Technologies in Self Provisioning Applications

Master of Science Thesis Project

Anita Duplancic & Katarina Lindberg

January 1998

Abstract

This master thesis concerns the technologies of Self Provisioning Applications in the context of distributed computing. Our investigation focuses on what distributed object communication technology to use in homogenous and heterogeneous environment respectively. This thesis consists of an evaluation of two modern communication technologies, Common Object Request Broker Architecture (CORBA) and Remote Method Invocation (RMI). The focus of the evaluation is from a programmer's perspective. To investigate this, we have studied literature, made interviews and analyzed frequently asked question on the Internet. We have also developed a prototype for a Self Provisioning Application using Java and Java DataBase Connectivity (JDBC). One part of the prototype was implemented using CORBA and the other part using RMI. We consider CORBA and RMI to be complementing rather than competing technologies. What technology to use depends on what environment the system will inter-operate within and the systems complexity.

Supervisors:

Henrik Fagrell (Deparment of Informatics, Göteborg)
Stefan Åberg (Ericsson Telecom AB, Mölndal)

Teknologier i

Self Provisioning Applikationer

Abstrakt

Den här magisteruppsatsen behandlar teknologier i Self Provisioning applikationer i distribuerade sammanhang. Vår forskning har fokus på vilken distribuerad objektkommunikationsteknologi som bör användas i homogena respektive heterogena miljöer. Uppsatsen består av en utvärdering av två moderna kommunikationsteknologier, Common Object Request Broker Architecture (CORBA) och Remote Method Invocation (RMI). Fokus för utvärderingen är från en programmerares perspektiv. För att undersöka detta gjorde vi litteraturstudier, intervjuer och analyserade frekventa frågor på Internet. Vi utvecklade även en Self Provisioning prototyp med hjälp av Java och JDBC. Den ena delen av prototypen implementerades med CORBA och den andra med RMI. Vi anser att CORBA och RMI är kompletterande snarare än konkurrerande kommunikationsteknologier. Vilken teknologi som bör användas beror på vilken miljö systemet ska agera i och systemets komplexitet.

*We would like to thank **Stefan Åberg**, our instructor at Ericsson Telecom AB. He initiated this master thesis project and has given a helping hand through our investigation. Most of all we would like to bless him for all technical support and feedback, and for being extremely patient.*

***Henrik Fagrell**, our supervisor at the department of Informatics, has also been great. With his enthusiasm, exuberant fantasy and energy he has encouraged us to accomplish this master thesis.*

*We are also grateful to the **interview persons** of this investigation.*

*Finally, thank you, **everyone** that gave us some advice or just popped by for chatting (and perhaps a cup of hot Java problems...) in the "Corridor" at Ericsson last summer.*

January 30, 1998

Table of Contents

1 Introduction

One of the major factors that is contributing to the rapid growth of the Internet as a communications medium, is the promise of Electronic Commerce[1]. Most people consider electronic commerce as the process of arranging transfer of goods or services, including arranging or performing payment and exchanging customer information. In this thesis, we use the term Self Provisioning Application, meaning a distributed user controlled on-line application.

One of today's greatest problems is the heterogeneous environment that commercial systems interoperate within. The diversity of software, platforms and operating systems is an issue of consideration for most application developers today.

1.1 Purpose

The purpose of this master thesis is to present a proposal to Self Provisioning application developers that intend to make a decision on what distributed object communication technology to implement. The question at issue that this thesis deals with is as follows:

What distributed object technology is the most suitable in the context of distributed object communication?

The main focus is to cover the issues of usability of two of today's most discussed communication technologies, RMI and CORBA. This investigation is made from a programmer's point of view. It concerns what technology is the easiest to learn and to use and what the differences in programming are, as this is an important question when it comes to the programmers productivity.

We discuss areas like functional differences between RMI and CORBA and emphasize advantages and disadvantages. We also discuss detected problems related to programming and environmental requirements and constraints.

1.2 Background

Ericsson Telecom AB in Mölndal has developed an application called Service Configuration. The application provides customers with services like Internet access, email accounts and so on. Ericsson's customers also use another Ericsson developed application, called the miniCustomerCare system. Through this application, operators can register and activate the ordered services at the Service Configuration application.

To reduce their customers' costs, Ericsson Telecom AB is examining the possibilities of developing a Self Provisioning Application. This application is intended to act as an on-line service available for any one that wants to order a service without any inference of a physical person. The requirements for the application is:

- It must be easy to use.
- It must be possible to run in any computer environment, this means different operating systems and/or different platforms.
- It should run within a web-browser.

[1] http://www.smartsec.se

- It should be written in Java.
- The architecture must be very open and follow a three tier design.
- The communication between the server and the client must be implemented using either Java RMI or CORBA.
- It must be easy to program (develop).

1.3 Disposition

The structure of this master thesis is as follows:

In section 2, **”Method”**, we discuss what different research methods we used to gather information about the problem area.

In section 3, **”Distributed Object Systems”**, we explain some keywords in distributed object technology and describe what distributed object systems are. Furthermore, we give a brief description of the features in the programming language Java. Finally in this section, we present an overview of the core topic of this master thesis, Remote Method Invocation and Common Object Request Broker Architecture.

In section 4, **”The Self Provisioning Prototype”**, we describe the development process when building the Self Provisioning prototype.

Section 5, **”Results”**, contains a fundamental comparison between CORBA and RMI gained from the literature we studied. We also give an overview of the interview answers and frequently asked questions about RMI and CORBA. The most important part of this section is presented at the end and covers our own experience while programming the Self Provisioning prototype.

Finally, in section 6, **”Discussion”**, we present our own thoughts about the central topics in this thesis and we also give some advice to Self Provisioning Application developers.

Appendix 1 and **2** cover the RMI and CORBA development processes.

Appendix 3 contains the prototype specification.

The interview questions can be found in **Appendix 4**.

2 Method

2.1 The Crossroad Metaphor

While examining the problem area, we used a method that we call the “Crossroad Metaphor”. We coined this concept by our self during this master thesis project. The main reason for inventing a new term was because of the lack of concepts that represent all four qualitative methods that we used. We believe that analyzing data based on the combination of subjective and objective approaches produce more reliable investigation results.

The Crossroad Metaphor approach is the perfect fusion of four “roads” namely literature studies, practical experience, interviews and observations, see figure 1.

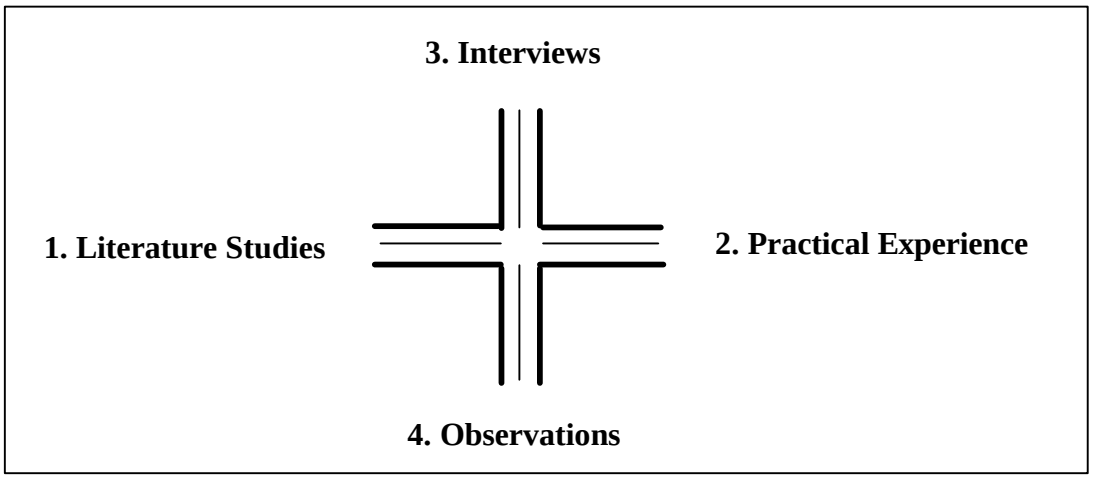

Figure 1 The Crossroad Metaphor.

The Crossroad Metaphor is designed to facilitate evaluation of software development technology. The main purpose is to produce high quality results out of the investigation. Standing in the middle of the crossroad is the best position to gain objective results out of mostly subjective information sources. The more complete subjective information sources to analyze, the more objective conclusions may be drawn. Taken colored information sources together, we may achieve a compromise, i.e. objectivity.

In addition to this, we also want to emphasize the importance of “driving on the roads” in the right order:

- First of all, get an overview of the problem (Literature studies).
- Second, get practical experience (prototyping).
- Third, get a picture of other programers experience (interviews).
- Finally, do some objective research (observation).

The following sections deal with the principles and procedures we worked through during the evaluation of RMI and CORBA.

2.2 Literature Analysis

When we set about this master thesis project we started out studying possible literature about basic principles of Java, RMI, CORBA, JDBC and modern client/server solutions.

In June 1997, the topic of distributed computing was quiet new. CORBA is a technology that has been available and recommended for a couple of years already, but RMI hardly no one had heard about. It was rather difficult to find books on RMI, mostly the books we had, only mentioned that RMI would be possible to use in the next version of the Java Development Kit (JDK) 1.1. A few weeks later new books covering Java RMI were obtainable on the market. Today, January 1998, RMI is a well-known concept among application developers.

While waiting for the delivery of the books we found most of the information needed on the Internet. We used well-known web engines, such as Alta Vista and Excite, and the keywords searched for were CORBA, RMI, Java, distributed systems, distributed objects and other related terms. First we only got sporadic hits of all keywords, except Java, but after a while we had collected a bunch of papers and articles. Sun Microsystems[1] and the Object Management Group (OMG)[2] were great information sources.

About five weeks were devoted for pure literature study. We consider five weeks to be the absolute minimum time to form a general opinion of the two communication technologies. Besides theoretically learning about how and when they may be used and what advantages respectively disadvantages they had, there must also be some time over to practically use and test the technologies. This was done during the second phase of the Crossroad Metaphor.

Information gained from a producer's manuals is often subjective information. Therefore, it could be risky to draw conclusions only from literature sources like handbooks and manuals. However, manuals are perfect knowledge bases to lean on for further investigation. To understand the kernel of communication systems, a number of factors must be understood first. For example, we were forced to learn Java before we could handle CORBA or RMI. The main reason for this was that Java was preferred according to our prototype specification.

2.3 Prototyping

The second phase of the Crossroad Metaphor is the prototyping moment. In the case of investigating communication technologies, practical experience means a lot. By programming, a developer faces all problems and moments of success by him or her self. This enriches the knowledge in a traditional research methodology way[3].

We started to program the Self Provisioning prototype by learning Java through easy programmable applets. In the beginning of this project, June 1997, the JDK version 1.1 was recently released. We used Symantec Visual Café Pro[4], which is a Windows adapted graphical programming tool, to construct the graphical user interface of the application.

As soon as we understood the principles of Java, we started to program the server of the Self Provisioning application. According to the specification (Appendix 3), the server should be able to communicate with a database. The database connection, through which we learned Java DataBase Connectivity (JDBC), was also a new challenge for us. While programming the prototype, we studied user documentations and other related articles and papers. Finally we

[1] http://www.sun.com
[2] http://www.omg.org
[3] Rubenowitz, S. (1980)
[4] http://www.symantec.com

connected the client-, server- and database-parts of the Self Provisioning application. The development process of the Self Provisioning application is described in chapter 4.

We would like to call our development process some kind of waterfall method[1] with features of iterative development. If some programming area or technology was quiet new to us, we first copied a small test example from a book and tried it out. If it seamed to be working well, we modified the code to match our application. We divided the programming code into logically related pieces and stored them on different files. By this we learned more about object oriented system development.

2.4 Interviews

We consider that only programming experience is not enough information to base analysis on when investigating. We decided, in accordance with the Crossroad Metaphor, to complete the information by independent persons' opinions. When evaluating something by testing it out, subjective opinions are generated. The same occurs when interviewing people. We claim that the more opinions to analyze, the better or more correct will the research result be. It is preferred to do the interviews after building the prototype. Otherwise, our own programming actions could have been influenced by the opinions of the interview persons.

Interviews can be done in an either highly structured or low structured way[2]. A highly structured interview contains predefined questions. Exactly the same questions are asked to the interview persons and in exactly the same way, irrespective of whether a question is relevant to a specific person or not. The interviews we made were low structured. With the low structured strategy we had the opportunity to extend the base questions and obtain qualitative answers.

2.4.1 Interview Persons

We made the choice of interview persons by our self, but we got a couple of suggestions from our instructor at Ericsson and recommendations from other people that we met during this project. We got into contact with several interesting people through our work at Ericsson, the university, lectures and IT-seminars.

The choice of interview persons should, in accordance with the Crossroad Metaphor, be based upon a principal called "sound reason"[3]. It means that a researcher him or herself, on a reasonable basis, chooses interview persons to represent a population. For example, if the interview persons were all modern client/server technologies experts, the results of this master thesis would have been rather unreliable. Below are our own sound criteria for choosing interview persons:

- **Number of interview persons.** Interviews generate qualitative information to analyze. In practice this means interviewing one person at time during at least an hour per person. This is a time consuming process, but because of the qualitative answers, the number of persons must not be high like in inquiry researches. We think that interviewing 6-8 persons is quiet enough to get a good analyzing basis. A good argument for not having more interview persons is that it is quiet difficult to find people who are in agreement with RMI.
- **Age.** The age of the interview persons should be as different as possible, so we can find out if there are any differences between young and old people.

[1] Sommerville, I. (1996)
[2] Patel, R. & Davidson, B. (1994)
[3] Befring, E. (1994)

- **Background.** We would like to interview beginners as well as experts and anything between. We define ”beginners” as persons with no communication experience, but some programming background.
- **Known / unknown.** We think that it is good to interview both known and unknown persons. If the person is well-known, the interview could easily be a bit too informal. We decided a third of the interview persons to be known.
- **Men / women.** In this technical line of business, most of the employees are men. Our objective was to get at least one woman to interview. However, this was impossible. One solution was to include our selves, but we choose not to do so as we give our own experience separate from the interviews in this thesis.
- **RMI / CORBA.** As told before, there is lack of RMI people, but we wanted to get 50% RMI experienced interview persons.

2.4.2 Interview Questions

The main purpose of the interviews was to find out how programmers in general experience communication technologies like RMI, CORBA, or both, implemented in Java. We also wanted to know the common arising problems with such technologies if there were any.

The interviews took place isolated from other people and performed in an informal way. The interviews took between 45-100 minutes.

The questions were colored by our own experience. After building the Self Provisioning prototype, we knew quiet well what problems may arise. The questions are about what experience the programmer have had while using RMI or CORBA in combination with Java, see Appendix 4.

2.5 Frequently Asked Questions, FAQ

The fourth “road” in the Crossroad Metaphor, the observation, makes it possible to get into the middle of the crossroad and contributes to produce a well-balanced analysis based on the research material chaos.

Studying frequently asked questions or news group messages, may actually be compared to observation[1]. We can get the information needed for the analysis without active participating. This method is specifically suitable if the purpose is to find advantages and disadvantages of communication technologies. One problem with this method is that we do not know the involved persons’ professional backgrounds. What we do find out is what problems really arise versus what the producers of the technologies promise.

We found a couple of different FAQ sites supplying information about distributed computing. There were also plenty of related news groups available. The FAQ dealt with a lot of diverse problem areas, therefore we decided to divide the questions in different categories. After that we analyzed the most important and interesting categories according to our question at issue.

It appeared that people have had lots of problems related to RMI and CORBA. This is very interesting. We consider this type of information sources to be one of the most reliable because it is a voluntarily forum. The problems are probably real problems that people out there had fight against while programming. They have not been forced to answer questions by an

[1] Esterby-Smith, T. et al. (1991)

interviewer and they have not been influenced by something that could cause misrepresentation of the reality.

We think that if a person gets influenced by, for example, the interviewer, he or she is prevented from spontaneously answer a question. Suppose a question like:

" We had this or that problem with RMI, did you experience the same problem?"

What happen is that the person starts thinking and probably gets to the conclusion that he or she actually had that problem too. In reality, the person may not even have noticed that it was a problem. Is the answer a correct answer then? We do not think so.

3 Distributed Object Systems

The purpose of the following chapter is to give an overview of what distributed object computing is and to explain some of the terminology belonging to it. We also present basic principles of Java and the communication technologies of RMI and CORBA that contribute to make distributed computing possible.

3.1 Distributed Object Computing

The concept of ”Distributed Object Computing” originates from a merge between two promising but previously independent paradigms[1][2], namely those of:

- ”Object Orientation” and of
- ”Client-Server Architecture”, which leads to Distributed Computing

Distributed object computing is a breakthrough framework for computing that has, as mentioned above, resulted from the merge of object-oriented and client/server technologies. Distributed object computing blends the distribution advantages of client/server technology with the richness of real-world information contained in object-oriented models.

3.1.1 Distributed Computing and Client-Server Architecture

The client-server architectural model (Figure 2) is a distributed system model, which shows how data and processing is distributed across a range of processors. The major components are[3]:

- A set of stand-alone servers which offer services to other sub-systems.
- A set of clients that call on the services offered by servers. These are normally sub-systems in their own right. There may be several instances of a client program executing concurrently.
- A network which allows the clients to access these services. In principle, this is not really necessary as both the clients and the servers could run on a single machine.

Clients must know the names of the available servers and the services that they provide. However, servers need not know the identity of clients or how many clients there are. Clients access the services provided by a server through remote procedure calls.

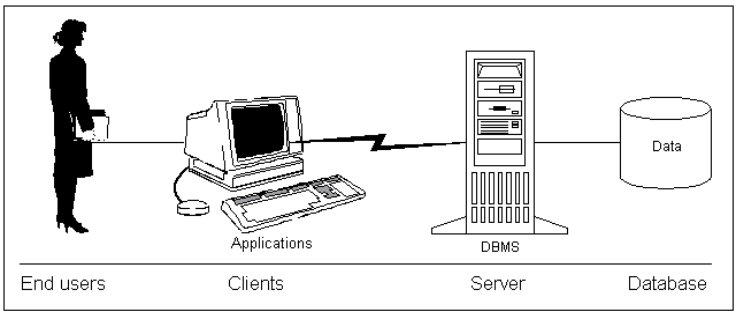

Figure 2. Client / server architecture.

[1] Gilbert, S.J & Landercy, A. (1997)
[2] Fingar, P. & Stikeleather, J. (1996)
[3] Sommerville, I. (1996) pp 225-247

When it comes to the merge of client/server technology and distributed objects, objects are distributed across a network, clients can be servers and conversely, servers can be clients. That really does not matter since it is about cooperating objects: The client requests services of another object, the server object fulfills the request. Clients and servers can be physically anywhere on the network and written in any object-oriented programming language. Although universal clients and servers live in their own dynamic worlds outside of an application, the objects appear as though they are local within the application since the network is the computer. In essence, the whole concept of distributed object computing can be viewed as simply a global network of heterogeneous clients and servers[1].

3.1.2 Object Technology

What is an Object?

An object (Figure 3) is a self-contained software package consisting of its own private information (data), its own private procedures (private methods) that manipulate the object's private data, and a public interface (public methods) for communicating with other objects[2]. An object contains both data and logic in a single software entity or package. Objects provide properties representing a coherent concept and a set of operations to manage these properties. The fusion of process logic with data is the distinguishing characteristic of objects.

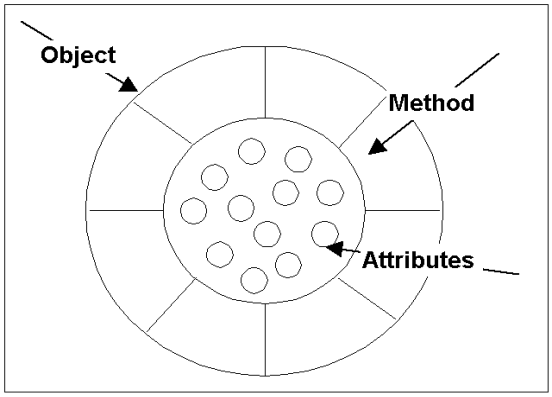

Figure 3. An object.

Each object is capable of acting in much the same way as the real object behaves in the real world. Objects are assigned roles and responsibilities, and they contain all of the information they need to carry out their actions. The only way to use an object is to send it a message that requests a service to be performed. The receiving object acts on the message and sends the results back as a message to the requesting object.

When a service is requested of an object, the object is sent a message, much like a traditional function call. However, the difference is that the rest of the system does not see how the object is implemented and cannot suffer any integration problems if the object's internal implementation (code) is changed[3]. This means programming without assumptions, where functionality is well-defined and programmers do not make assumptions about how the shared routines or systems work on an internal level.

[1] Fingar, P. & Stikeleather J. (1996)
[2] Orfali, R. et al. (1996) pp 1-42
[3] Fingar, P. & Stikeleather J. (1996)

What is a Distributed Object?

A ”classical” object of the C++ or Smalltalk variety is a blob of intelligence that encapsulates code and data. Classical objects provide code reuse facilities via inheritance, encapsulation and polymorphism[1]. However, these classical objects only live within a single program. Only the language compiler that creates the objects knows of their existence. The outside world does not know about these objects and has no way to access them. They are ”buried” in that program. In contrast, a distributed object is a blob of intelligence that can live anywhere on a network[2]. Distributed objects are packaged as independent pieces of code that can be accessed by remote clients via method invocations. The language and compiler used to create distributed server objects are totally transparent to their clients. Clients do not need to know where the distributed object resides or what operating system it executes on; it can be on the same machine or on a machine that sits across an intergalactic network. Distributed objects are smart pieces of software that can interrogate each other - ”tell me what you do.” Distributed objects are dynamic - they come and go and move around.

Heterogeneous

In a heterogeneous environment different pieces of a single application may execute on different platforms[3]. The definition for a platform is here the underlying hardware and the operating system.

In a heterogeneous environment, developers should not need to be concerned with weather all the systems they may use are of the same type of hardware. They should not need to be concerned that all pieces of the software were developed using tools from the same software vendor. They should not even need to be concerned that all pieces of the application are written in the same computer language.

Homogeneous

Homogeneous is the opposite of heterogeneous. In a homogeneous environment different pieces of a single application have to be executed on the same platform. However, it must be noted that in the literature the word homogeneous is often used to describe a distributed system that requires to be written in the same language. When we will describe RMI later in this chapter, we will refer to the latter description of homogeneous.

3.2 Distributed Object Systems

Distributed Object Systems are mechanisms whose intent is to allow programmers to deal with objects on remote systems, as if the object were local[4]. In fact, in some systems, there is absolutely no difference in programming for a local object versus a remote one.

Most Distributed Object Systems utilize sockets for communication at a lower level. They use encapsulation and remove from the programmers cognitive load the complexity of socket based programming. In short, their purpose is to make life simpler for programmers building distributed application.

[1] Orfali, R. et al. (1996) pp 1-42
[2] Orfali, R. et al. (1996) pp 1-42
[3] Kollars, C. (1997)
[4] Jayson, R. (1997)

3.2.1 How Distributed Object Systems Work

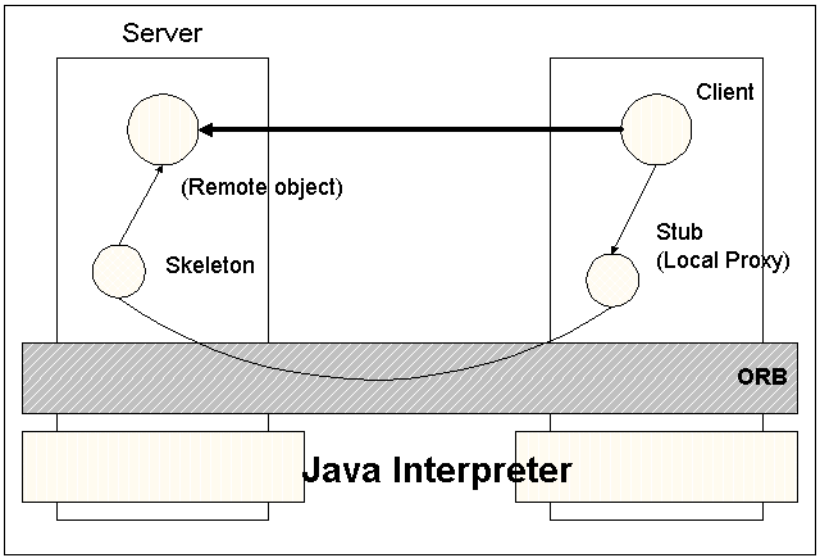

Figure 4. How distributed systems work.

From the programmers' perspective, all that needs to be thought about is that the client is referring to an object, as depicted in the bold black lines in Figure 4[1]. Depending on the exact system used, the programmer may or may not have to define the exact computer the remote object resides on.

What is actually occurring by the Distributed Object System however, is that all communication to that remote object is occurring through a local Proxy (Stub) which communicates with its counterpart Skeleton on the remote system. For every method that is called on the remote object, the parameters are sent across and the method results returned (this is known as marshalling).

The Proxy (Stub) and Skeleton are typically automatically generated by running a precompiler.

Examples of Distributed Systems are Distributed Component Object Model (DCOM) from Microsoft, Common Object Request Broker (CORBA) from the Object Management Group, and Remote Method Invocation (RMI) from Sun.

[1] Jayson, R. (1997)

3.3 Java

3.3.1 Use of Java

A Web Designing Language and More

Java is not only a new programming language, it is the first true programming language whose primary use is for web pages and consequently perfect for building Self Provisioning applications. With Java, programmers can design web pages containing small embedded applications, called applets, which can be used as commercial on-line web-shops.

Intelligent Agents

Another potential use of Java is for the creation of primitive intelligent agents. Intelligent agents (or just agents) are small processes, usually running in the background, that performs tasks for the user. For example, the Java-based agent may be told to search the web for advertisements for used cars made in 1994 for under $12,000. This amounts to little more than a search engine, but agents can do more complicated tasks as well. The agent can be asked to check each of the web pages on the users hotlist to find out if they have changed and to create a web page entitled "Changed Hotlist Links" with links to each of the changed pages in the hotlist.

Platform Independent Programming

The basic concept behind Java is a simple one: true platform independence[1]. In today's distributed computing environment, if the programmer wants to develop a new application then it has to be written for a specific combination of hardware and operating system - for example, a word processor developed for Windows would not work on a UNIX workstation. So if a software developer wants all computers to be able to use its applications, a separate version has to be developed for each operating system. This is an expensive business, and so most manufacturers tend only to support a limited number of platforms - those with large user bases such as Windows-based PCs, Macs and UNIX systems - and ignore the others.

Java is designed to break the fixed links between the application and the operating system, so that developers can create applications that can be run on any computer, just as a web document can be viewed by any browser. Fundamentally, this will allow developers to create an application once and then distribute it to anyone, no matter what computer and operating system they have.

3.3.2 Features

According to Sun, Java is:

”A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded and dynamic language.”[2]

The software engineers wanted to construct a language that was easier to use, but at the same time more reliable and powerful than for example C++. Although the syntax in Java became much like that in C++, the behavior is not nearly so analogous - it turned out to be just better.

[1] http://www.javasoft.com
[2] http://java.sun.com

Simple

Java does not support the goto statement like in C/C++. Instead it provides labeled break and continues statements and exception handling. Java does not use header files and it eliminates the C preprocessor. Because Java is object-oriented, C constructs like struct and union have been removed. Java eliminates the operator overloading and multiple inheritance of C++. Instead of declaring multiple inheritance in a traditional way, Java uses interfaces similar to Objective C protocols, which are easier to use and result in the same effect. Additionally, there is no need to explicitly free storage in Java. Unreferenced memory is automatically garbage collected.

Object-Oriented

Unlike C++, which is a confusing combination of object and functions, everything in Java is an object. Strings are objects, numbers are objects, threads are objects, and even applets are objects. Because of this, Java has all the helpful features of object-oriented systems. It is ”objects all the way down”.

Distributed

Distributed simply means that the language provides a lot of high level support for networking. For example, the URL class and related classes in the java.net package make it almost as easy to read a remote file or resource, as it is to read a local file. Similarly, in Java Remote Method Invocation (RMI) API allows a Java program to invoke methods of remote Java objects, as if they were local objects. Java also provides traditionally lower-level networking support, including datagrams and stream-based connections through sockets.

Interpreted

The Java compiler generates bytecode that is interpreted by the Java Virtual Machine (JVM) at runtime. The fact that bytecode are generated is important, because it avoids the problem of basing the binary code on a basic set of primitive types such as integers and floating point which would be tied to a specific platform.

The Just In Time (JIT) compiler is an integral part of the Java Virtual Machine, and makes Java run faster. When a JIT compiler is present, the Java Virtual Machine does something different. After reading in the .class file for interpretation, it hands the .class file to the JIT. The JIT compiler will take the bytecodes and compile them into native code for the machine that it is running on. It can actually be faster to grab the bytecodes, compile them, and run the resulting executable than it is to just interpret them.

Robust

Java’s run-time garbage collector removes objects from memory that no longer is needed. A program may not run out of memory because a programmer forgot to explicit delete an object. The garbage collector takes care of this. Java’s total orientation toward objects removes another construct that has been a blight of programmers - the pointer. Because of this combination of garbage collection and removal of the pointer construct, Java has generally taken the problem of memory management away from the programmers.

Secure

The designers of Java knew that applets could be downloaded over insecure networks, so they included a bytecode verifier in the Java Virtual Machine’s interpreter. This checks that memory addresses are not forged to access objects outside of the virtual machine, that applet objects are accessed according to their scope (public, private, and protected), and that strict runtime

type enforcement is done both for object types and parameters passed with method invocations. The bytecode verifier does these checks after the bytecodes are downloaded but before they are executed. This means the verified code runs faster because it need not perform these security checks during execution.

Architectural Neutral

Because Java programs are compiled to an architectural neutral byte-code format, a Java application can run on any system, as long as that system implements the Java Virtual Machine. This is particularly important for applications distributed over the Internet or other heterogeneous networks.

Portable

The bytecode-based system is important to writing a portable interpreter. The bytecodes generated by the compiler are based on the specification of Java Virtual Machine, which, as its name suggest, is not a specific hardware platform but a machine implemented software. The virtual machine is very similar to a real CPU with its own instruction set, storage formats, and registers. Since it is written in software, however, it is portable. All that is needed to take Java code compiled on one platform and run it on another is to provide a Java interpreter and runtime environment. The runtime system is written in an easily portable fashion. Once this system is available, everything becomes easy. There is no need to port the Java compiler - it is written in Java.

High-Performance

Java is high-performance. This point is often argued about, since Java is an interpreted language. Indeed, while the performance of Java's interpreted bytecodes is much better than what high-level scripting language generally offers they are still on the average about 20 times slower that native C/C++ codes.

Multithreaded

Java is a multi-threaded language, meaning that many separate processes can be executed simultaneously. It contains many primitives to synchronize operations between threads, making it easier for programmers to develop multi-threaded applications.

Dynamic

By being able to load parts of new code on the fly - without compilation - over the net, Java applications can upgrade themselves dynamically to adapt to an evolving environment, and hence keep pace with fast-changing net technology. There is no need for developers to create, and for users to install major new software versions. New features are incorporated transparently as needed.

Free

Marketing strategy contributed a great deal to Java success and world-wide use. Sun early realized the importance of generating broad product interest and acceptance, and therefore has been freely offering the binaries of key Java components via the Internet. Since the 1995 Sun Conference, the Java Development Kit can freely be downloaded from Sun's site[1]. Anyone with Internet access can quickly obtain the latest version of the JDK, whatever his or her operating system is.

[1] http://java.sun.com

3.4 RMI

3.4.1 Overview of RMI

Remote Method Invocation (RMI) is a modern technology for distributed computing, which allows computers in different address spaces to communicate. Java, like many other programming languages support sockets for such communication. However, sockets communicating will soon become an old-fashioned approach and the newer remote methods will take over. In spite of this, we must remember that these modern methods still communicate via sockets – although in a different and transparent way.

RMI is closely related to the Remote Procedure Call (RPC), common in C++. RPC abstracts the communication interface to the level of a procedure call and gives the programmer an illusion of calling a local procedure call[1].

RMI has several advantages over traditional RPC systems because it is part of Java's object oriented approach. RPC does not translate well into distributed object systems, where communication between program level objects residing in different address spaces is needed. In order to match semantics of object invocation, distributed object systems require a system like RMI.

In the Java distributed object model, a remote object is one whose methods can be invoked from another Java Virtual Machine, potentially on a different host. An object of this type is described by one or more remote interfaces, which are Java interfaces that declare the methods of the remote object. Remote Method Invocation (RMI) is the action of invoking a method of a remote interface on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

An extra feature that most RPC systems do not have is that RMI passes objects as parameters in remote procedure calls[2]. Passing a remote object as a parameter is actually a passing of a stub for the object. The real object always stays on the machine where it was originally started. The stub that it passes then invokes methods back to the original object. To marshal and unmarshal parameters, RMI uses Object Serialization[3]. It does not truncate types and supports true object-oriented polymorphism.

In a distributed system, a way for clients to find the server is needed. RMI provides a simple name lookup object that allows a client to get a stub for a particular server based on the server's name.

RMI is a part of JDK 1.1 and accordingly platform independent. The programmer does not need to learn a new programming language to implement RMI. He or she just needs to let Java do all the work automatically. This also differentiates RMI from other RPC systems. Unfortunately RMI can only be written in Java, which means that it is a language dependent system and works only in homogeneous environments[4].

[1] http://java.sun.com
[2] http://java.sun.com
[3] http://java.sun.com
[4] http://java.sun.com

3.4.2 Architectural Overview

3.4.3 The Stub / Skeleton Layer

When a client invokes a server, several layers of the RMI system come into play. The first, and most important to the programmer, is the Stub/Skeleton Layer[1], Figure 5. The stubs are Java code that communicates with the other layers. The Java RMI system automatically enables the use of several helper functions. By inheriting from the RMI classes, a class implements the stubs or skeletons. Stubs are reserved for client code and skeletons refer to server code. Once the stubs and skeleton layers are completed, they pass through the other two layers in the RMI system.

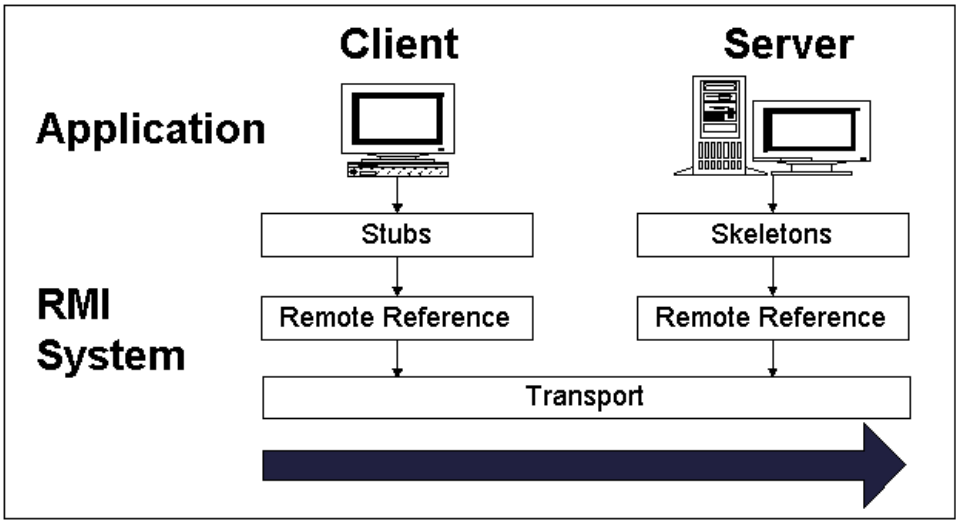

Figure 5. The RMI system architecture.

3.4.4 Remote Reference Layer

The second layer is the Remote Reference Layer[2], which is responsible for determining the nature of the object. Does it reside on a single machine or across a network? Is the remote object the kind of object that must be declared and initialized beforehand? The remote reference layer handles all of these situations, and many more, without a programmer's intervention.

3.4.5 Transport Layer

Finally, the Transport Layer[3] is similar to a translator that takes RMI code, turns it into TCP/IP (or whatever communication mechanism is used), and let it fly over the network to the other end. Because the RMI system supports a technology called object serialization, any object passed as parameter to a remote method, no matter how complicated, are converted into simple streams of characters that are then easily re-converted into their original object representation.

[1] http://java.sun.com
[2] http://java.sun.com
[3] http://java.sun.com

A client that invokes a remote server first talks to its stub code, which, in turn, sends a message to the remote reference layer, which then passes it through the transport mechanism to the other machine. The other machine takes what it gets through the transport layer, re-translates into the remote reference layer representation, which passes it to the skeleton code where the request finally makes its appearance at the remote method.

3.4.6 Object Implementation

Most of the methods needed for working with remote objects are in three packages: java.rmi, java.rmi.server, and java.rmi.registry. The first package defines classes, interfaces and exceptions that will be seen on the client side. The second package deals with corresponding server features and the third package handle features that are used to locate and name remote objects.

The server side

There are a few things to mention according to server programming:

- To create a remote object, an interface that extends the java.rmi.Remote interface must be defined.
- Each method in the interface must be declared that it throws a java.rmi. RemoteException.
- A class must implement the remote interface. This class should extend java.rmi.UnicastRemoteObject.
- After constructing an object, it must be bound to a name in the Naming registry:

```
ObjectImpl object = new();
Naming.rebind(”Name”, object);
```

The Naming Registry keeps track of the available objects on a RMI server and the names by which they can be requested.

The client side

Before a client can call a remote method, it needs to retrieve a remote reference to the remote object. A program retrieves a remote reference by asking the Naming Registry on the server for a remote object. It asks by calling the registry’s lookup (String URL) method:

```
ObjectRef objRef = (ObjectRef) naming.lookup(”Name”);
```

Once the object is retrieved, the client may invoke a remote method. Figure 6 shows how a client talks to the Naming Registry server.

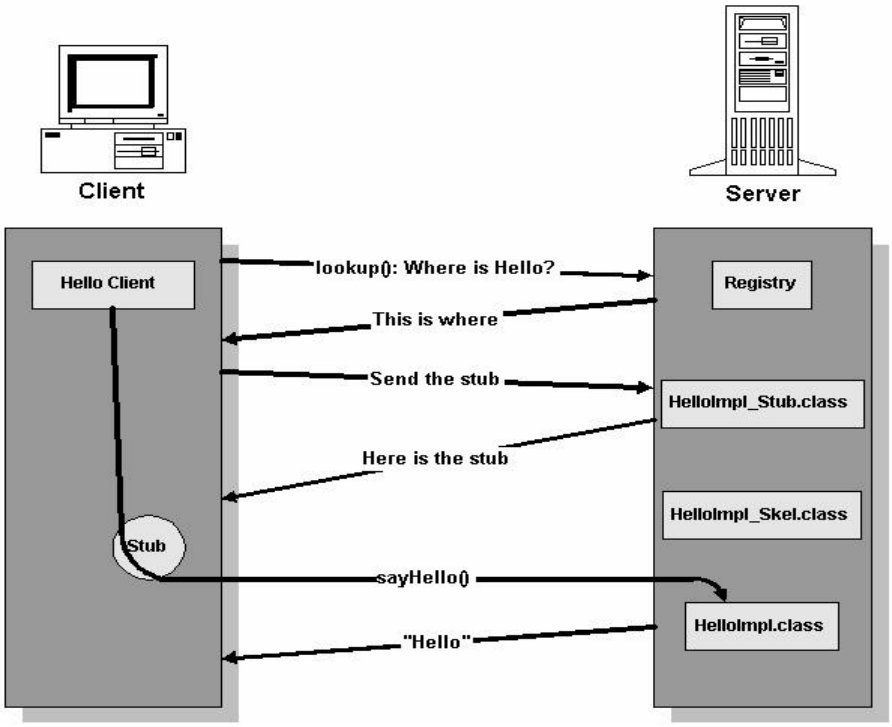

Figure 6. A Client talking to registry

The complete programming process of an RMI system is described in Appendix 1.

3.4.7 The Vision

RMI was designed with the idea that with minimal effort a programmer will be able to create complex distributed systems with all the advantages of Java, and none of the detriments of other distributed designs. In fact, with the addition of a single line in the code, a programmer can make an object a distributed object instead of a local one.

Even though the code gives the illusion of a normal, single process application, it is in fact a distributed system. Java RMI says rather than overloading an application, why not delegate to other applications.

3.5 CORBA

3.5.1 The Object Management Group, OMG

The Object Management Group, is a non-profit consortium created in 1989. It was originally formed by 13 companies but has now grown to over 700 software vendors, developers and users[1]. The OMG was founded to specify a standard for the interoperability of object-oriented software across operating systems and platforms in a heterogeneous network environment. In other words, OMG do not produce software or implementation guidelines; only specifications, which are, put together using ideas of OMG. CORBA is the specification of an architecture and interface that enables applications to request the services of an object in a transparent independent manner, regardless of language, operating system, or other local considerations[2].

3.5.2 The Object Management Architecture, OMA

The Object Management Architecture (OMA) is the center of all the activity undertaken by the OMG. OMG was formed to help reduce the complexity, lower costs, and hasten the introduction of new software applications. OMG does this through the introduction of the architectural framework OMA with supporting detailed interface specifications. The implementations are the domain of vendors, end-users, and those developing products and projects to solve a particular computing or business problem. Specifications are the domain of OMG membership. These specifications are intended to drive the industry towards interoperable, reusable; portable software components based on open, standard object-oriented interfaces.

OMA is a high-level vision of a complete distributed environment. It forms a conceptual roadmap for assembling resultant technologies while allowing for different design solutions. Specifically, the reference model identifies and characterizes components, interfaces, and protocols that compose the OMA but does not in itself define them in detail[3].

The OMA consists of four components; Object Request Broker, Object Services, Common Facilities, and Application Interfaces. Figure 7 illustrates the primary components in the OMG Reference Model architecture. Descriptions of these components are available further below. Portions of these descriptions are bases on material from OMG[4], and Orfali R. & Harkey D[5].

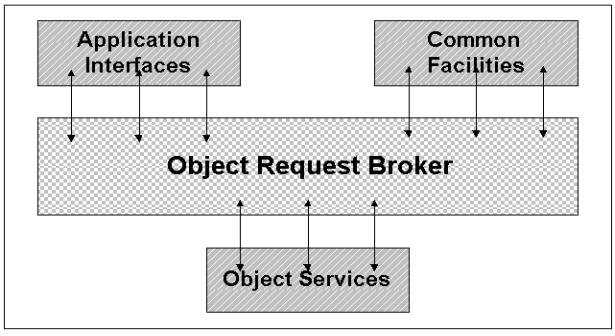

Figure 7. OMG Reference architecture

[1] Orfali, R. et al. (1997) pp 1-27
[2] de Jager, N. (1996)
[3] OMG1
[4] OMG1
[5] Orfali, R. et al. (1997) pp 1-27

Object Request Broker

The Object Request Broker is the component, which constitutes the foundation of OMA and manages all communication between its components[1]. It is commercially known as CORBA and the communications heart of the standard. It provides an infrastructure allowing objects to converse, independent of the specific platforms and techniques used to implement the objects. Compliance with the Object Request Broker standard guarantees portability and interoperability of objects over a network of heterogeneous systems.

Object Services

Object Services or CORBA services, which it is commercially known as, are collection of system-level services with Interface Definition Language[2] (IDL) specified interfaces. The Object services can be seen as augmentation and completion of the functionality of the ORB. Object Services is covered by another OMG specification, Common Object Services Specification[3] (COSS) that defines a set of objects that perform fundamental operations such as lifecycle, naming, event, persistence services, relationships, externalization, transactions, concurrency control, security, licensing, queries and versioning[4]. It is though important to note that ORB vendors are not obliged to implement all of the object services mentioned above to become CORBA compliant, but any implemented service must conform to the specification.

Common Facilities

Common facilities are commercially known as CORBA facilities. Common Facilities (CF) are the newest addition by the OMG. Unlike the ORB and Object Services that are low-level fundamental operations, the CF is focused on the application. Common facilities are collections of IDL-defined frameworks that provide services of direct use to application objects. Common Facilities is thus the next step up in the semantic hierarchy.

Application Interfaces

These are interfaces developed specifically for a given application. Because they are application-specific, and because the OMG does not develop applications (only specifications), these interfaces are not standardized. However, if over time it appears that certain broadly useful services emerge out of a particular application domain, they might become candidates for future OMG standardization[5].

3.5.3 The Architecture of an ORB

A CORBA 2.0 Object Request Broker is the middleware that establishes the client/server relationship between objects[6]. Using an ORB, a client object can transparently invoke a method on a server object that can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results. The client does not have to be aware of where the objects is located, its programming language, its operating systems, or any other system aspects that are not part of an objects interface. The client/server roles are only used to coordinate the interactions between two objects. Objects on the ORB can act as either client or

[1] Keahey, K. (1997)
[2] OMG 1
[3] OMG 1
[4] Orfali, R. et al. (1997) pp 1-27
[5] Schmidt, C. (1997)
[6] Orfali, R. et al. (1997) pp 1-27

server, depending on the occasion[1]. Figure 8 below shows the CORBA ORB architecture and the client and server sides.

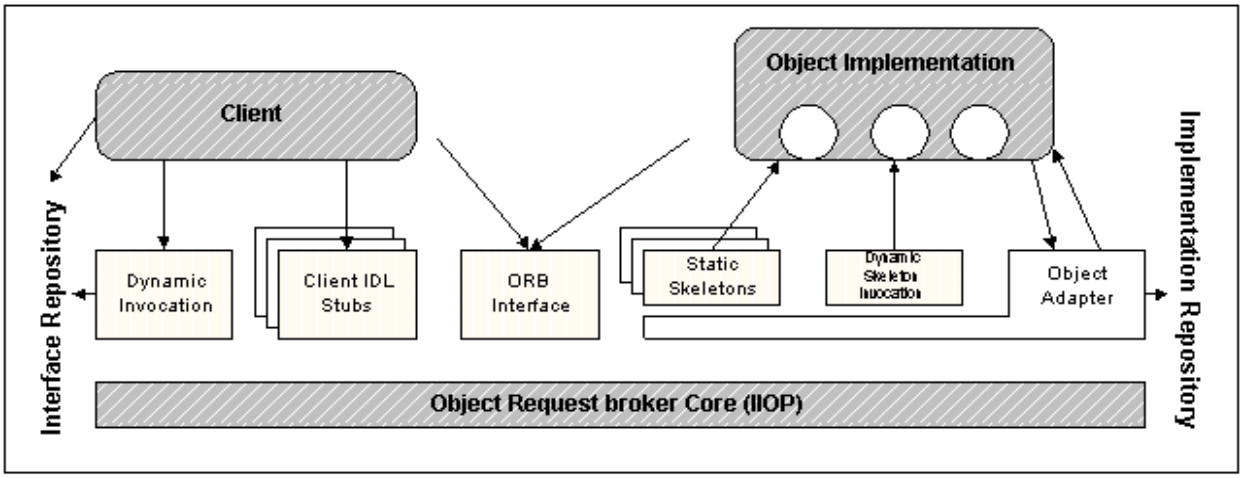

Figure 8. CORBA ORB Architecture.

Object Implementation. This defines operations that implement a CORBA IDL interface. Object implementations can be written in a variety of languages including Java, C, C++, Smalltalk and Ada[2].

Client. This is the program entity that invokes an operation on an object implementation. See figure 9.

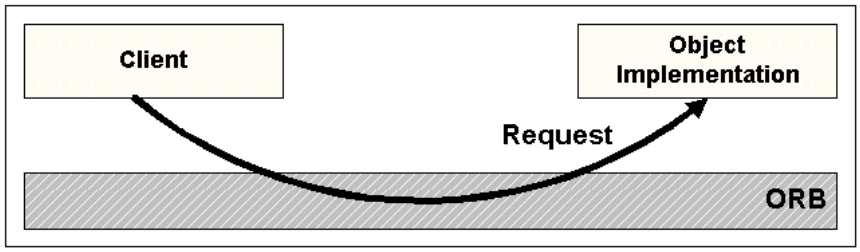

Figure 9. A request being sent through the Object Request Broker.

Object Request Broker, ORB. The ORB provides mechanisms for transparently communicating client requests to target object implementations[3].

ORB Interface. This interface provides various helper functions such as converting object references to strings and vice versa, and creating argument lists for requests made through the dynamic invocation interface described below.

CORBA IDL Stubs and Skeletons. CORBA IDL stubs and skeletons serve as the "glue" between the client and server applications, respectively, and the ORB[4].

Dynamic Invocation Interface, DII. This interface allows a client to directly access the underlying request mechanisms provided by an ORB. Applications use the DII to dynamically issue requests to objects without requiring IDL interface-specific stubs to be linked in.

[1] Orfali, R. et al. (1997) pp 1-27
[2] Orfali, R. et al. (1997) pp 1-27
[3] Schmidt, C. (1997)
[4] Schmidt, C. (1997)

Dynamic Skeleton Interface, DSI. This is the server side's analogue to the client side's DII.

Object Adapter, OA. The Object Adapter provides the run-time environment for instantiating server objects, passing requests to them, and assisting them object IDs - CORBA calls the IDs object references, and registers the classes it supports and their run-time instances (i.e. objects) with the Implementation Repository.

The Interface Repository APIs. This interface makes it possible to obtain and modify the descriptions of the entire registered component interfaces, the methods they support, and the parameters they require.

The Implementation Repository. The Implementation Repository provides a run-time repository of information about the classes a server support, the object that are instantiated, and their Id's.

The Interface Definition Language, IDL

Most interprocess object models are expressed in terms of a language for defining interfaces. Since the definition of RPC mechanisms, these languages have been known as IDL[1]. The basic purpose of an IDL is to enable the language-independent expression of interfaces, including the complete signatures (name, parameters and result types) of methods. This is accomplished by providing a mapping between the IDL syntax and whatever language is used to implement client and server objects.

CORBA IDL is a C-like language with many constructs similar to C++. IDL follows the same lexical rules as C++, while introducing a number of specific keywords to address the distributed system environment. Writing interface definitions in IDL is like writing class declarations in C++. Since IDL is designed only for interface definition, it lacks the constructs of an implementation language. In particular there is no concept of public and private parts of the interface declaration, since the notation of encapsulation is implicit in the separation of the IDL interface from the implementation[2].

Internet Inter-ORB Protocol, IIOP, and General Inter-ORB Protocol, GIOP

CORBA 1.1 was only concerned with creating portable object applications; the implementation of the ORB core was left as an "exercise for the vendors". The result was some level of component portability, but not interoperability. CORBA 2.0 added interoperability by specifying a mandatory Internet Inter-ORB Protocol (IIOP). The IIOP is basically TCP/IP with some CORBA-defined message exchanges that serve as a common backbone protocol. Every ORB that calls itself CORBA-compliant must either implement IIOP natively or provide a half-bridge to it. It is called a half-bridge because IIOP is the "Standard" CORBA backbone[3]. In order to make bridges possible it is necessary to specify some kind of standard transfer syntax. This function is fulfilled by General Inter-ORB Protocol (GIOP) defined by the OMG. It has been specifically defined to meet the needs of ORB-to-ORB interaction and is designed to work over any transport protocol that meets a minimal set of assumptions[4].

[1] OMG 1
[2] de Jager, N. (1996)
[3] Orfali, R. et al. (1997) pp 1-27
[4] Keahey, K. (1997)

3.5.4 Different Vendors, Different ORBs

There are many implementations of CORBA currently available and they vary in the degree of CORBA compliance, quality of support, portability and availability of additional features.

Examples of implementations that are CORBA compliant (2.0) today are:

- VisiBroker from VISIGENIC
- ObjectBroker from Digital
- Orbix from Iona

Further examples of CORBA ORB vendors are: AT & Global Information Solutions, Black and White Software Inc., Expertsoft, IBM, International Computer Limited, ILOG inc., NEC Corporation, NetLinks Technologies Ltd., Object Oriented Technologies Ltd., PostModern Computing Technologies and Sunsoft

3.6 The client/server development process using CORBA and RMI

When designing a client / server solution, the development technique for most distributed communication technologies follows a kind of step-by-step process. The process plans for RMI and CORBA are discussed in Appendix 1 and Appendix 2.

4 The Self Provisioning Prototype

Prototyping is a well-known concept in the information technology trade. System developers use prototyping to make sure that the information system really matches the users requirements[1]. The prototype is a model of the real world and should give the illusion of the functionality of a final software system.

The main purposes of developing a prototype, according to Sommerville[2], are:

- It is easier to identify misunderstandings between software developers and users by demonstrating the functions of the system.
- Missing user services may be detected.
- Difficult-to-use or confusing user services may be identified and refined.
- Software development staff may find incomplete and/or inconsistent requirements as the prototype is developed.
- A working, though limited, system is available quickly to demonstrate the feasibility and usefulness of the application to management.
- The prototype serves as a basis for writing the specification for a production quality system.

Many software prototypes are demonstrations of user interfaces, evaluated by end users. During the prototype development process of the Self Provisioning application, no end users of the system were involved. The information required to build the prototype was collected by conversations between us and our instructor at Ericsson.

This section covers the Self Provisioning prototype development process.

4.1 Objectives and Requirements of the Prototype

The primary reason of developing the Self Provisioning prototype was to find out advantages and disadvantages in communication programming using RMI and CORBA. In addition, we were asked to design a proper user interface that would be easy for the end user to understand and use. Another research area was to examine and implement the Java DataBase Connectivity (JDBC) technology in the application.

The application should operate within a distributed environment, which means across a network, like the Internet. The user of the application should be able to use a web browser to perform service functions provided by the Self Provisioning system. We do not know what kind of platform the user uses and because of this, the application has to be written in a platform independent language, such as Java. The system consists of a client, a server and a database part.

4.2 The miniCustomerCare System

The miniCustomerCare (mCC) application is a system developed by Ericsson Telecom AB. The application is developed for and used by operators a telecommunication companies. The

[1] Andersen, E.S. (1994) pp 405
[2] Sommerville, I. (1996) pp 138

operators take service orders from customers by telephone and forward the orders to a Service Configuration (SC) application at Ericsson.

This process is however, an expensive as well as a time consuming process. The operators want to reduce the costs by allowing their customers to order and implement services by them selves and consequently minimize the cost of employees.

4.3 What the Self Provisioning Application Offers

The Self Provisioning application is supposed to provide users with the ability to order different internet services. The objective is to allow any person or organization, who has internet access, to order additional internet services from their common web browsers, without any contact with humans. Figure 10 shows the architectural principles of the environment where the Self Provisioning application is supposed to act in. The Self Provisioning server is running within a larger server machine at Ericsson. This server also includes a web server, from where the Self Provisioning client (applet) can be downloaded. The application should be able to handle client and service functions. Implementation of a service is completed as soon as the order is confirmed. We will describe this in more detail later.

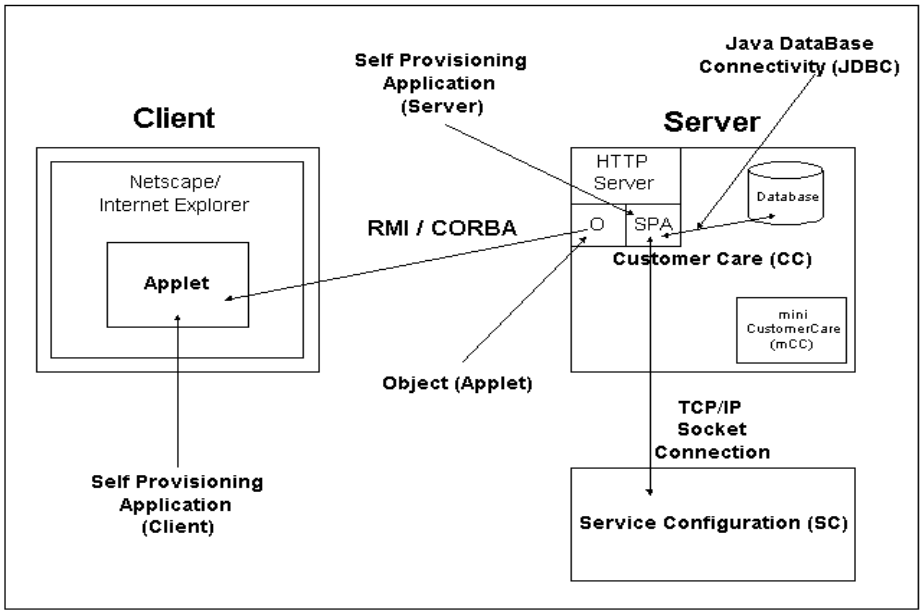

Figure 10. The Self Provisioning application architecture.

4.4 Method

Building the Self Provisioning prototype was, like building a real software system, an iterative process[1]. We wanted to quickly learn the principles of Java programming. Therefore, we started to program a traditional ”Hello World” applet. We thought that, if we could understand the basic principles of Java programming, we could also manage to create a larger system. This was an important strategy; build small test modules - if they work out, start from the beginning and do additional changes to them. Repeat this process until the whole system is complete.

[1] Andersen, E.S. (1994) pp 409

Next move was to figure out how the object model of the final prototype would look like.

4.5 Modeling the Self Provisioning Application

When developing a software system, it is important to create an object model of the system. An approach of creating a good object model of a system is to split the system into small and logically connected pieces. It seemed to be rather easy to divide the Self Provisioning application into smaller parts. Because of its three tier architecture it was naturally divided into a client, server and database structure. According to the principle of object technology[1], every single entity is an object. Therefore, both the client and the server side may be divided into even smaller parts.

Figure 11 shows a small piece of the Self Provisioning object diagram with relationships between customers and services.

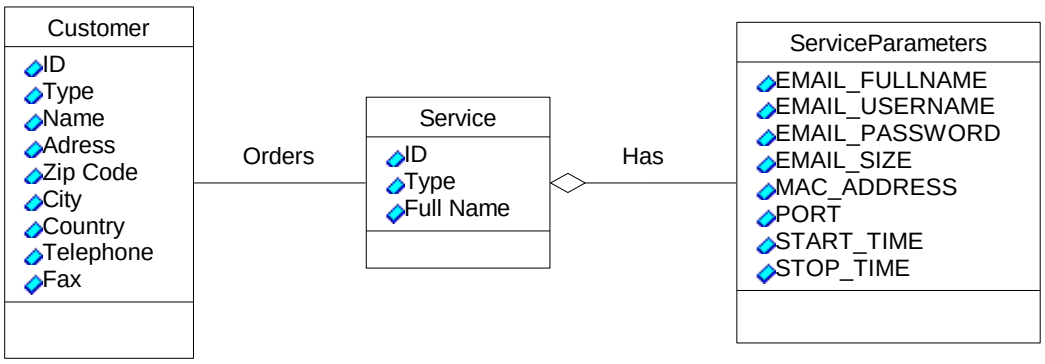

Figure 11. A customer orders a service. A service has service parameters.

4.6 Designing the User Interface

4.6.1 Who is the User?

This is an important question. The user is a subscriber, preferable a private person, public administration or an organization. The user should be able to order and supervise his or her own net services by operating through a web-site on the Internet. The user may not be an IT-expert, but he or she should be familiar with a couple of terms, such as ”mailbox size” or ”mac address”.

4.6.2 Use Cases

A typical use case scenario[2] of using the Self Provisioning Application would look something like the one shown in Figure 12.

[1] Booch, G. (1994)
[2] Booch, G. (1994) pp158-159

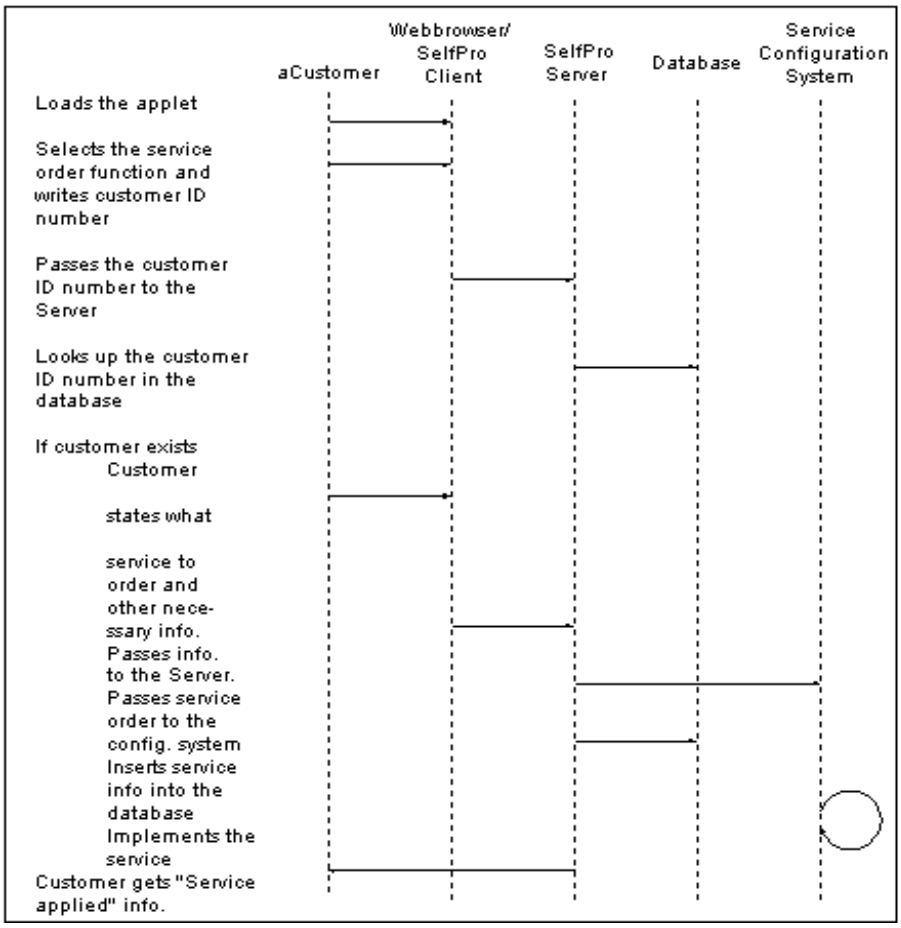

Figure 12. Use case scenario. Customer orders a service.

4.6.3 Designing the Client

User manuals about distributed object communication recommend, as we describe in Appendix 1 and Appendix 2, a step by step process when developing a distributed client/server application[1]. This process starts by writing the required interfaces for the application. We developed the Self Provisioning Application in a different order. Actually we started by creating the Graphical User Interface (GUI) for the client. By doing this we could easily survey the situation. We used Symantec's Visual Café Pro[2] to program the GUI.

4.6.4 Visual Café Pro

Symantec's Visual Café Pro[3] gives the complete environment, plus all the tools needed to visually create applets and applications. It is possible to build dynamic Web pages with interactive database links in one complete, fully-integrated environment[4].

Visual Café Pro takes Java development to a new level by incorporating three powerful development tools: Visual Café, dbANYWHERE Workgroup Server, and the Pro Extension. Assemble complete Java applets and applications from a library of standard and third party objects without writing source code. Visual Café Pro seamless integrates visual and source

[1] Sridharan P. (1997)
[2] http://www.symantec.com
[3] http://www.symantec.com
[4] http://www.symantec.com

development of Java software, allowing the developer to switch effortlessly between visual and source views. Modifications that are made in the visual view are immediately reflected in the source code.

We used Symantec Visual Café Pro when building the GUI of the Self Provisioning prototype. The experience we gained using this visual tool are presented later in section 5, ”Results”.

4.6.5 The Graphical User Interface

The Graphical User Interface is the visible part of the client. It contains some integrated, hierarchical ordered Java panels. Each panel is stored in a separate class and file. Figure 13 shows the file system principal.

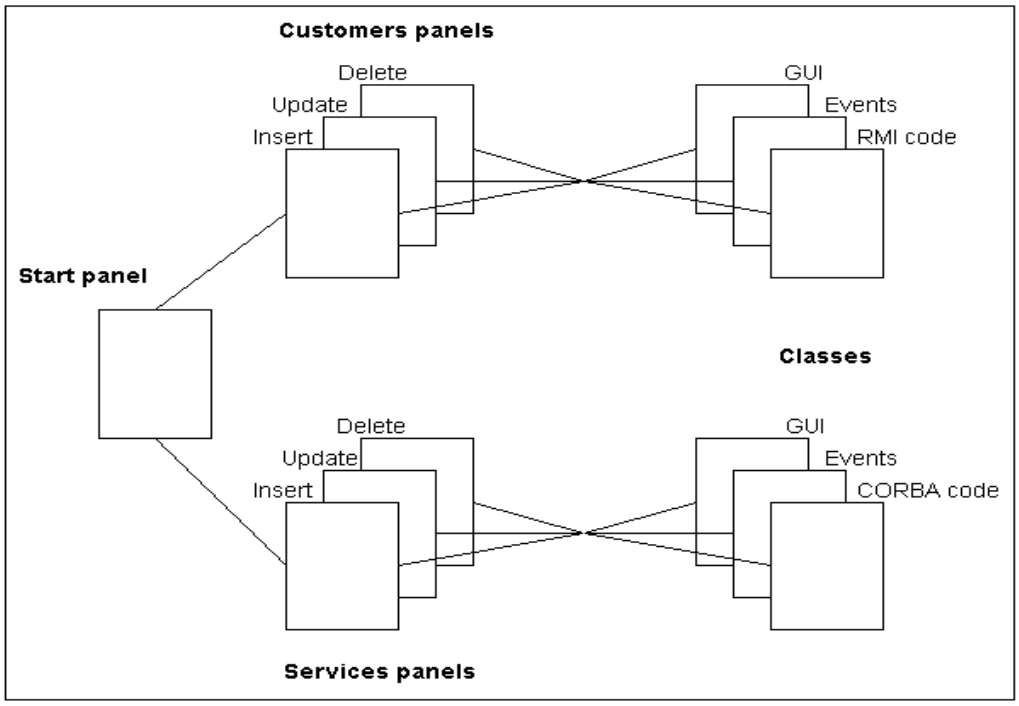

Figure 13. The different files and classes in the Self Provisioning GUI.

The client is a single Java applet, containing the panel hierarchy in the picture above. The start panel is a kind of welcome window only containing two buttons, Figure 14. All panels appear in the same applet area, either in show or hide mode. All panels consist of labels, text fields, text areas, buttons, choices or check boxes, which belongs to the java.awt package. The tab- and border-panels belongs to the Symantec.awt package.

Figure 14. The StartPanel.

When the user clicks on the customer button for example the tab panel situation shown in Figure 15 appears in the applet and the start panel is hidden. This is done using the card layout property provided in Java.

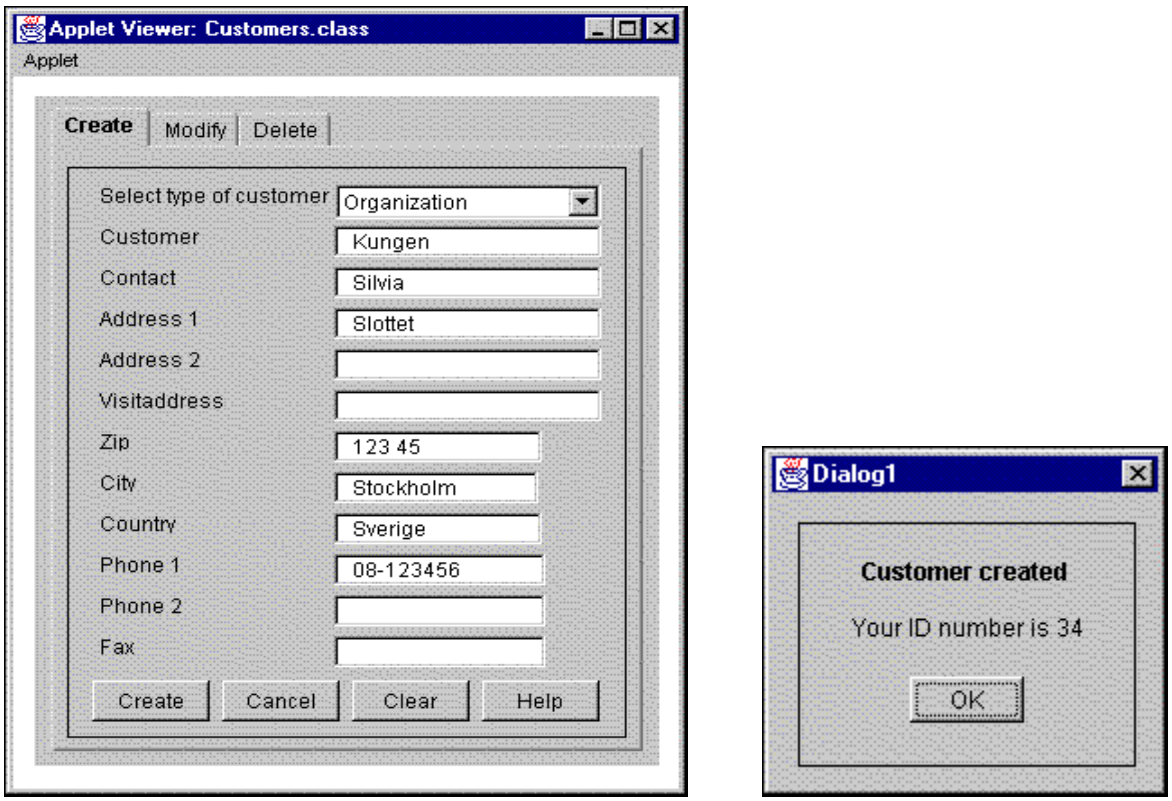

Figure 15. The create customer panel and a dialog box.

When choosing the ”Create new customer” tab panel, the user is prompted for personal user information. When the Create button is clicked a server invocation event occurs and a new

customer is being inserted into the existing customer database. After inserting the customer the dialog box in Figure 15 appears, showing the new assigned customer ID number.

The Modify and Delete Customer tab-panels are similar to the Create panel. The difference is that the program first requires a customer ID number as shown in Figure 16.

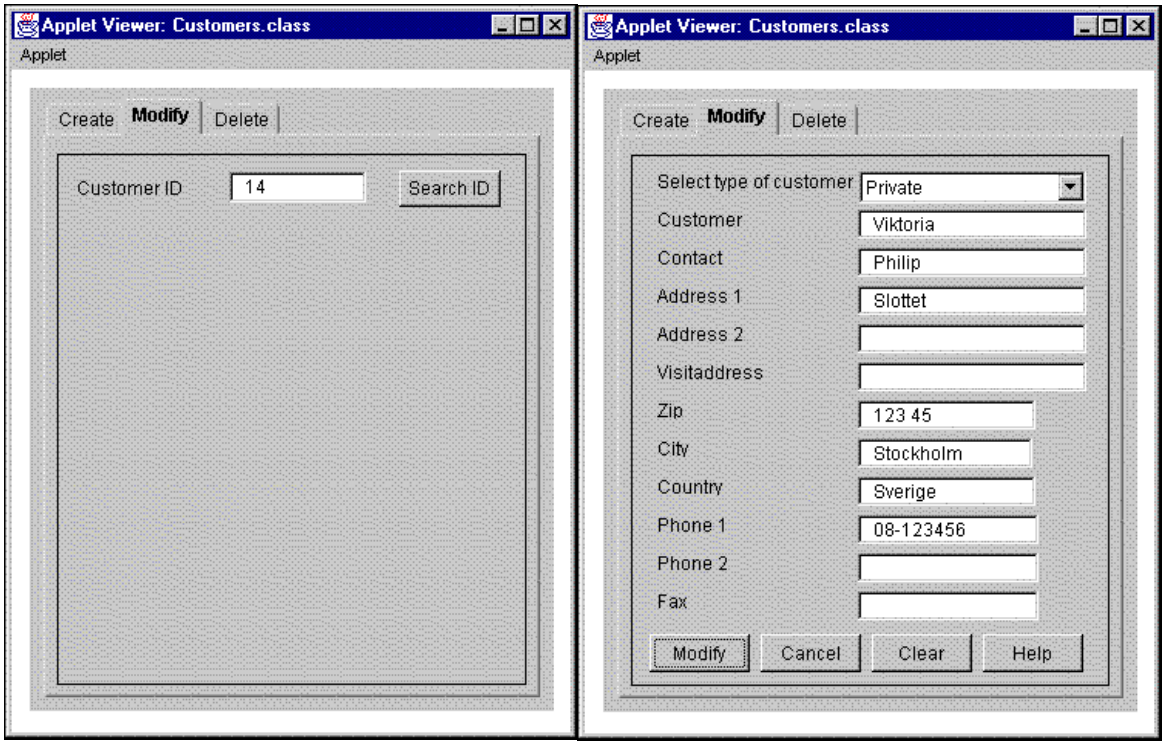

Figure 16. Modify customer panels.

When clicking on the Search ID button the server is triggered by a client invocation. The server checks if the user exists. If the user is valid the server returns all necessary information about him or her. When this is done the user can simply modify the information, click on the Modify button and wait for a ”Modify Success” dialog box to appear. Exactly the same occurs when a user wants to stop being a customer. For security reason the user has to ensure twice that he or she really wants to delete customer information. This is also done using dialog boxes.

On this customer side, the communication between the client and the server uses RMI.

When the Service button on the start panel is clicked, the form in Figure 17 is shown. This is the service order panel. A user can also modify and delete a service and the panels for those functions are similar to the two below.

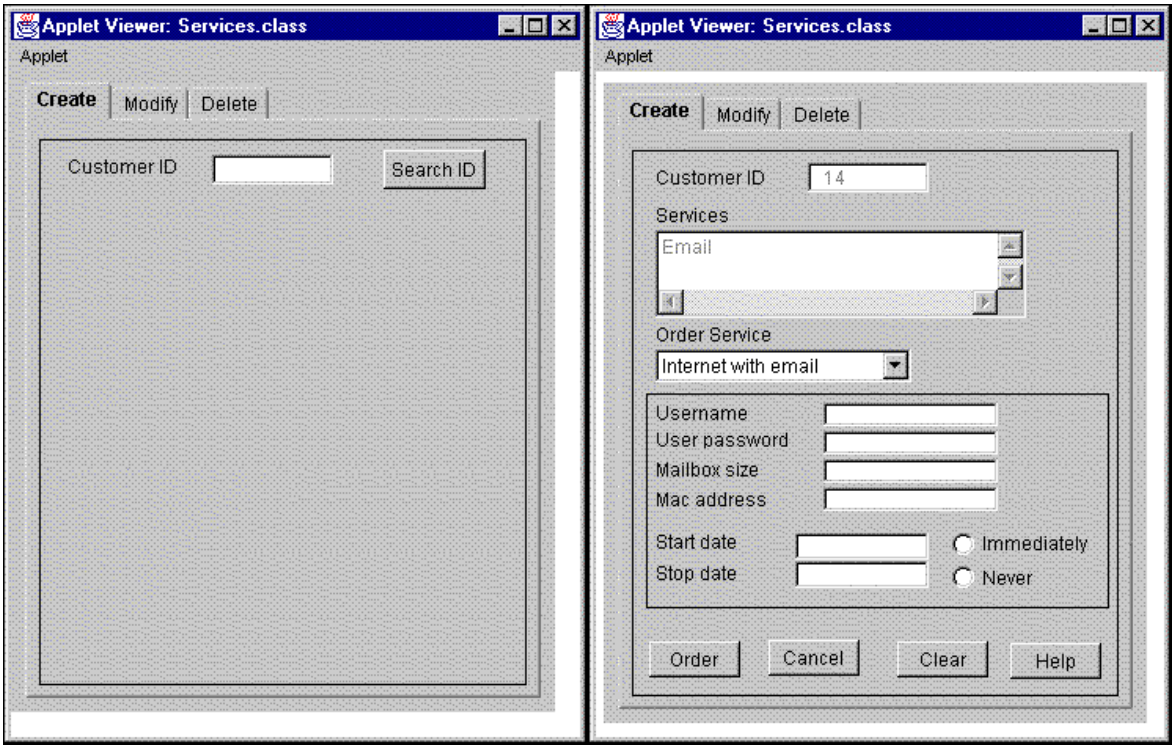

Figure 17. The service order panels.

The form on the right side of Figure 17 is shown when the customer ID number is identified and approved. The user can now watch already applied services in a text area in the applet. He or she can chose a new service to implement from a choice box. When a new service is picked a dynamic border panel appears in the middle of the applet, prompting for required service information. As we mentioned before, the user should recognize some terms, like for example ”mailbox size” or ”mac address”, to understand how to order a service. But, just in case the user does not know the terms, all forms include a Help button containing additional instructions on how to order, modify or delete a service. A user can at any time navigate back to the start panel by pressing the Cancel button.

On the service side, the communication between the client and the server uses CORBA.

4.7 Writing the Server

While the GUI and event handling are on the client side, all other logic relies on the server side. It is the server’s responsibility to take care of all SQL-invocations to the database, to insert, modify and delete customers and services. The server also handles the communication to the Service Configuration application. Writing the server is naturally the biggest part of creating a Self Provisioning prototype.

Because of lack of time, we did not manage to program a pure object oriented server. Instead, the server is one big class including all SQL-statements, communication methods and related functions. Of course, this works just as good as a pure object solution, but we can not say that the source code is beautiful.

The server is connected to a Microsoft Access test database. This is done using Java DataBase Connectivity (JDBC). For simplicity we created the database of our own. It consists of a couple of related tables, but the most used during our tests was a table named Customer shown in Figure 18.

Customer : Tabell

Cus	Type	Customer	Contact	Address1	Address2	VisitAddress	Zip	City	Countr
2	Private	Adam	Eva	Hemma		Gatan	123 45	Göteborg	Sverige
5	Organization	Evert		Vägen 3			222 22	Göteborg	Sverige
6	Organization	Berit	Arne	Vågen 4	Stigen		223 33	Vishy	Sverige

Figure 18. The Customer table in the test database.

4.8 Overview of JDBC

The JDBC API defines Java classes to represent database connections, SQL statements, result sets, database metadata, etc. It allows a Java programmer to issue SQL statements and process the results. JDBC is the primary API for database access in Java[1]. The JDBC API is implemented via a driver manager that can support multiple drivers connecting to different databases. JDBC drivers can either be entirely written in Java so that they can be downloaded as part of an applet, or they can be implemented using native methods to bridge to existing database access libraries.

4.9 Implementing RMI and CORBA

The Self Provisioning Application was implemented in two versions. When clicking the Customer button on the start panel, the RMI version is activated. When clicking the Service button, the CORBA version will execute.

The interfaces made using RMI or CORBA were easily written. There is no need to produce complicated source code in such a small application as the Self Provisioning prototype.

When compiling the server interfaces (we did not write client interface, there was no need for call-backs anyway), we used the rmic compiler for the RMI side and the IDL compiler for the CORBA side. All source files were stored in a package called prototyp.classes. When compiling in Java we used the -d option to move the compiled .class files to a selected directory.

The lookup() method was used to get a reference to a RMI remote object. To get a CORBA object reference we used the bind() method. When getting the reference of a server object, the communication line was served. Now we could let the client invoke methods on the server, just as it would invoke local methods.

Every software developer, no matter how small the application is, must fight against the time to manage to accomplish his or her commission. We could have worked on the prototype for ever, just doing more and more face lifts and adding more functionality, like in a real software development process. But we must remember that the Self Provisioning application is just a prototype. A prototype can not be perfect. The important thing was that we knew that such a system as the Self Provisioning application could easily be implemented. Building the prototype took about 5-6 weeks. We will discuss the overall experience and problems furthermore in section 5.

[1] Jepson, B. (1997)

5 Results

5.1 RMI versus CORBA

5.1.1 Features of RMI and CORBA

In this section, we will cover up some of the differences between RMI and CORBA that were found in literature and materials on the Internet. The differences are concerned with properties of RMI and CORBA and not subjective opinions. The following features are found in various Distributed Object Systems. Much of the material is based on materials from ”Client/Server Programming with Java and CORBA”[1], ”Java and Distributed Object-Systems”[2] and “Providing Easier Access to Remote Objects in Distributed Systems”[3]. Table1, ”CORBA versus RMI,” on page 48 give a summary of all of these following features.

Language Independent (Homogeneous or Heterogeneous)

CORBA was designed for language-independent distributed computing environment where the heterogeneity of the underlying systems is assumed and the objects communication are written in compiled languages.

RMI, on the other hand, was originally designed for single language environment where the objects are running in a homogeneous environment and where new code can be downloaded at any time.

RMI does not provide language-neutral messaging services. In other words, RMI objects can talk to other RMI objects. With RMI, the user can not invoke objects written in other languages.

Licensing

CORBA requires licensing and is not for free. RMI is only available with the Java programming environment. RMI however, is now shipped with the current release of the Java Development Kit, and is therefore available for anyone downloading the JDK.

Platforms

Through Java, RMI has several platforms. CORBA has more than 21.

Registry

This is a feature of the name service. CORBA supports a registry with ongoing persistence, while RMI's registry is restarted every time the server is restarted.

Dynamic Class Download

Systems that provide dynamic class downloading are typically Java based, since binary incompatibilities are not an issue in Java. RMI allows the download of stubs to address a remote object. It should be noted that this term does not refer to the initial download of the applet or component from the web browser.

Interface Definition Language

[1] Orfali, R. et al. (1997) pp 239-268
[2] Jayson, R. (1997)
[3] Aldrich, J. et al. (1997)

This is the mechanism for defining interfaces. The language dependence/independence difference shows in the object model each has. The CORBA object model is defined by IDL, which is a different object model than will be found in any object-oriented language. RMI has no separate interface definition language. The remote interface is defined in Java.

Interface Multiple Inheritance

Multiple inheritance for interfaces, are features that ease the programmers workload. Both CORBA and RMI have this feature.

Maintain State Between Connections

This is a feature that maintains state between connections. If there had not been maintain state there would have been problems; if the programmer has not saved the state object before losing a connection, the objects state will be lost. CORBA and RMI are based on a 'true' Object Oriented Programming model, the object does not need to be explicitly saved, and a connection loss will not result in the loss of the objects state.

Through Firewall

Firewalls can pose problems to distributed object systems, since many are based on TCP/IP connections that are often filtered out of a companies Internet feed for safety's sake. Several mechanism have been put in place to remedy this, including tunneling in HTTP connections (which is considered inefficient) using proxies such as SOCKS, or replacing the firewall with a Distributed Object System friendly one. Iona, for example, has replaced their firewall with such and they call it WonderWall.

Asynchronous Method Calls

Systems supporting this feature allow Clients and/or Servers to call methods on the other without awaiting a response before proceeding.

In the CORBA world the Dynamic Invocation Interface is typically referred to as Message Oriented Middleware, which provides asynchronous messages queues on both the client and the server sides. RMI does not have this feature.

Pass Object by Value

Systems supporting this feature allow an object and it's current state to be passed over the wire to the remote system (Figure 19).

RMI does support this. CORBA does not currently support this, however an Request For Proposal[1] (RFP) exists but it is not implemented. However, CORBA will in the future allow passing primitive types (non-objects) by value.

[1] OMG 1

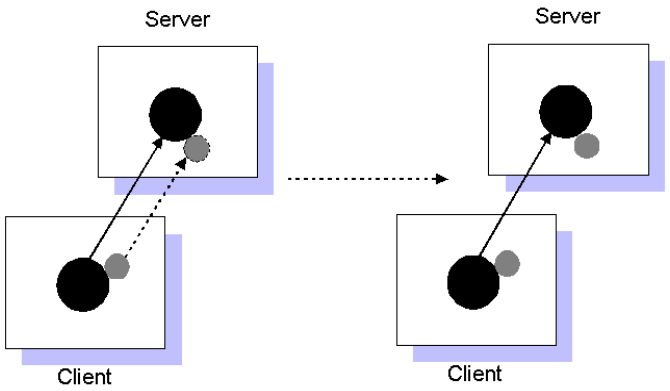

Figure 19. Object pass by value.

Pass Object by Reference

Systems supporting this feature allow a reference to an object to be passed over the wire to the remote system (Figure 20).

Both CORBA and RMI support this feature.

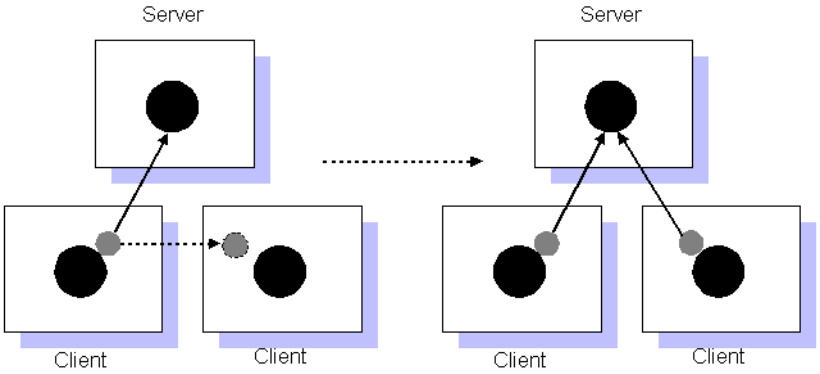

Figure 20. Object pass by reference.

Dynamic Invocation

Systems providing this functionality allow objects to dynamically discover what methods are available on other objects and call them, all without advanced knowledge of the other object. CORBA has Dynamic Invocation via the Interface Repository. RMI does not provide this functionality.

Distributed Garbage Collection

The function of garbage collection is to remove remote objects when they are no longer needed.

In RMI systems garbage collection is handled by remote reference counting, which means that if a client disconnects for a period of time, the object they were connected to could be garbage collected (although an exception does notify the program when this occurs).

In CORBA systems garbage collection is addressed in the Life Cycle Management Services. It basically tracks the number of connections an object has to a client, save and/or delete the object when the number of connections falls to zero.

Performance

On a particular platform and implementation a round trip ping requires 3.3 ms under CORBA's IIOP and 5.5 ms under RMI. The comparison of CORBA to RMI is from Orfali et al., 1997.

5.1.2 Table of the Features

All of the features mentioned above are put together in table 1 below.

Table 1 CORBA versus RMI

	CORBA	**RMI**
Language-independent (homo/hetero)	Yes	No
Licensing	Yes	No
Platforms	>21	Many
Registry	By Name, Persistent	By Name, Not persistent
Dynamic Class Download	No	Stubs Only
Interface Definition Language	IDL	Java
Interface Multiple Inheritance	Yes	Yes
Maintain state between connections	Yes	Yes
Through Firewalls	Via http: WonderWall (Iona)	Via http
Asynchronous Method Calls	Yes	No
Pass Object by Value	No	Yes
Pass Object by Reference	Yes	Yes
Dynamic Invocation	Yes (via IR)	No
Distributed Garbage Collection	No	Yes
Dynamic Discovery	Yes (via IR)	No
Performance	Best	Good

5.2 Interviews

The information we wanted from the interviewed persons, was their view of CORBA and RMI according to common programming experience. The purpose of the first five questions was to get a picture of the programmers background regarding age, education and programming experience. The following questions dealt about how the programmer perceived RMI or CORBA, see Appendix 4.

5.2.1 Age

Among the interviewed CORBA programmers, there were two persons in the interval 20-29 and one person in the interval 30-39.

The age dispersion among the RMI programmers was the same as among the CORBA programmers; two persons in the interval 20-29 and one person in the interval 30-39.

5.2.2 Sex

All of the interviewed persons were men. Because of this it is not possible to draw any conclusion if there are different perceptions among men and women among the interviewed persons.

5.2.3 Education

To learn more about the interviewed persons background, we asked what education they had. Six of the persons was higher educated (more than three years academic education in computer related areas). One of the interviewed person about CORBA was almost educated technical doctor.

5.2.4 Profession and Work Description

The reason why we asked about profession and work description was to get a wider perception about the interviewed persons daily work and overall experience today. The figures below show how long they have been working in the area of software design.

Of the persons interviewed about CORBA, two was pure software designer, and one had been software designer but worked now also as architect technology expert with a leading position.

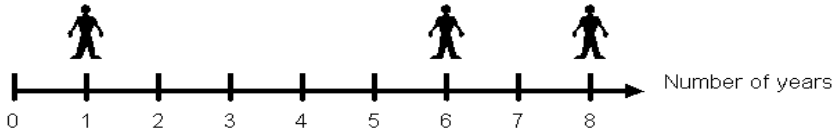

Figure 21. Number of years of total programming experience for the CORBA programmers.

Of the persons interviewed about RMI, one of them was about to finish his academic education. One had been a software designer but was now studying, and one of them had been working as a software designer in two months.

Figure 22. Number of years of total programming experience for the RMI programmers.

5.2.5 Programming Experience

Besides the interviewed persons education and work description, we considered it to be important to ask about experience of object oriented programming on the whole and experience of Java programming. This was important because we wanted to see if experience in these areas could be coupled to the rest of the answers. If persons with more experience consider it easier to use RMI/CORBA or if it did not matter. The classification below is based on the interviewed persons' opinions and our judgment.

Object Oriented Programming Experience on the Whole

Of the persons interviewed about CORBA, the experience of object oriented programming was for example in C++, Java and Smalltalk. The range of object oriented experience for the three persons is shown in the figure below.

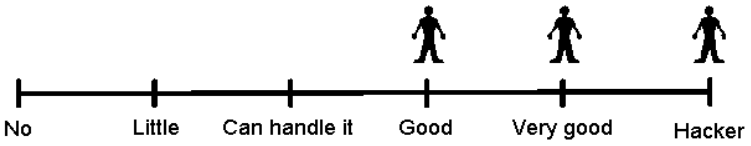

Figure 23. Object Oriented Programming experience.

Of the persons interviewed about RMI the experience of object oriented programming was in Java and C++. The range of experience is shown below in the same way as for CORBA.

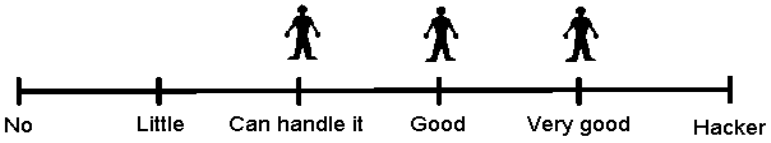

Figure 24. Object Oriented Programming experience.

Programming experience of Java

The range/degree of Java experience for the persons interviewed about CORBA are shown in the figure below.

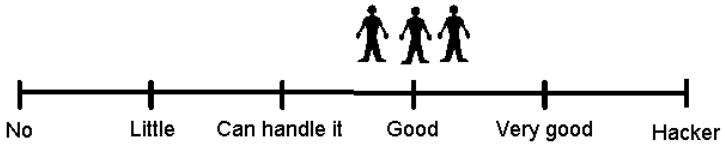

Figure 25. Java experience.

The range/degree of Java experience for the persons interviewed about RMI are shown in the figure below.

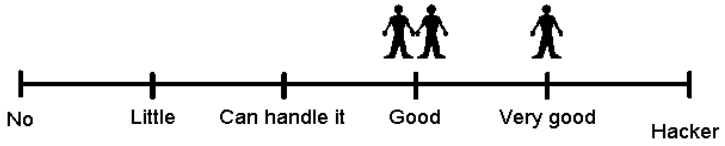

Figure 26. Java experience.

The over all programming experience of the RMI programmers and the CORBA programmers are distinct. The RMI programmers have less experience. The reason for this could be that RMI is still not as common as CORBA in organizations. The RMI programmers that we have

found and interviewed were persons that had been programming either in school or as a part of a thesis. Because of that the RMI programmers have less experience.

5.2.6 Experience of RMI/CORBA

To be sure to get a correct picture of the background for the interviewed persons judgments of RMI and CORBA, we asked how long experience they had of CORBA respectively RMI.

The figures below show how long time of experience that the interviewed persons have in CORBA respectively RMI.

Figure 27. Experience in months of CORBA programming.

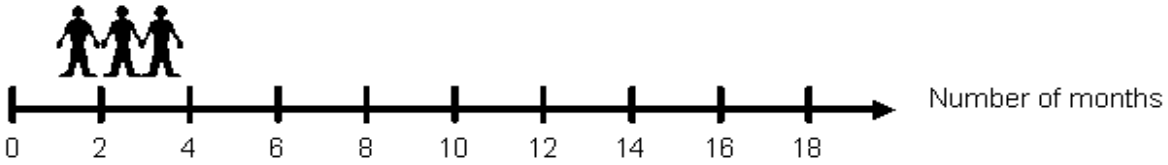

Figure 28. Experience in months of RMI programming.

The figures show that the persons interviewed about RMI only have some few months of experience. This could depend on what we said earlier; that RMI is relatively new compared with CORBA. This should be kept in mind in the following part of this chapter, that the RMI users and the CORBA users are in different experience stages.

5.2.7 The First Experience of CORBA/RMI

One relevant evaluation criteria, is how the first experience of the program was:

The way of learning, experience of the user documentation, the perception of the syntax and installation of the programming environment.

Way of Learning

Both the CORBA programmers and the RMI programmers learned to program by themselves by studying books and materials on the Internet, and not by courses.

Used User Documentation

The user documentation that the CORBA programmers used were the CORBA vendors' user documentation and one person also used the CORBA specification. The latter was used to get a semantic understanding of how the different services were working.

The opinions of the user documentation were diverse. One person thought that it was a drawback that there was no complete example codes for an example application. Two of the persons thought that the user documentation had been helpful enough for their purpose. The opinion about the CORBA specification was though that it was difficult to read and understand.

The user documentation that the RMI programmers used varied from information found on the Internet to user documentation available in different books.

One person thought that the books not were easy to use at all but that the Internet provided excellent information. One person thought that used user documentation was to little help and that the Internet provided good information. The third person did only use documentation found on the Internet. However, the overall impression was that it was easy to find example code on the Internet, that it was simple, easy to understand and had complete code examples.

Perception of Syntax

When it comes to CORBA and the syntax, it must be noted that the programming of the client and server can be done in different languages, it is here only considered the syntax of the IDL, that is the same independent on what language the server or client is written in.

All three of the CORBA programmers thought that the syntax was easy to grasp. It should be noted that the IDL is based on C++, and all the three of them had experience of C++.

One of the RMI programmers thought that it from the beginning was hard to understand the syntax, but that it did not take long time before it felt common, as much was frequently recurring. Two of the RMI programmers thought that the concept of RMI was easy to grasp. One of those two thought that it was possible from the beginning to start on a rather high level. It was not necessary to go through simply examples such as a Hello World before starting programming.

Installation of the Programming Environment

One part of the programming process that could be time wasting, is how difficult or tricky the installation of the program and the programming environment is. With the installation it is here meant the installation of the RMI / CORBA implementation, and with the programming environment it is meant other applications (JDK or Symantec Cafe, Netscape) used when developing Self Provisioning applications.

Two of the CORBA programmers did not consider the installation hard, but they had only installed the CORBA implementation and not the whole environment. One of them had done both and he thought that it was a little bit tricky to set up the environment.

Of the RMI programmers, there was unfortunately no one that had any experience of installation or from setting up the programming environment.

5.2.8 Experience of Compilation Messages

The help functions for the programmer is another important issue, and especially when the programming environment is unfamiliar for the programmer.

One of the CORBA programmers considered the compilation messages cryptic and difficult to understand, while one considered C++ compilation messages much more cryptic, and one person did not perceive the compilation messaged more difficult than other languages compilation messages.

Two of the RMI programmers considered the compilation messages difficult to interpret. One person thought that the compilation messages were rather easy to understand.

5.2.9 Discovered Bugs

Bugs do maybe not determine if a program should be used or not, but it could however try the programmers patience, if the bugs are frequently detected.

Among the CORBA programmers, there was only one bug found.

Among the RMI users there were no bugs found.

Sometimes bugs can be difficult for the programmer to distinguish - it could sometimes be the programmer himself that have done wrong. But some languages could be more bug frequent than others, especially languages that often do new releases. If the interviewed persons would judge, neither RMI nor CORBA are especially bug frequent.

5.2.10 Used Browsers and Appletviewers

Experience of used browsers and appletviewers

We did also want to investigate if the programmers had used browsers or appletviewers and how it was perceived the part of setting up the environment for communication via an applet.

Two of the CORBA programmers had never tried to set up the communication via an applet. One of them had tried to set up the environment for communication and thought it was tricky.

Three of the RMI programmers had used JDK's appletviewer and Netscape 4.04. One of them remarked that it was many problems before it worked. An earlier version of Netscape did not support RMI at all, it was not until Netscape 4.04 that it worked. Netscape seemed to be one step behind the development of RMI.

5.2.11 Experience of the Programming Process

An interesting question in this case was how the development process was perceived, as it looks a little bit different from programming in classical languages as for example C or C++.

One of the CORBA programmers thought that the programming process was easy. One person thought that it was many files to keep track on. And the third person thought that it was difficult to always remember to recompile files and to keep track of which files that had to be recompiled.

The RMI programmer thought that it was difficult from the beginning to keep track of the files. It was easy to compile wrong file as well as forget to compile all files. However one of them thought that keeping track of the files should not be a problem if the programmer was accurate with file structures.

5.2.12 Other Experience of RMI/CORBA, and Pros and Cons

It is also important to catch the programmers overall impressions, if they have perceived any pros or cons or if they have any special requirements.

The pros that the CORBA programmers came up with were that CORBA was platform independent, that CORBA is a standard and not has one special vendor, that it is language independent, that the communication is transparent and the programmer do not have to care about how it works on a low level, that it for example generate the stub and skeleton files automatically and that it is transparent; the programmer do not have to care about if the object is local or external.

The cons that the CORBA programmers had thought about were that CORBA is perceived to be complex. It is for example difficult to know how all the services that are used and it is also difficult to know because it was perceived that some parts are hidden from the programmer. It was also difficult to get different implementations of CORBA to work together

The RMI programmers considered that the pros were that it was easy to use once the program was installed and the environment set up, it was also relatively easy to get useful code examples and advises on the Internet.

The cons were that the RMI programmers came up with was that the programming process from a beginning was perceived intricate, because there were many steps to keep in mind, and that they had to be done in a special order. Another con was that the rmic compiler not was satisfying.

5.3 Frequently Asked Questions, FAQ

The study of FAQ was intended to contribute with material in our work to identify differences between RMI and CORBA from the programmer's point of view. The purpose was to identify and analyze programmers' common problems with RMI respectively CORBA.

As the questions covered a wide scope, from questions concerned with functionality to more complicated programming issues, we decided to divide the questions in different categories:

- General Product information
- Issues concerning the initial part of the programming process
- Basic Programming
- Errors and exceptions

Among all the questions and discussions, we have selected and analyzed the issues that a programmer will have to deal with, from the first encounter of RMI/CORBA and to the programming process. As CORBA is a standard, we decided to choose frequently asked questions about OrbixWeb, one of the implementations of CORBA. Below we will write OrbixWeb and not CORBA when it concerns the OrbixWeb.

General Product Information

The general questions about RMI were fewer than the questions about OrbixWeb. The most frequently question was about licensing concerns. When it comes to OrbixWeb, there were more questions. They dealt about the whole concept of CORBA, the different implementations of CORBA, about IIOP, platforms available and development tools that can be used with OrbixWeb.

Issues Concerning the Initial Part of the Programming Process

The questions about RMI were concerned with Windows 95, versions of Netscape and debugging in Windows environment. The questions about OrbixWeb were concerned with questions about JDK and Symantec Cafe, but also with problems in Windows 95. Other questions about OrbixWeb dealt about settings for OrbixWeb together with Java.

Basic Programming

Questions about RMI were for example concerned with the RMIregistry and the host name of the caller of a remote method and problems with refused connection to the host. In the OrbixWeb case, there were questions about options for the OrbixWeb compiler, the registry function ”orbwebvars.bat”, the codebase parameter and running example code.

Errors and Exceptions

Both for RMI and CORBA, these kinds of questions dealt about strange error messages and exception messages. It was no difference between CORBA and RMI, besides that the name of the errors and exceptions had different names.

5.3.1 Conclusion

We noted that there were generally more questions about CORBA and especially in the part for general information. We think that this could depend on either that CORBA is more complex or that the programmer that have questions about programming with CORBA in a Java environment are not as familiar to Java as the RMI programmers are. It could be possible

that the programmers that choose RMI already have experience of Java and the problems that could occur in the Java environment. We do not know the background of the programmers that have asked the questions, and because of that we can not draw a sure conclusion from this.

Some of the questions did resemble each other. The questions about the initial part of the programming process were especially uniform. With our experience of both CORBA and RMI, we think that almost all of the questions could have been asked about CORBA instead of RMI and vice versa. The initial part of the programming, installing and starting the program is a process that looks much of the same either the programmer uses RMI or CORBA.

The questions about basic programming did also resemble each other, though not as much as the initial part of the programming process. But the pattern of the questions is the same, which could depend on that the programming process for both RMI and CORBA are similar. When it comes to the last group of questions, it did look much the same. The programmers did ask about cryptically messages and work around.

Our opinion is that much of the questions were concerned with the environment that both RMI and CORBA was working in, i.e. platforms, operations system, settings, JDK, Symantec Café etc. The questions that dealt about pure programming, were of course specific, but were following the same pattern and issues.

5.4 Prototyping Experience

Building the Self Provisioning application contributed to a lot more understanding of distributed computing. To learn both RMI and CORBA we decided to implement both technologies in the same application. We were told that RMI was rather easy to understand for beginners. The reason for classifying RMI as easy was that it is a much smaller communication technology than CORBA. Therefore, it was natural to implement RMI first.

Looking back, we know that our plan of learning was good. As we were told; RMI is less comprehensive, in other words; we think CORBA is complex. We must remember that we had never heard of these three-tier communication systems in the beginning of this master thesis project. We think that learning CORBA first had been more complicated. We base this conclusion on the discussion and arguments below.

5.4.1 Documentation

First of all, we have a few things to say about user documentation's and manuals. The RMI Specification[1] contains about 60-70 pages easy read documentation. In contrast to this, the CORBA Security Service specification contains 290 pages of documentation, and is just a "small" part of the CORBA standard specification. The question is: is it worth using CORBA for a less complicated application, even if the solution is smarter? Time is money, and reading documentation of that kind takes time.

Another observation is for example the OrbixWeb user documentation for Java implementing CORBA. The information included in it is of little use for a non-expert programmer. The language is of heavy technical nature and there are very few how-to-begin examples represented. Instead of this we used small how-to-begin examples, found on the Internet to get started with CORBA and RMI.

While CORBA is a standard, much documentation provides pseudo code based programming examples to match several different CORBA vendors. This is, of course, frustrating when a programmer wants to use the documentation as a programming guide.

User documentation and manuals contained lots of errors, which did not make it easier for us when building the Self Provisioning prototype.

5.4.2 JDK Configuration and CLASSPATH Problems

Programming in Java is not as complicated as configuring the environmental settings for JDK. This may sound strange, but the fact is that we consumed a lot of time just trying to get things work with JDK. These problems do not have much to do with RMI or CORBA, but it must be mentioned while it causes time losses. CLASSPATH - the term speaks for it self. It is obvious that it has to do with an environmental class pointer that has one purpose; to find the JDK classes and the user defined classes at run-time. But, this knowledge is not enough if cryptically error messages (i.e. null pointer exception), caused by a wrong-pointed CLASSPATH variable, are thrown during run-time. Not being familiar with pointers (remember, one of Java's greatest properties is that it is a pointer-free language), a programmer will have trouble understanding such an error message. The fact is that, in this case, no errors are generated trough the compilation.

[1] http://java.sun.com

Another problem area concerning the RMI and CORBA configurations, is the host and domain name network settings in Windows 95. Problems occurred specially when running the Self Provisioning application on a stand-alone machine against the local host.

5.4.3 The Compiling Process

This is also an unnecessary complicated property with Java in combination with CORBA or RMI. The RMI written interfaces are compiled using Java's javac compiler, which means that all .java source code (including RMI and ordinary source code) files may be compiled at the same time, using the -depend option. Next step is to run the stub and skeleton compiler on the generated remote .class files, using the rmic compiler. Then it is time to start the RMIregistry through the "start rmiregistry" command (DOS) before the server can be activated using the "java serverName" command. Using CORBA, the remote interfaces must be compiled using the IDL compiler. Finally, the client can be activated using either a web-browser or the Java appletviewer (applets) or the "java className" command (applications).

If changes are made in the programming code, the .java files must be recompiled. If changes are made in the remote interfaces, the compiling process must start from the beginning before the changes are applied, especially if they are made in the objects or the server.

Compiling in Java is a slow process, in spite of the fact that we developed the prototype on a Pentium 133 mHz computer. The time we spent on compiling, was enough time for building another three Self Provisioning applications, if everything had worked out like the manuals promised.

5.4.4 Path Spelling

Following examples shows how specifying paths may differ in the commands mentioned above. These commands are executed on a stand-alone machine not connected to a network. We have only mentioned the DOS way of "spelling" the path, UNIX has its own variants.

```
// Compiling.java files. If the -d option is avoided, the.class files are  generated in the  same
// directory as the source files.
javac c:\prototype\sources\*.java

// Generating stubs and skeletons. The point notation and the space after the back slash are
// very important.
rmic c:\ prototype.classes.SelfProServer

// Binding the server to the registry. Notice the slash after the first colon and the back slash
// after unit c
java -Djava.rmi.server.codebase=file:/c:\some_directories/
prototype.classes.SelfProServer

// Using the Appletviewer to execute an applet. A HTML-file containing the applet tag, must
// be defined. No point notation.
appletviewer file:/c:\some_directories\prototype\classes\SelfPro.html
```

Unfortunately, many manuals or user documentations do not clearly explain how to write paths because of their general pseudo code notation.

5.4.5 The File Structure

It is important, in an early stage, to decide what the file structure, or the packages of the application, will look like. This will facilitate the programming and compiling a lot. A good advice is to separate the source code (.java files) from the class-files and store them in different directories. Distinct problems may occur if the file structure is tangled, for example:

- Compiling of wrong .java files. The application uses old .class filer. No visible changes when running the application.
- Forgetting to run the rmic compiler (RMI only) on remote interfaces. The application uses old stubs and skeletons.
- When compiling, the .class files are generated in a different location than the user think. Causes a lot of wondering.

5.4.6 Syntax

There is no problem understanding the CORBA IDL syntax, while it is similar to Java syntax. RMI is written in Java, which makes it quiet easy to define remote interfaces. The big difference between CORBA and RMI syntax is that RMI does not support in, out and inout parameters like CORBA do. The reason for this is that local objects are passed by copy and remote objects are passed by reference to a stub. For more details, see Parameter Passing in RMI Specification[1].

5.4.7 Java DataBase Connectivity, JDBC

The most frequent syntax error message we got when programming with JDBC was ”To few parameters in query string”. This means that the number of the parameters passing within an SQL-query, was less or more than the number of columns in a database table.

The configuration of JDBC may be implemented through the ODBC (Open DataBase Connection) Data Source Administrator shipped with Windows 95. This is not a tricky procedure.

5.4.8 Web Browsers

RMI is a Java package included in JDK 1.1 and most browsers (until autumn 1997) did just support JDK 1.02. This means that an applet works fine until a communication process initializes. Then the Java Virtual Machine (JVM) throws a run-time exception. The reason for this is that the web-browser does not understand or support RMI classes. Netscape Communicator version 4.04 (version 4.03 with an upgrading patch) truly supports JDK 1.1 and our Self Provisioning application seamed to execute well when testing.

5.4.9 JDK Versions

The problem of old-fashioned web-browsers is these days just a question of time. We think that everyone soon will have a JDK 1.1 supporting web-browser, with which to communicate across the Internet. Java is still an immature programming language, through this investigation period, a several upgrading version of the JDK were released. From the beginning, June 1997, until now, January 1998, we have dealt with JDK 1.02, 1.1.1-1.1.5 and recently (in December 1997) the JDK 1.2 Beta 2 was released. We do not believe that there are big differences in

[1] http://java.sun.com

these versions, but it is a bit confusing when new releases appear every month. The same problem as we experienced with web-browser affects the visual tools for Java programming (i.e. Symantec Visual Café Pro).

5.4.10 Symantec Visual Café Pro

Symantec[1] proudly assures that the source code automatically updates changes to the visual view of Visual Café Pro. Unfortunately, this is not quiet true. Visual Café Pro has actually trouble with updating changes, that was manually written in the source code. This means that although new code was inserted in the source or something was just modified, no changes are made in the visual view. In run-mode, the program acts as expected, but after inserting or changing some source code manually, the user will be forced to continue programming manually.

Another disadvantage of Symantec Visual Café Pro is that it is rather difficult to program pure Java applets or applications. Symantec has invented a lot of special packages and classes of their own. When running programs developed in Symantec Visual Café Pro, all used packages and classes must be available to the Virtual Machine.

There are lots of bugs in the product. We used the Symantec Visual Café Pro 1.0e version, most of the time when developing the Self Provisioning GUI. Comparing to, for example Microsoft Visual Basic, the difference is big. Visual Café Pro is a young programming environment and has a long way to go to become a stabile system as Visual Basic.

A big advantage though is that no extra work must be done to locate the generated .class files before running an applet or application. Visual Café Pro handles this by it self. This is great, it can be rather complicated to scan for reasons to mystical error messages when trying to run a Java program from a DOS or Unix shell environment.

Although there are many complications and problems with Visual Café Pro, it is in fact one of the hottest Java tools on the market today. By using Symantec packages and classes the user can build great applets or applications and save plenty of time. It is also easy to understand the functionaries of the program and to scan error messages.

5.4.11 Compatibility

Though RMI is a powerful mechanism, it does suffer from a few limitations. The first of these is that RMI, though similar to CORBA, is not compatible with CORBA. Therefore, RMI-based applications and CORBA-based applications can not directly communicate with each other.

5.4.12 Time Consuming

Phillips[2] has identified seven main stages in the process of conducting a master thesis report like this. These are enthusiasm, isolation, increasing interest, increasing independence, boredom, frustration and a job to be finished. We agree that we actually adopted some of these stages during the prototype development process, especially the frustration condition. Most of the time we dealt with different kind of troubles mainly related to JDK settings. Building a client / server solution using Java in combination with CORBA and/or RMI for the first time, is really a time consuming process. The analysis of the interview persons' experience showed that

[1] http://www.symantec.com
[2] Phillips, E.M. (1984) pp 6-18

most of the RMI and CORBA programming time was devoted to pure troubleshooting activity. We completely agree with this.

5.4.13 Level of Learning

Learning to implement these technologies from a low level of complexity (small easy understood examples), is quiet more reasonable from a beginners point of view. In opposite to this statement, one of our interview persons (the person has never used CORBA) claimed that RMI is easy to learn from a higher level of difficulty (complex examples). He also stated that because the transparency of this kind of communication technologies, a programmer does not have to understand exactly what is happening. This might be correct, but we may not draw any conclusions from this since CORBA has the same advantages.

5.4.14 Conclusion

Because of the syntax simplicity of the communication technologies, it is not difficult to learn how to use them, most of the problems arises in the environment.

Choosing RMI as a communication solution does not mean that CORBA is an uninteresting technology. The positive thing is that CORBA can be integrated as well. Therefor, we think that there actually is no ”choose between” these technologies. When building a small application, for example an applet communicating with a small server, why not use RMI for its simplicity’s sake?

6 Discussion

The main purpose of this master thesis was to evaluate the technologies of RMI and CORBA in the context of distributed computing. On our way to the conclusion of this investigation, we have learned a lot about distributed and object oriented system development through Java, RMI, CORBA and JDBC. What is left for us to do is to give the reader of this paper an overview of what he or she should think of when programming a Self Provisioning system.

Making an architectural decision of a Self Provisioning application that may be used in a commercial purpose can be a question of careful consideration.

6.1 Properties of the Application and the Environment

One important issue is weather the application will be a large (complex) or a small (simple) one. It is of importance as different size and degree of complexity could demand different technologies. CORBA is required if the planned system is complex.

It is also important to know if the environment the developed system will run in is heterogeneous or homogenous. If the environment is a network composed of distinct platforms, operating systems and applications the answer is heterogeneous, and the obvious solution is to use a CORBA implemented system. This is true also if different parts of the system will be programmed in different languages. CORBA was, until recently, the only distributed communication technology that really worked out within most web-browsers. We experienced that in, for example, Netscape Navigator 3.X, only the CORBA-side communication of our Self Provisioning application functioned as required.

Despite the versatility of CORBA, it may not be the perfect solution for every application, which shares objects across a network. A primary reason for this is that commercial CORBA-based products tend to be expensive. For a small application developer, the costs associated with using CORBA can be prohibitive, as commercial CORBA products are currently targeted towards enterprise-wide applications for businesses.

The Java Development Kit, however, is free and can quickly be downloaded from Sun.

6.2 The Choice of CORBA or/and RMI

The user documentation and information on how to use CORBA and RMI differ, but the programming process is resembling, there are the same problems with the environment set up and there are approximately the same action to do when programming a client and a server.

The choice of technology is more important and dependent of how the programmer is doing when the programming starts to become more complex. If it takes more time to search for help in manuals and in the specification than it takes for the programmer to come up with something him or herself - then the technology is not a productive one. CORBA is claimed to be more complex than RMI. For example, the RMI specification does only cover less than 100 pages, while the CORBA standard cover a lot more documentation. The documentation for CORBA is arranged in different documents and types of documents and the volume is vast.

6.3 We Suggest

CORBA is great because of the multi-language support. It is great when it comes to transporting data across the net and probably it is the best medium to transfer data across the

net portable and efficiently. But it lacks the ability to move true objects across the net. CORBA goes a long way towards making the network a computer but not all the way.

RMI on the other hand goes all the way. It allows the ability to move true objects across the net, but it is limited when it comes to legacy systems. With RMI the network really becomes a computer. It really opens up possibilities when it comes to theory and idealistic computing.

The ideal system, to us, seem to be something which does CORBA over the Internet and on the front/back end, which might contain a distributed environment of its own, does RMI.

We suggest: Do RMI when programming Java-to-Java across for example a Local Area Network (LAN), like an Intranet, and CORBA when computing across a Wide Area Network (WAN), like the Internet.

6.4 Future

Though RMI is an ORB in the generic sense that it supports making method invocations on remote objects, it is not a CORBA-compliant ORB. The OMG has moved rapidly to incorporate support for Java into its architecture. In addition to specifying a standard language mapping from IDL to Java, the OMG is developing specifications for automatically generating IDL from Java interfaces. The OMG is in the process of extending the CORBA specification to support passing objects by value, similar to the Java RMI mechanism for passing objects by value.

Sun has recently, in December 1997, released the JDKTM 1.2 Beta 2 version of Java which include IDL-to-Java mapping features provided through the Java IDL package.

CORBA has great value to offer to the Java world-primarily, an open standard integration model and direct interoperability with CORBA based on the IIOP protocol. In the next version of JDK (version 1.2, probably released in the second quarter), Java RMI will be implemented on top of IIOP which will make RMI CORBA-compliant and give it access to multi-language environments. The result for Java will be the great gain of transparent interoperability with the most powerful, widely adopted distributed computing infrastructure existing today.

The OMG membership recognize the importance of Java as a programming platform for distributed object technology, and work proceeds apace to merge the two technologies (Java and CORBA) to produce a combined capability whose whole is more powerful than the sum of its parts.

6.5 Conclusion

CORBA and RMI are not competing systems; they are complementary ones. They each suit different needs, and which to use boils down to what the requirements are. The most basic requirement is whether the application will act in a homogeneous or heterogeneous environment.

Systems made up of components in languages other than Java demand CORBA. Systems, which must be deployed partially in Java, may use a hybrid of RMI interoperating with CORBA. The hybrid solution makes sense if the delivery window is after IIOP support has been added to RMI and there is a need within the system of Java-to-Java distributed computing in addition to interoperability with other languages.

Even in the 100% pure Java scenario, there is a place for CORBA. RMI is specifically not capable of handling very massively distributed systems. It is at this time missing some of the

key services that CORBA supports. These services such as transaction management[1] are a requirement of very large distributed object solutions. RMI will support the CORBA services in the future and eventually be a viable large scale solution, but it is not there today.

[1] http://www.omg.org

7 References

7.1 Books

Andersen, Erling S. (1994) ***Systemutveckling - principer, metoder och tekniker***, (Andra upplagan). Oslo: Studentlitteratur.

Befring, Edvard.(1994). ***Forskningsmetodik och statistik***, Lund: Studentlitteratur.

Booch, Grady (1994), ***Object-oriented analysis and design with applications*** (Second Edition), Addison-Wesley Publishing Company Ltd.

Esterby-Smith, Mark, Thorpe, Richard, and Lowe, Andy. (1991***), Management Research An Introduction***, SAGE Publications Ltd.

IONA Technologies Ltd. (1996), ***Orbix Web Programming Guide***, IONA Technologies Ltd.

IONA Technologies Ltd. (1996), ***Orbix Web Reference Guide***, IONA Technologies Ltd.

Jepson, B. (1997), ***Java Database Programming***, John Wiley & Sons Inc.

Orfali, R. and Harkey, D. (1997), ***Client/Server Programming with Java and CORBA***, John Wiley & Sons Inc.

Patel, Runa & Davidson, Bo (1994). ***Forskningsmetodikens grunder, Att planera, genomföra och rapportera en undersökning***. Lund: Studentlitteratur.

Rubenowitz, Sigvard (1980). ***Utrednings- och forskningsmetodik***. Göteborg: Esselte Studium.

Rusty, Harold Elliote (1996), ***Java Network Programming***, O'Reilly & Associates Inc.

Sommerville, I. (1996), ***Software Engineering***, Fourth Edithion, Addison-Wesley, Reading MA.

Sridharan P. (1997), ***Advanced Java Networking***, Prentice Hall Inc.

7.2 Online Articles

Aldrich, Jonathan (1997**). *Providing Easier Access to Remote Objects in Distributed Systems***, Pasadena: California Institute of Technology.
URL: http://csvax.cs.caltech.edu/~adam/jedi/paper/

Fingar , Peter & Stikeleather Jim (1996). ***Distributed objects for business. Getting started with the next generation of computing***, Sunworld (1996),
URL: http://www.sun.com/sunworldonline/swol-04.1996/swol-04-oobook.htm

Gilbert, S.J. & Landercy, A. (1997). ***Distributed Object Computing***, Université de Mons-Hainaut,
URL: http://colos01.info.fundp.ac.be

de **J**ager, Nico (1996). ***A Report on the Context of CORBA***,
URL: www.cs.up.ac.za/hsn/docs/report/NDJ96a.html

Jayson, Raymond (1997). ***Java and Distributed Object-Systems***, Corbis, Corp,
URL: http://www.seajug.org/html/DisObSys

Keahey, Kate (1997). ***A Brief Tutorial on CORBA***,
URL: http://www.cs.indiana.edu/hyplan/kksiazek/tuto.html - A Brief Tutorial on CORBA

Kollars, Chuck (1997). ***Distributed Object Systems for Computing***,
URL: http://www1.shore.net/~ckollars/distobjs.html#further

OMG1. ***OMA Executive Overview***,
URL: http://www.omg.org/about/omaov.html

Schmidt, C. Douglas (1997). ***Overview of CORBA***,
URL: http://www.cs.wustl.edu/~schmidt/corba-overview.html

7.3 Important Sites

Internet Smartsec	URL: http://www.smartsec.se
Object Management Group	URL: http://www.omg.org
Sun Microsystems	URL: http://java.sun.com, http://www.javasoft.com,
Symantec	URL: http://www.symantec.com

8 Appendix 1 - The RMI Development Process

To implement a complete RMI system a programmer must follow a step-by-step process. The process contains ten different steps as shown below. This example is based on Orfali's[1] suggestions when designing a client / server system using RMI. At the end of this appendix a figure is illustrating the process.

1. Define a remote interface

A server object must declare its services via a remote interface. It does this by extending the java.rmi.Remote interface. The interface must use the java.rmi.RemoteException so each method in the interface can throw a java.rmi.RemoteException. The sole purpose of the remote interface is to tag remote objects so they can be identified. A remote object may have many public methods, but only those declared in a remote interface can be invoked remotely[2]. The example code below is a simple interface for a ”Hello World” remote object. It contains a single method, sayHello(), which returns a string.

```
import java.rmi.*;
public interface HelloInterface extends Remote
          {
              public String sayHello() throws java.rmi.RemoteException;
          }
```

2. Implement the remote interface

Next step is to define a class that implements the remote interface. This class must be derived from java.rmi.UnicastRemoteObject, either directly or indirectly. The HelloImpl class implements the remote interface Hello and extends UnicastRemoteObject.

The HelloImpl class

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public class HelloImpl extends UnicastRemoteObject implements
HelloInterface
{
  public HelloImpl() throws RemoteException
   { super();
   }
      public String sayHello() throws RemoteException
       {
        return ”Hello World!”;
       }
      public static void main (String args[])
       {
       try
          {
           HelloImpl hImpl = new HelloImpl();
           naming.rebind(”Hello”, hImpl);
           System.out.println(”Hello Implementation ready!”);
          }
       catch (RemoteException re)
          {
           System.out.println(”Error in HelloImpl.main: ” + re);
          }
```

[1] Orfali, R. et al. (1997) pp 242-244
[2] Rusty, H.E. (1996) pp 355

```
        }
}
```

The main() method in the example above is specially used in Java applications. Java applets use the init() method instead.

3. Compile the server class

The server class is compiled using javac. The following command generates a .class-file. In Java this is the runable file type.

```
prompt> javac classname.java
```

The created interface above is also compiled by javac before the next step in the developing process.

4. Run the stub and skeleton compiler

RMI provides a stub compiler called rmic. To generate stubs and server skeletons for the remote classes a programmer must run the rmic against the .class files. Like their CORBA counterparts, RMI stubs provide client proxies for the server objects. They marshal remote calls and send them to the server. The RMI server skeleton receives the remote call, unmarshals the parameters, and then calls the implementation class. Following command executes the rmic compiler:

```
prompt> rmic classname.class          // The extension .class is optional
```

This generates classname_stub.class and classname_skeleton.class files. These are the stubs and skeletons used in communication between a client and a server.

The rmic stub compiler takes the same command line argument as javac. For example, the specification of the classes can be specified using the CLASSPATH environment variable or via the -classpath command line argument. The compiled class files are placed in the current directory unless a different location using the -d argument is specified.

5. Start the RMI registry on the server

RMI defines interfaces for a nonpersistent naming service called the Registry. RMI has a remote object implementation of this registry. It allows to retriever and register server objects using simple names. Each server process can support its own registry, or there can be a single stand alone registry that supports all the virtual machines on a server node. Entering the following command at the prompt can start the registry object on the server:

```
dos-prompt> start rmiregistry
unix-prompt> rmiregistry &
```

The registry is by default listening to port 1099. If this port is used by some other program during the runtime, an exception will occur. To avoid this a registry can be run on a different port. This is done by appending a port number like this:

```
dos-prompt> start rmiregistry 2048
unix-prompt> rmiregistry 2048 &
```

The registry database is empty when the server first comes up (or boots). A programmer must populate it with the remote objects.

6. Start the server objects

The server classes must be loaded and the instances of remote objects created. This is done by the following command.

```
dos-prompt> java -
Djava.rmi.server.codebase=http://a.domain.name.suffix/some/directory/
HelloImpl

unix-prompt>java -
Djava.rmi.server.codebase=http://a.domain.name.suffix/some/directory/  \
HelloImpl &
```

7. Register the remote objects with the registry

The instances of all remote objects must be registered with the RMI registry so clients can reach them. The methods of the java.rmi.Naming class are used to bind a name to the server object. This class uses the underlying Registry to store the names. The server objects are now ready to be invoked by clients.

8. Write the client code

The client consists mostly of ordinary Java code. It can be an applet or an application. Before a client can call a remote method, it needs to retrieve a remote reference to the remote object. A program retrieves a remote reference by asking the registry on the server for a remote object. It asks by calling the registry's lookup(String url) method.

```
Object o1 =
Naming.lookup(”rmi://a.domain.name.suffix/some_directory/Hello”);
Object o2 =
Naming.lookup(”rmi://a.domain.name.suffix:2048/some_directory/Hello”);
```

The object that is retrieved from a registry loses its type information. Therefore, before using the object it must be cast to the remote interface that the remote object implements. This is simply done by using the following argument:

```
HelloInterface hInterface = (HelloInterface) Naming.lookup(”Hello”);
```

Once the object has been retrieved and its type restored, the client can call the object's remote methods exactly as it would call methods in a local object:

```
String message = hInterface.sayHello();
```

Next example is a simple client for the HelloInterface interface of the last section.

The Hello client.

```
import java.rmi.*;
public class HelloClient
{
     public static void main(String args[])
           {
           System.setSecurityManager(new RMISecurityManager());
           try {
                HelloInterface hInterface = (HelloInterface)
                Naming.lookup(”Hello”);
                String message = hInterface.sayHello();
                System.out.println(”HelloClient: ” + message);
               }
           catch (Exception e)
               {
                System.out.println(”Exception in main: ” + e);
               }
           }
}
```

The System.setSecurityManager(new RMISecurityManager()) invocation above is required for all distributed applets. Without the RMISecurityManager installed, applets can not run, and if they could they would be a potential security risk to machines that download them.

9. Compile the client code

This is done using javac.

```
prompt>javac HelloClient.java
```

10. Start the client

Finally, the client can be run. If the client is an applet, a web-browser or an alternative appletviewer can be used to run it. This means that a HTML-page must be written. If the program is a stand-alone application, there is no need to write HTML-code.

Now this small client/system is complete. Figure 29 shows the step-by-step process described.

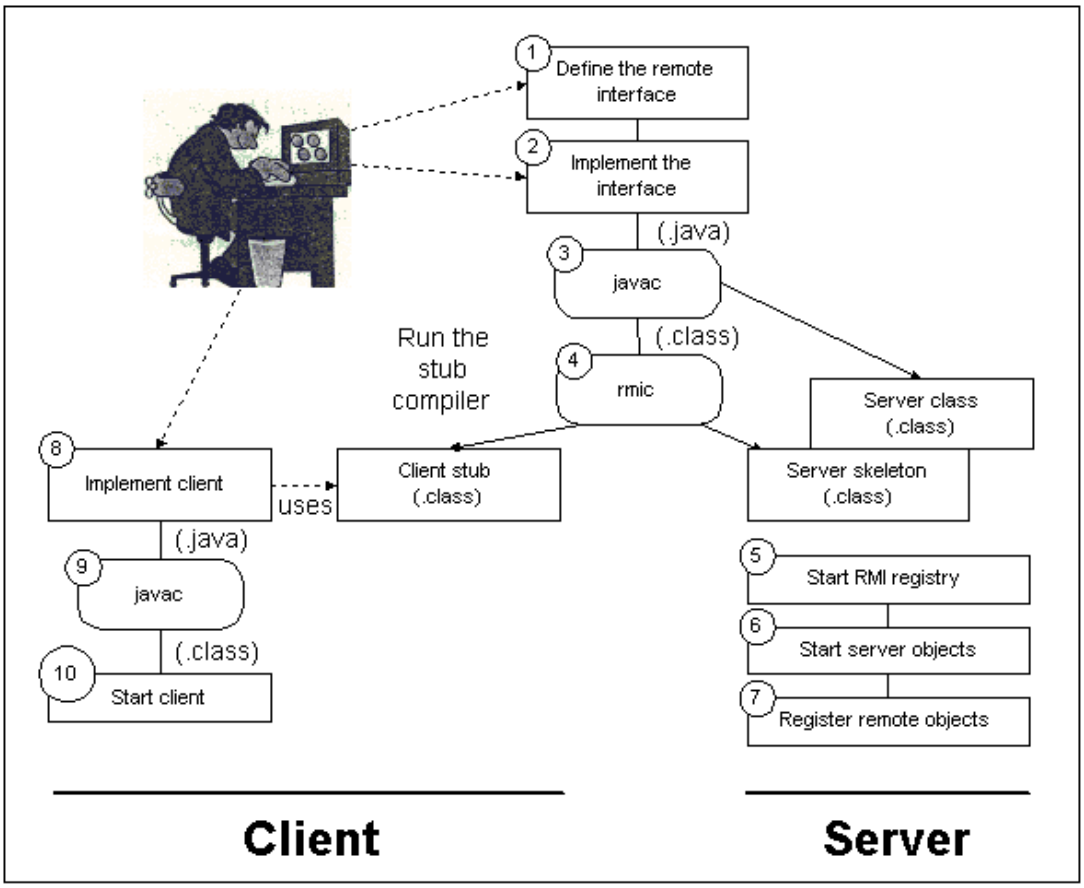

Figure 29. The step-by-step process in writing an RMI client and server application.

9 Appendix 2 - The CORBA Development Process

As CORBA is a standard specification, there are several different vendors and ORBs. However, the creations of an application with a CORBA compliant implementation do, as the RMI development process, normally follow a similar step-by-step development process[1]. The development process is depicted in Figure 30 at the end of this appendix.

As in the RMI example in Appendix 1, it will described a development process for a Hello World example. The example code is mainly written in Iona's OrbixWeb[2].

1. Define an Interface in the IDL

The first major step in developing CORBA-based applications is to write the IDL describing the interfaces and data types used by the applications. The definition of the interfaces and data types for the application must be done early in the development process, while the application depend on the code generated from the IDL. Each time the IDL is changed the code must be re-generated and updated in the application code. The following example IDL describes a CORBA object whose single sayHello() operation returns a string.

```
// HelloWorld.idl
module HelloWorld
{
    interface SayHello
        {
        string hello();
        };
};
```

2. Compile the IDL

Before writing the client and server applications, the IDL code must be compiled. The compilation must be done both to check the specification and to map it into the target programming language. The compilation is done with a "idl to java compiler" that does the necessary mapping from idl to Java.

```
Prompt> idl HelloWorld.idl
```

There are differences in the code generated by different CORBA-compliant compilers, but it is typically capable of generating at least two types of output files:

1 The Client stubs for the IDL defined methods - these stubs are invoked by a client program that needs to statically access IDL-defined services via the ORB

2 Server skeletons that call the methods on the server - they are also called up-call interfaces.

The automatic generation of stubs frees developers from having to write them, and frees applications from dependencies on a particular ORB implementation.

Iona's OrbixWeb creates seven different files. Among them three will be mentioned here: _SayHello.java, _SayHelloHolder.java and _SayHelloRef.java.

When running the idl to Java compiler, the module HelloWorld is mapped to a Java package of the same name, and the corresponding directory is created. The IDL interface SayHello is

[1] Orfali, R. et al. (1997) pp 65-87
[2] Iona Technologies Ltd. (1996)

mapped to a Java interface _SayHelloRef. The stub code, or client-side proxy, is implemented by the Java class _SayHello. The class _SayHelloHolder deals with inout and out parameters.

3. Object Implementation

The next step is to implement the object whose interface that has been specified in the IDL. Object implementation classes must be associated with the skeleton class generated by the IDL compiler. This can be done by inheritance.

In the example below there is an implementation class SayHelloImpl that extends the skeleton class (_boaimpl_HelloWorld).

SayHelloImpl.java

```
import CORBA.*;

class SayHelloImpl extends _boaimpl_HelloWorld.
{
    // method
    public String hello() throws CORBA.SystemException
     {
        return "Hello World ";
     }
}
```

The method hello() is implemented and it returns a string composed of the message "Hello World".

Next step is to compile the implementation file.

```
prompt> javac SayHelloImpl.java
```

4. Write the server application

Every CORBA server must have some kind of main program that initializes the ORB environment and start the objects. A server class must be written to implement that main function. In addition, the server must provide implementations of the CORBA interfaces that are defined in the IDL. In its simplest form, the class that provides the main function on the server side has to do the following:

1 Initialize the ORB

2 Initialize the Basic Object Adapter (BOA)

3 Create Implementation objects

4 Notify the BOA of the existence of the object.

5 Wait for incoming requests

The ORB initialize by calling ORB.init() which returns a reference to the ORB object. With this reference the method BOA_init() is invoked which initializes the BOA and returns a reference to it.

To create an implementation object Say_Hello_impl we call Java's new operator and supply one parameter for the constructor, which we copy from the command-line argument.

The Hello Server

```
import CORBA.*;

public class HelloWorldServer
{
    public static void main(String[] args)
  {
           try
          {
           //init ORB
           CORBA.ORB orb = CORBA.ORB.init();

           //init Basic Object Adapter
           CORBA.BOA boa = orb.BOA_init();

           // create a SayHello object
           SayHelloImpl _good_day_impl = new SayHelloImpl( args[0]);

           // export the object reference
           boa.obj_is_ready( _good_day_impl);

           // print stringified object reference
           System.out.println(orb.object_to_string( _good_day_impl));

           // wait for requests
           boa.impl_is_ready();
        }
        catch(CORBA.SystemException e) {
           System.err.println(e);
        }
    }
}
```

Once the implementation object is created, the BOA can be notified that this object is available by calling the method obj_is_ready(). The stringified version of the object reference is printed out and is obtained by calling object_to_string() on the ORB. (This is the object reference used in the Java-application client to establish a connection with a server).

The BOA is notified, by calling impl_is_ready(), that the server is ready and that it can receive requests from clients. Finally, any CORBA system exceptions are caught and handled.

5. Compile the server

The server class is compiled using javac. The following command generates a .class-file. In Java this is the runable file type.

```
prompt> javac HelloWorldServer.java
```

6. Start the Server

Next step is to start the server

```
prompt> java HelloWorldServer
```

7. Write the client application

A client implementation follows these steps:

1 Initialize the CORBA environment, that is, initialize the ORB.

2 Obtain an object reference for the object on which it wants to invoke operations.

3 Invoke operations and process the results.

Initializing the ORB

This is done to define a Java class, HelloWorldClient and define the main() method for this class

Obtaining an Object Reference

References to objects can be obtained by various means. Here is an unsophisticated method used. An object reference can be made persistent by converting it into a string Stringified object references are re-convertible into "live" object references. This is done using the two corresponding operations object_to_string() and string_to_object().

```
// get object reference from command-line argument
IE.Iona.Orbix2.CORBA.Object.Ref obj = _CORBA.Orbix.string_to_object
(args[0]);
```

In this example, it is assumed that a stringified object reference is provided as the first argument to the client program. It is then provided as the argument to the method string_to_object(), which is invoked on the ORB pseudo-object. The method returns an object reference of type CORBA::Object, the base type of all CORBA objects. To make use of the object it needs to be narrowed to the appropriate type. Narrowing is equivalent to down-casting in some object-oriented programming languages.

OrbixWeb generates the narrow method _narrow() in the class HelloWorld.SayHello. The Java interface, which corresponds to the IDL interface, is HelloWorld._SayHelloRef. (Leading underscores are generally used to prevent name conflicts with user-defined type names because IDL syntax does not allow leading underscores.)

```
// and narrow it to HelloWorld.SayHello
HelloWorld._SayHelloRef good_day = HelloWorld.SayHello._narrow (obj);
```

Invoking the Operation

Once the ORB is initialized and an object reference is obtained, CORBA programming looks very much like standard object-oriented programming. The programmer invokes methods on objects and it looks exactly the same for remote and local objects.

```
// invoke the operation and print the result
System.out.println(good_day.hello());
```

The client invokes the method hello() on the object good_day and the result is printed to standard output.

The HelloClient.

```
import java.io.*;
import IE.Iona.Orbix2._CORBA;
import IE.Iona.Orbix2.CORBA.SystemException;

public class HelloWorldClient
{
    public static void main(String args[]) {
      try
      {
            System.out.println( "string_to_object");
            IE.Iona.Orbix2.CORBA.Object.Ref obj =
_CORBA.Orbix.string_to_object ( args[0]);
            System.out.println( "narrow");
            HelloWorld._SayHelloRef good_day = HelloWorld.SayHello._narrow
(obj);
            // invoke the operation and print the result
            System.out.println( "invoke");
            System.out.println( good_day.hello());
        }
        // catch CORBA system exceptions
        catch (SystemException ex)
       {
            System.err.println(ex);
        }
    }
}
```

8. Compile the client

To make the client program executable by a Java virtual machine it needs to be compiled. This is done by calling the Java compiler.

```
prompt> javac HelloWorldClient.java
```

9. Start the client

Finally, the client can be run. If the client is an applet, a web-browser or an alternative appletviewer can be used to run it. This means that a HTML-page must be written. If the program is a stand-alone application, there is no need to write HTML-code.

Figure 30 shows the step-by-step process described.

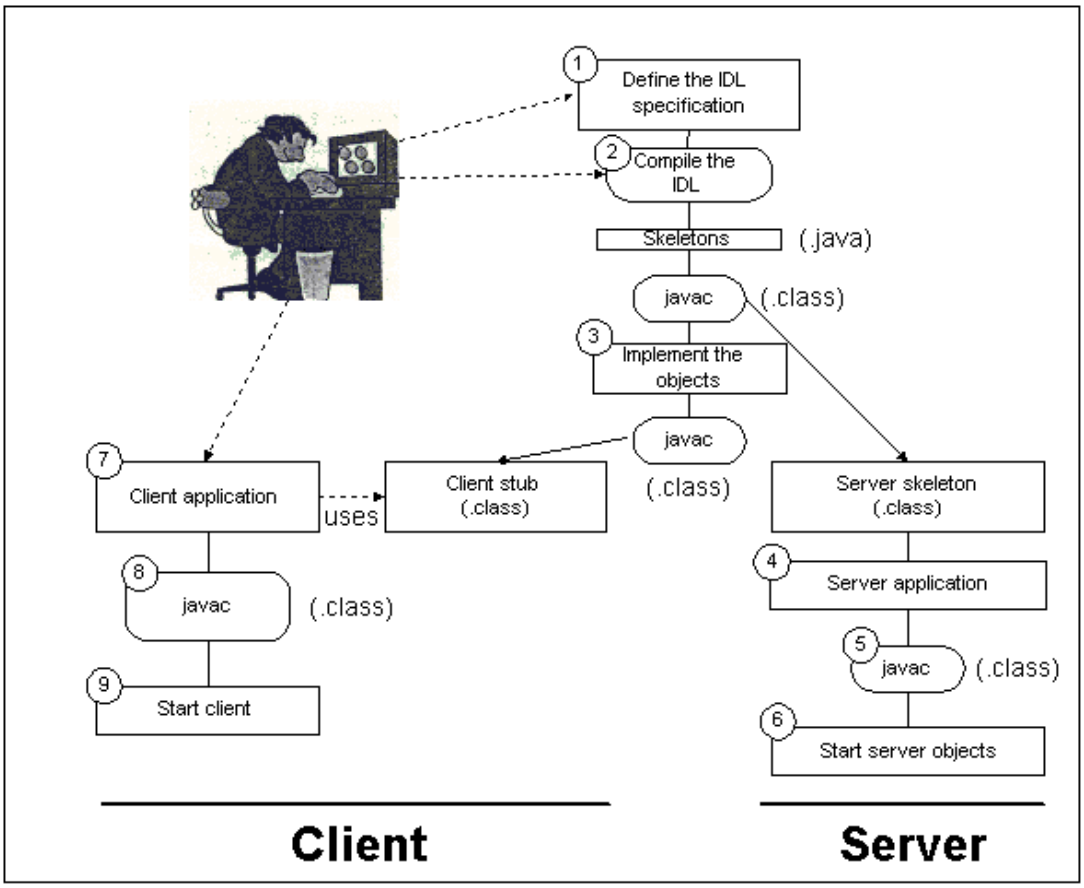

Figure 30. The step-by-step process in writing an CORBA client and server application

10 Appendix 3 - The Prototype Specification

Implementing a Self Provisioning Application

Abstract

This document describes a master thesis project, which will implement a Self Provisioning application.

Introduction

miniCustomerCare (mCC) is an application that will be used by the operators. Some operators want to reduce their costs by implementing an application, through which their customers would be able to order distinct services.

This kind of application could be Self Provisioning (SP). The application shall execute within a web-browser and communicate with an application at the operators' company. The operators' application registers and activates the requested services. The SP application shall be designed according to an open architectural principle. Owing to this, Java, JDBC and CORBA technologies should be applied.

Self Provisioning

The application may be implemented with Suns Java WorkShop or Symantec Visual Café Pro[1]. The JDBC connection may be implemented with PowerSoft (Sybase) Jato or Symantec dbANYWHERE. Suns JOE, OrbixWeb (Java-CORBA connection) or Orbix MT may be used to design the SP and the Service Configuration (SC) applications' interfaces. The Orbix products are from Iona.

Cost

Java Workshop and Jato may freely be evaluated for 30 days (can be downloaded from the www-sites of Sun or PowerSoft), Orbix MT and OrbixWeb are available at Ericsson Telecom AB. So is also Symantec Visual Café Pro and the dbANYWHERE product is included.

Instructions during 3 months will take about 40 hours.

[1] http://www.symantec.com

11 Appendix 4 - Plan for the Interviews

The Problem area

How do the interviewed person perceive the programming environment of CORBA respectively RMI? Is the programming environment easy to adopt and use?

The purpose of the interviews is to survey information about how programmers in general perceive technologies as RMI and CORBA, used together with Java and to investigate problems that the programmer usually encounter during the programming process.

The Interviews content and scope

The content of the interview is demarcated of every persons frame of reference.

Type of interview

The form of the interview was an open interview.

How the interviews are performed

The interviews are done in an informal way and with only the interviewed person present. Every interview may approximately take one hour.

The questions are not in advance exactly formulated questions. The purpose is to ask questions that are adapted and suitable for everyone and every situation. The questions are instead based on a number of problem areas or areas of questions. Those are included below.

The first questions however have a character of straight questions with straight answers.

Information to the interviewed persons

- The interviews are done to be used for the purpose to be an integral part of a thesis at ”Institutionen för Informatik” at Gothenburg university.
- The purpose of the interview is to get a survey of how a programmer perceives the programming environment.
- The interview will be documented with notes and tape recordings.
- The interview will take approximately an hour.
- The interview will be treated anonymously, but the analysis of the interview will be a part of the thesis, that is an public paper.
- The results of the interviews will be a contribution to the judgement of a programmers perception of RMI and CORBA’s programming environment.

Question areas

1. Sex

2. Age

3. Education

4. Profession and work description

5. Programming experience

- Programming experience on the whole
- Object oriented programming experience on the whole
- Programming experience of Java

6. Experience of RMI/CORBA

7. The first experience of CORBA/RMI

- Way of learning
- Used user documentation
- Perception of syntax

8. Discovered bugs

9. Installation of the programming environment

10. Experience of compilation and compilation messages

11. Used browsers and appletviewers

- Experience of used browsers and appletviewers

12. Experience of the programming process

13. Other experience of RMI/CORBA, and pros and cons