance monitoring were supported. This would also make the applications more portable. It would be beneficial for the programmer to have advanced QoS handling and re-configuration mechanisms like the service planner in the QuA project, but this would require a major re-write of the EJB specification.

The performance of the JBoss server was tested with three different implementations by measuring the round trip times of the audio packets. The first implementation where only the server was made using EJB, showed that the server and the JMS service were fast enough to work as a relay, and the jitter was low. This was hardly surprising, though, since audio data processing should be an easy match to a modern computer. But the results were interesting for comparison with the results of the other implementations of the application.

The two implementations where all parts of the client was made of enterprise beans, performed a little worse than the client without enterprise beans. In a matter of milliseconds the difference is not very big, but the

relative difference was significant. A little surprising, the removal of the MDB (used for packet reception), and thus also the elimination of repeated JNDI lookups, caused a big (percentual) drop in performance. Since the system was not a real-time system, and the increase in number of milliseconds was not very big, it is possible that the drop in performance could have been caused by, or at least been under the influence of, larger CPU consumption of other background services.

All three of the applications performed well enough to work as an Internet phone. Since audio normally should be an easy match to a modern computer, it is still hard to conclude that the performance of JBoss is satisfactory for all real-time multimedia applications. But the test do not show that JBoss fails either. A test with more demanding requirements would be necessary to make any firmer conclusions about JBoss's performance.

11.2 Further work

Chapter 10 suggests changes to the EJB specification and the creation of new services. The design and implementation of the new services and changes to the EJB standard was beyond the scope of this work, but would better answer if it is possible and beneficial to add these changes and additions.

This thesis work has not looked very thoroughly at resource reservations, QoS monitoring and run-time re-configuration. These mechanisms are central for applications where the resources are changing during the runtime. A more thorough investigation of these subjects would thus be a valuable contribution to the evaluation of EJB's future as a platform for QSAs.

List of Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

AWT Advanced Windowing Toolkit

BMP Bean Managed Persistence

CBSE Component Based Software Engeneering

CCM CORBA Component Model

CMP Container Managed Persistence

COM Component Object Model

CORBA Common Object Request Broker Architecture

COTS common-off-the-shelf

DBMS Database Management System

DCOM Distributed Component Object Model

DES Data Encryption Standard

DNS Domain Name System

EJB Enterprise Java Beans

FIFO first in - first out

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol with SSL

IDL Interface Definition Language

IEEE Institute of Electrical & Electronics Engineers

IETF Internet Engeneering Task Force

IIOP Internet Inter-Orb Protocol

IP Internet Protocol

ISDN Integrated Services Digital Network

ITU International Telecommunication Union

J2EE Java 2 Enterprise Edition

J2ME Java 2 Micro Edition

J2SE Java 2 Standard Edition

JAR Java Archive

JDBC Java Database Connectivity

JMS Java Messaging Service

JMX Java Management Extensions

JNDI Java Naming and Directory Interface

JRMP Java Remote Method Protocol

JSP Java Server Pages

JTA Java Transaction API

JVM Java Virtual Machine

LAN Local Area Network

LDAP Lightweight Directory Access Protocol

MBean Managed Bean

MC Multipoint Controller

MCU Multipoint Control Unit

MDB Message Driven Beans

MOS Mean Opinion Score

MP Multipoint Processor

NFS Networking File System

OMG Object Management Group

ORB Object Request Broker

PCM Pulse-Code Modulation

QSA QoS Sensitive Application

QoS Quality of Service

QuA QoS-Aware Component Architecture

RMI Remote Method invocation

RTCP Real-time control protocol

RTP Real-time protocol

RTT Round Trip Time

SAP Session Announcement Protocol

SDP Session Description Protocol

SIP Session Initiation Protocol

SQL Structured Query Language (database query lanquage)

SSL Secure Socket Layer

TCP Transmission Control Protocol

TOAST Toolkit for Open Adaptive Streaming Technologies

UDP User Datagram Protocol

XML Extensible Markup Language

List of Definitions

List of Figures

Bibliography

[1] Richard Staehli, Frank Eliassen. QuA: A QoS-Aware Component Architecture. Technical Report Simula 2002-12, Simula Research Laboratory, 2002.

[2] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley, Reading, Mass., 1998.

[3] Richard Staehli, Frank Eliassen, Gordon Blair, Jan Øyvind Aagedal. QoS-Driven Service Planning in an Open Component Architecture. Work in Progress paper submitted to Middleware 2003, 2003.

[4] J. Tárraga, V. Messerli, O. Figueiredo, B. Gennart, and R. D. Hersch. Computer-aided parallelization of continuous media applications: the 4d beating heart slice server. In *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*, pages 431–441. ACM Press, 1999.

[5] J2ee v1.3 glossary. `http://java.sun.com/j2ee/1.3/docs/glossary.html` (Last accessed 2005.05.04).

[6] Jboss faq. `http://www.jboss.org/modules/html/faq.pdf` (Accessed november 2004).

[7] Frank Eliassen and Richard Staehli and Gordon Blair and Jan Oyvind Aagedal. QuA: Building with Reusable QoS-Aware Components. In *OOPSLA '04 Conference Companion*, 2004.

[8] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, f. Costa, h. Duran-Limon, T. Fitzpatrick, L. Johnson, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implemenetation of open orb 2. *IEEE distributed systems online*, 2(6), 2001.

[9] Tom Fitzpatrick, Julian J. Gallop, Gordon S. Blair, Christopher Cooper, Geoff Coulson, David A. Duce, and Ian J. Johnson. Design and application of toast: An adaptive distributed multimedia middleware platform. In *IDMS '01: Proceedings of the 8th Interna-*

tional Workshop on Interactive Distributed Multimedia Systems, pages 111–123, London, UK, 2001. Springer-Verlag.

[10] M.A. de Miguel., J.F. Ruiz, and M. Garcia. Qos-aware component frameworks. In *Proceedings of the Tenth IEEE/IFIP International Workshop on Quality of Service, (IWQoS 2002)*, pages 161 – 169, 2002.

[11] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15:109–126, 2002.

[12] Arch C. Luther. *Principles of Digital Audio and Video*. Artech House, Boston, London, 1997.

[13] H. Afifi S. Mohamed, F. Cervantes-Perez. Audio quality assessment in packet networks: an "inter-subjective" neural network model. In *Proceedings. 15th International Conference on Information Networking*, pages p. 579–586. INRIA/IRISA, Instituto Tecnologico Autonomo de Mexico (ITAM), 2001.

[14] Nashwa Abdel-Baki, Bernd Aumann, and Hans Petter Grossmann. Analyzing multimedia streaming in a dsitributed environment. *Universal Multiservice Networks, 2002. ECUMN 2002. 2nd European Conference*, 2002.

[15] Vicky Hardman Martina Sasse Mark Handley Anna Watson. Reliable audio for use over the internet. *Proceedings of the INET'95*, 1995.

[16] Randy H. Katz Chen-Nee Chuah. Characterizing packet audio streams from internet multimedia applications. *Communications, 2002. ICC 2002. IEEE International Conference*, 2002.

[17] Andres Vega-Garcia Jean-Chrysostome Bolot. Control mechanisms for packet audio in the internet. *Proceedings of the IEEE Infocom '96, San Fransisco, CA*, pages pp. 232–239, 1996.

[18] Fred Halsall. *Multimedia Communications*, pages 256–262. Addison-Wesley, 2001.

[19] C. Schlatter. Basic architecture of h.323, the swiss education & research network. `http://www.switch.ch/vconf/ws2003/h323_basics_handout.pdf` (Last accessed 2005.05.04), 2003.

[20] M. Handley et al. Sip: Session initiation protocol. RFC 2543, 1999.

[21] H.L. Goh K.K. Tan. Session initiation protocol. In *Industrial Technology, 2002. IEEE ICIT '02*, pages p1310 – 1314 vol.2, 2002.

[22] J. Rosenberg H. Schulzrinne. The session initiation protocol: Internet-centric signaling. *Communications Magazine, IEEE*, (Volume 38, Issue 10):p134 – 141, 2000.

[23] J. Huang A. P. Black J. Walpole and C. Pu. Infopipes – an abstraction for information flow. *In ECOOP 2001 Workshop on The Next 700 Distributed Object Systems, June 2001. Also available as OGI technical report CSE-01-007.*

[24] `http://java.sun.com/products/ejb/2.0.html` (Last accessed 2005.05.04).

[25] Java management extensions (jmx) technology overview. `http://java.sun.com/j2se/1.5.0/docs/guide/jmx/overview/JMXoverviewTOC.html` (Last accessed 2005.05.04).

[26] `http://www.omg.org/cgi-bin/apps/doc?formal/02-06-65.pdf` (Last accessed 2005.05.04).

[27] Bill Burke and Sacha Labourey. Jboss 3.2 workbook for enterprise javabeans, 3rd. edition. `http://www.oreilly.com/catalog/entjbeans3/workbooks/` (Last accessed 2005.05.04).

[28] Akhil Nigam Waqar Ali. java.net.rtp. `http://www.cs.columbia.edu/~hgs/teaching/ais/1998/projects/java_rtp/report.html` (Last accessed 2005.05.04).