Master's Thesis in Computer Science

Preliminary version
August th29, 2001

GEM Security Adaption

Karin Almstedt
The Royal Institute of Technology
Kungliga Tekniska Högskolan

Examiner: Prof. Seif Haridi
Department of Microelectronics and Information Technology
The Royal Institute of Technology

Supervisor: Vladimir Vlassov
Department of Microelectronics and Information Technology
The Royal Institute of Technology

Johan Petersson
AU-System AB

Abstract

Distributed applications provide an opportunity to establish and maintain a competitive advantage by creating a flexible IT infrastructure. That brings, however, new requirements on distributed business applications. They must be able to work on a variety of hardware and software platforms. Users and programs should also be able to dynamically join and leave the network, and discover each other. It should also be possible to have the same naming conventions to locate any resource on the network. Since 1989, the Object Management Group (OMG) has been busy specifying the architecture for an open software bus on which objects written by different vendors can interoperate across network and operating systems.

Common Object Request Broker Architecture (Corba) is a specification defined by OMG. The Corba specification describes a software bus, called an Object Request Broker (ORB) that provides an infrastructure for distributed object computing.

The company AU-System has created a Java component library, Generic Element Manager (GEM), to be able to easily create a system for supervision of network elements. GEM is using Java RMI for the remote communication between programs. But as Java RMI can only be used for object written in Java, AU-System wanted to migrate GEM to the more general Corba and also improve GEM by using Corba over Secure Socket Level (SSL) as the session security model.

Security is very important in distributed systems and the second part of the Master thesis is to design a new security model for authentication and authorisation.

In this Master thesis the migrating of GEM turned out well. Test of the modified version showed that it contains the same functionality as the previous version. The extra facility is that GEM is more general now, which was a requirement from customers to AU Systems.

Summary of contents

1 Introduction 9
2 Overview of GEM: Generic Element Manager 10
3 Distributed objects 18
4 Security in distributed objects 46
5 Migration of GEM from Java RMI to Corba/SSL 50
6 Design of GEM Security Model 59
7 Conclusions and Future work 72

Contents

1 Introduction ... *9*

1.1 Background ... **9**

1.2 The objective of the Master thesis ... **9**

2 Overview of GEM: Generic Element Manager ... *10*

2.1 Consultant unit TM/D ... **10**

2.2 Project GEM ... **10**

2.3 Overview of the elements: NE, EM, NEM ... **10**

2.4 Technical solutions in GEM ... **12**

2.4.1 Challenging design issues ... 12
2.4.2 Definition of a component in GEM ... 12
2.4.3 Functions ... 12
2.4.4 Network Adapter ... 15
2.4.5 Requirements for using GEM ... 15
2.4.6 The GEM library ... 15
2.4.7 User Roles in GEM ... 16
2.4.8 Start-up and synchronisation ... 17

3 Distributed objects ... *18*

3.1 Definition of a distributed object ... **18**

3.2 Overview of different approaches to distributed computing ... **19**

3.3 RMI: Remote Method Invocation ... **20**

3.4 Corba Technology ... **22**

3.4.1 Overview of Corba, ORB and OMG ... 22
3.4.2 Interfaces, IDL and stubs and skeletons ... 24
3.4.3 Interface Repository (IR) ... 27
3.4.4 Object Adapter ... 28
3.4.5 Object Implementation and Object Reference ... 28
3.4.6 Static and Dynamic Invocation ... 28
3.4.7 Standard object interfaces ... 29
3.4.8 Interoperability ... 33
3.4.9 The Portable Object Adapter (POA) ... 36
3.4.10 Tie Mechanism ... 40

3.5 Corba compared with Java RMI ... **43**

4 Security in distributed objects ... *46*

4.1 Threats in a distributed object system ... **46**

4.2 Overview of security in distributed objects ... **46**

4.2.1 Low level ... 46
4.2.2 High level ... 46

4.3 Security features ... **47**

4.3.1 Authentication ... 47
4.3.2 Privacy ... 47
4.3.3 Integrity ... 47
4.3.4 Authorisation ... 47

4.4 Public-key encryption ... **47**

4.5 Auditing ... **48**

4.6 Investigation of Corba/SSL products....48
4.6.1 VisiBroker SSL package....48

5 Migration of GEM from Java RMI to Corba/SSL.... 50

5.1 General method of porting....50

5.2 The Analyse Phase....50
5.2.1 Client-Server communication in GEM....50
5.2.2 GEM Clients....51
5.2.3 GEM Server....52

5.3 Design and Implementation....53
5.3.1 The Corba Development Process....53
5.3.2 Write and generate IDL files....53
5.3.3 Using Name Service....56
5.3.4 Using the Tie-mechanism....57
5.3.5 Using Callback....57
5.3.6 SSL Security Service Package....57

5.4 Test procedure of GEM with Corba/SSL....57

6 Design of GEM Security Model.... 59

6.1 Security Management....59

6.2 Design processes and tools....59
6.2.1 Rational Unified Process....59
6.2.2 Rational Rose....60

6.3 Design phases....60

6.4 The existing GEM Security Model....61

6.5 Security policies....62
6.5.1 Access policy architecture....62

6.6 Use cases and scenarios in GEM Security Model....63
6.6.1 A representative use case....64

6.7 Iteration plan....67

6.8 Relation Database Design....67
6.8.1 Relation Model Concepts....67
6.8.2 Conceptual Schema....68
6.8.3 Relation Database Schema....70

7 Conclusions and Future work.... 72

List of Figures

Figure 1. Overview of a Network Management system. 11

Figure 2. The GEM Fault Management main window. 13

Figure 3. The GEM Security Management main window. 13

Figure 4. The GEM SA main window - Cleared Alarms Log. 14

Figure 5. Functional overview. 15

Figure 6. System overview. 16

Figure 7. The different user roles. A SecurityAdm is allowed to do the same things as a SystemAdm and an Operator. A SystemAdm the same things as an Operator, but an Operator can only have the role as an Operator. 17

Figure 8. Java RMI distributed application. 21

Figure 9. An overview of Corba. An IDL interface acts as a contract between developers of objects and the users of their interfaces. 22

Figure 10. Corba IDL bindings provide client/server interoperability. 25

Figure 11. Stub, skeleton and ORB. 26

Figure 12. Generated files after compiling an IDL-file (using VisiBroker 4.1). 26

Figure 13. Example of an IDL-file, Common.idl. 27

Figure 14. BankAccount.idl. 27

Figure 15. Corba Architecture. 29

Figure 16. Naming scheme for an order entry system. 32

Figure 17. Binding, resolving, and using an object name from a naming context within a namespace. 32

Figure 18. Event channel between Consumer and Supplier. The channel acts as an intermediate object. 33

Figure 19. Interoperability via ORB-to-ORB communication. 34

Figure 20. Structure of the Corba Interoperability Specification. Shaded components are mandatory for Corba Interoperability compliance. Interoperability Architecture contains the basic architecture built on bridging; the Interoperable Object Reference (IOR); the interoperability interfaces including the DSI; and the provision for context-specific services. 35

Figure 21. IIOP is GIOP over TCP/IP. 35

Figure 22. Overview of the POA. 36

Figure 23. Servant manager function. 37

Figure 24. The FaultManagement interface in the IDL file Interfaces.idl. Just a part of the file is shown. 38

Figure 25. The server implementation of the code. Note: This shows just the instantiation of FaultManagement part of the code. 39

Figure 26. The client implementation of the code. Note, only the implementation of FaultManagement is shown. 40

Figure 27. Single class inheritance. 41

Figure 28. A schematic picture of the Tie-mechanism. 41

Figure 29. The implementation class only needs to implement <InterfaceName>Operations and can still inherit another class. 42

Figure 30. Changes in the server code example in Figure 25 when using Tie-mechanism. 42

Figure 31. Server side, using RMI. LocateRegistry.createRegistry creates and exports a Registry on the local host that accepts requests on the specified port (1099). Naming.rebind rebinds the specified name to a new remote object. Any existing binding for the name is replaced. 43

Figure 32. Client side, using RMI. Naming.lookup returns a reference, a stub, for the remote object associated with the specified name (rmiHost). 44

Figure 33. Server side, using Corba. 44

Figure 34. Client side, using Corba. 45

Figure 35. Public and private keys performing inverse function of one another. 48

Figure 36. Client-Server communication 51

Figure 37. A part of the new IDL file CommonDeclare.idl. 54

Figure 38. The Server side is using an Alarm object and the parameters in the object receive its data from the database (db). Corba is used between the Client and the Server. When sending alarm information between the Client and the Server is it necessary to place the object parameters in a struct CorbaAlarm that is defined in the file CommonDeclare.idl. 54

Figure 39. Help class, GetCorbaAlarm, to pass parameters from Alarm to CorbaAlarm. 55

Figure 40. Process Structure – Lifecycle Phases 59

Figure 41. Use-case model. 60

Figure 42. Use case diagram for Configure Network Policy. 64

Figure 43. Example of Configure Network Policy dialog. 66

Figure 44. Example of Copy Network Policy dialog. 66

Figure 45. Conceptual schema for the GEM Security model. 69

Figure 46. Graphic view of some tables. 71

List of Tables

Table 1. Use cases for GEM Security Model. ... 63

Table 2. The iteration plan for design of security model use cases. ... 67

Table 3. Table with table name UserMembership and column names GroupName and UserName ... 67

Table 4. Description for every object in the conceptual schema ... 70

1 Introduction

1.1 Background

The next generation of client/server systems will most certain be built using distributed objects. The explosive growth of the Web, the increasing popularity of PCs and the advances in high-speed network access have brought distributed computing into the main stream.

The middleware technology Corba [6], among others, simplifies distributed systems in many ways. The distributed environment is defined using an object-oriented paradigm that hides all differences between programming languages, operating systems, and process locations. Corba was invented to provide a new networking infrastructure intended to solve software interoperability problems. It replaces legacy remote procedure call (RPC) infrastructures and precursory technologies by providing an object-oriented network interface that greatly simplifies distributed computing.

The company AU-System [www.ausystem.se] has consciously built up specialist expertise for the development of Operations Support Systems (OSS), in other words, systems that are used to configure, govern and monitor the supplier's products. AU-System primarily develops so-called Element Managers and Network Managers for equipment suppliers. Their solutions are adaptable, follow prevailing standards and are modular and able to be integrated into a complex operating environment with an operator. This guarantees a solution that is adaptable to the future and has greater acceptance among operators.

Their solutions are platform independent with reusable architecture, own developed components and the market's best third-party products. They have for example developed a Java component library, General Element Manager (GEM) that are used for supervision (O&M).

1.2 The objective of the Master thesis

AU System wanted to improve GEM by

- change from Sun’s Java RMI to the more general Corba, for remote communication
- use Corba over SSL as the session security model
- design a new security model for authentication and authorisation.

The Master thesis consists of two main parts: the first part to migrate from Java RMI to Corba/SSL in the existing Java component library (GEM) and the second part is to design a new security model for authentication and authorisation in GEM. Other “smaller” parts included in the Master thesis is to decide which Corba/SSL product to use and to study the technology used in GEM and learn the procedures used at AU-System, such as RUP (Rational Unified Process) [7]. In Appendix A is an overview of the products and technologies to study in the Master thesis.
The Master thesis should result in a modified GEM.

2 Overview of GEM: Generic Element Manager

AU-System has created a Java component library, Generic Element Manager (GEM), to be able to easily create a system for supervision of network elements [1].

2.1 Consultant unit TM/D

TM/D (Telecom Management/Development) is a consultant unit at AU-System. Its main enterprise is development and consultant in the area of supervision of network elements and networks in the telecommunication sector.

2.2 Project GEM

The trend within NEM (Network Element Manager) [1] for telecommunication and data communication is, as in other lines of business, to start using web-based solutions. There are great interests for tailor-made web-based NEM solutions among the supplier of telecommunication and data communication equipment. As there are lots of similarities between the demanded systems, TM/D realised that it could be an advantage to create a library with NEM components to be able to fast and easily create similar systems.

TM/D started the project GEM and the two main goals are to deliver ordered NEM functions to customer X, and to make components of the developed code to add to the NEM component library.

The supervision system should be web-based and developed in Java. During development the RUP[7] with UML (Unified Modelling Language)[8] and Rational Rose[9] should be used.

2.3 Overview of the elements: NE, EM, NEM

This paragraph will give a brief overview of Network Elements (NE), Element Manager (EM) and Network Element Manager (NEM).

Equipment providers must provide well-functioning, comprehensive operations and maintenance solutions for their products [25]. These solutions are often called *Element Managers* (EM) because they are so specific to the equipment provider's network equipment. The term *Network Element Manager* (NEM) is often used in a broader sense to denote the entire range of functionality for configuring and operating an equipment provider's product. This often means element and sub-network management functionality (i.e. the ability to manage the capabilities provided by interconnected network elements are also included), see Figure 1. GEM contains some of the functionality in NEM, such as supervision of status (alarm handling).

A *Network Element* (NE) is a minor physical part that has a task in a network. A network element can for example be a router, base station or a radio link.

An *Element Manager* (EM) is a system for supervision of network elements. An EM system is used for

- installation and configuration
- supervision of status and performace

- operations and tests etc. of network elements

Element Managers normally address the entire FCAPS[1] scope (Fault, Configuration, Account, Performance and Security Management) to the extent that is applicable to the services provided by the networking equipment. This means Element Managers have to provide functionality for service assurance, service provisioning, planning and accounting.

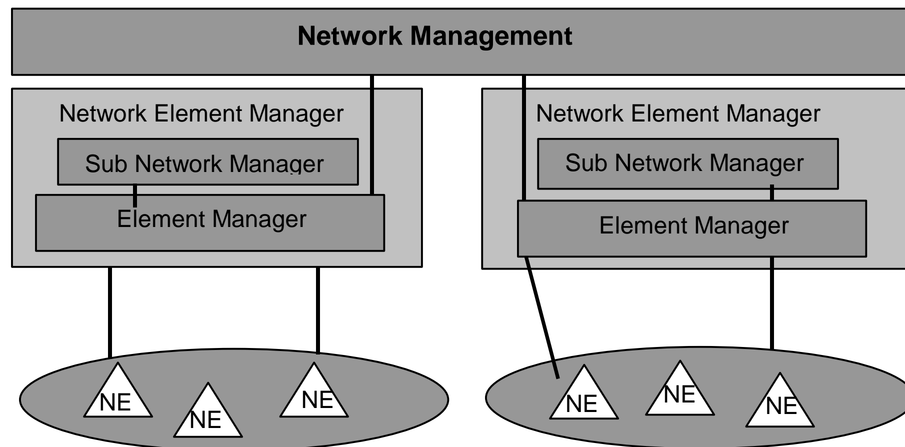

Figure 1. Overview of a Network Management system.

A *Network Element Management* (NEM) system can be divided into two parts:

- *Product specific applications* that are strong connected to the type of network element to be used and can not be used for other types of network elements. Typical functions are configuration, test and diagnostic.

- *Generic applications* that can handle network elements from different vendors. They can be used as stand-alone applications or be integrated with other applications.
  The generic applications can further be divided into:
  Technology dependent applications, that are restricted to a specific technical area (e.g. SDH or ATM). This kind of application exists for example within Performance Management.
  Technology independent applications, that can handle different kinds of NE within several technical areas. One example is Fault Management applications.

A technology dependent generic application can have similarities, which makes it suitable to do some parts of the functionality generic with an adaptation to the respective technology area.

It is important that developers and customers have the same definitions about a Network Element Management. Therefore, the applications in GEM shall follow the ITU standard for Network Element Management [1].

[1] The FCAPS model is a contribution from the TMN (Telecommunications Management Network) model M.3100 by ITU. FCAPS is the division of management functionality into a number of functional areas.

2.4 Technical solutions in GEM

This section gives a short overview of the technical solutions in GEM.

2.4.1 Challenging design issues

During the design phase of GEM the biggest challenge was to decide which Java packages that should define components and to decide the interface of the packages. It was also difficult to follow a specific alarm standard as it turned out that the customer had a different point of view of what an alarm was.

2.4.2 Definition of a component in GEM

In the GEM project it is decided that a component will have the following definition:

A component is a *java package* and a package consists of several compiled Java classes and Java interfaces. Each package has a unique name and according to the convention, a package name shall precede with the company's Internet domain name, backwards. The whole component family is named GEM, so the Fault Management component has the unique name: *se.ausys.gem.fm*. This package can have sub packages, such as *se.ausys.gem.fm.gui* for the user interface and *se.ausys.gem.fm.server* for the server program.

The user of a component imports the package to his/her Java code and uses the included methods via the exported interfaces.

2.4.3 Functions

The main functions in GEM are Fault Management, Security Management and System Administration.

Fault Management (FM)
The GEM Fault Management comprises:

- the handling of alarms and notifications related to hardware and software faults
- logging of alarms
- localisation of faulty replaceable units
- correlation between different alarms

Alarms and clear operations arrive to the server from a network element, via an adapter, see 2.4.4. The server updates the database as well as the view on all clients to inform the operators.
The GEM Fault Management provides a user interface that consists of a tree view with two main nodes, one for alarms and one for network elements (NE), see Figure 2[2]. In the alarm node the alarms are grouped after the measure status: open, acknowledge or cleared. Alarms are even presented in a list, in chronology order. By using a filter the user can select what kind of alarms to be shown. In the network element node alarms are presented per network elements and are grouped per severity. Changes in alarm status made by any operator are reflected on all clients.

[2] [3], figure 7.

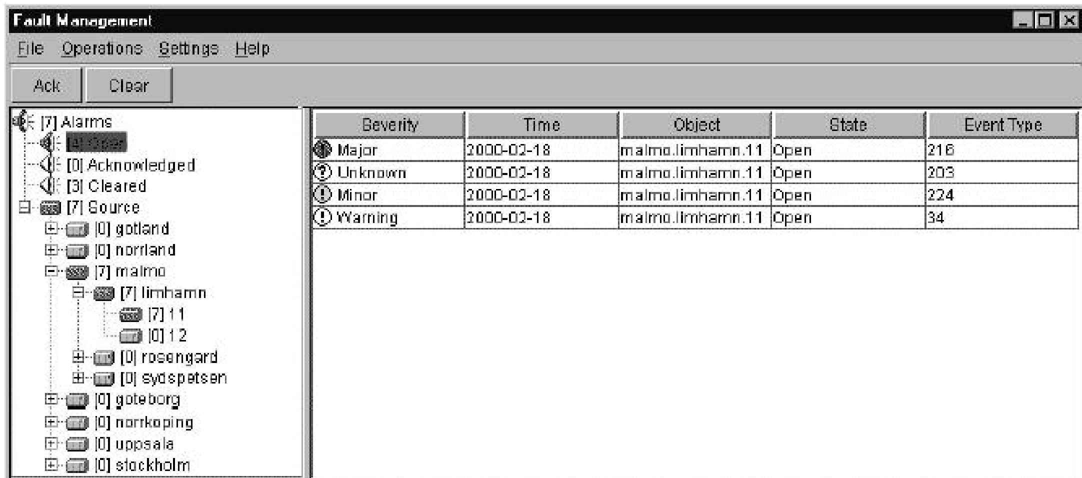

Figure 2. The GEM Fault Management main window.

Security Management (SM)

The GEM Security Management comprises the handling of access restrictions to management operations. It also provides the possibility to define user profiles and maintains a log of events related to security, see Figure 3[3]. All events are logged by the SM function.

The GEM Security Management contains functions to administrate users and passwords, which is stored in cryptic form in the database. A user is connected to a role: operator, security administrator or system administrator, and role is allowed to do certain operations. In the new security model, part two of the Master thesis, it will even be possible to connect the user to one or several group/s, and a group is allowed to do certain operations.

By using/contacting GEM Security Management, other applications can control if a user is qualified for doing a certain operation. Note, not implemented yet. Implementation is not a part of the Master thesis.

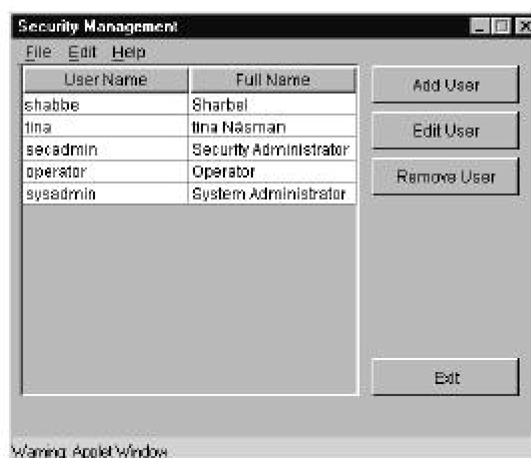

Figure 3. The GEM Security Management main window.

[3][3], figure 12.

System Administration (SA)
The GEM System Administration comprises the supervision and configuration of the system, see Figure 4[4]. The GEM System Administrator controls the server side functionality.

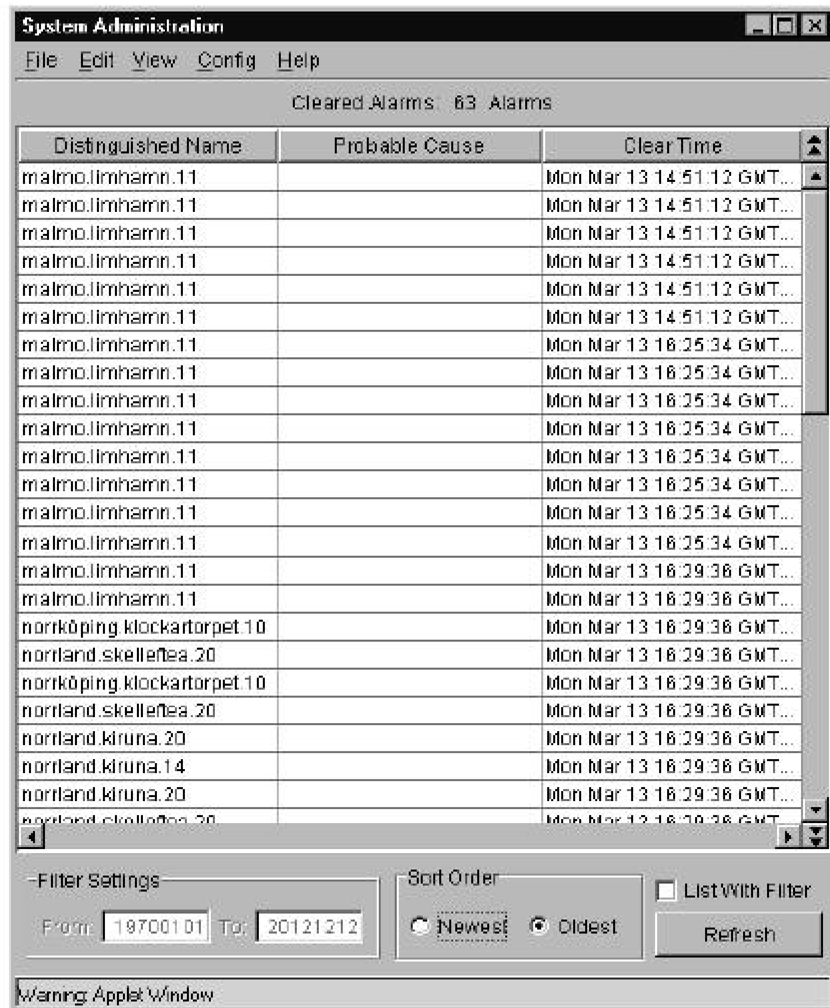

Figure 4. The GEM SA main window - Cleared Alarms Log.

Note. Only the GEM Fault Management supports change reflection to other clients.

In Figure 5[5], the three main functional areas of the GEM are present. The purpose is that other functional components easily can be incorporated if needed, e.g. Configuration Management.
The broken line surrounding the client and server components represents a customer specific "container" for the desired functionality. Such a container is not a part of GEM, rather is it designed to requirements of each customer.

The communication between client and server is by using a distributed interface (e.g. Java RMI).

---

[4] [3], figure 14.
[5] [2], page 6.

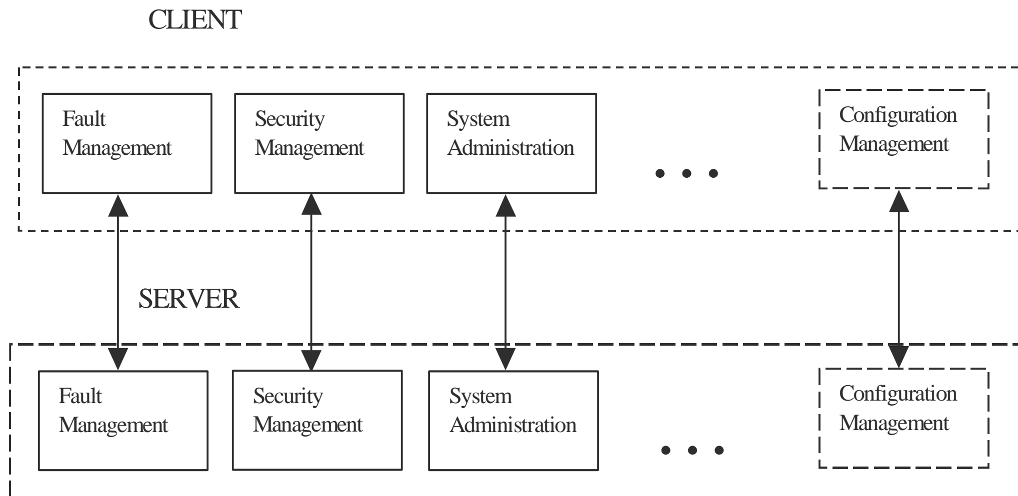

Figure 5. Functional overview.

2.4.4 Network Adapter

The *Network Adapter* is not considered to be a part of GEM but a part of the customer/network element specific part of an element manager system based on GEM [2]. The Network Adapter is responsible for polling the network elements, generating internal alarms when contact is lost with network elements and synchronise the alarm list in the network elements with the alarm database in GEM when the system is started and when contact is re-established with a network element.

The Network Adapter is also responsible for receiving alarms from the network elements and to convert them to the internal GEM alarm format and forward them to the GEM server.

2.4.5 Requirements for using GEM

The requirement for using GEM is that the protocol between the Network Element and the Network Adapter is known, see Figure 6. It is also necessary to do an adapter for every protocol.

2.4.6 The GEM library

The GEM System constitutes of functional components that will be used in a framework designed for a specific system. That means that the components will be reusable in a new context for a new system. The GEM focuses on the functional components, not the framework.
All components in the GEM library are presented in Figure 6[6]. The GEM database holds a list of network elements associated with the network equipment. It also holds alarms, users, logs etc.

[6] [2], page 7.

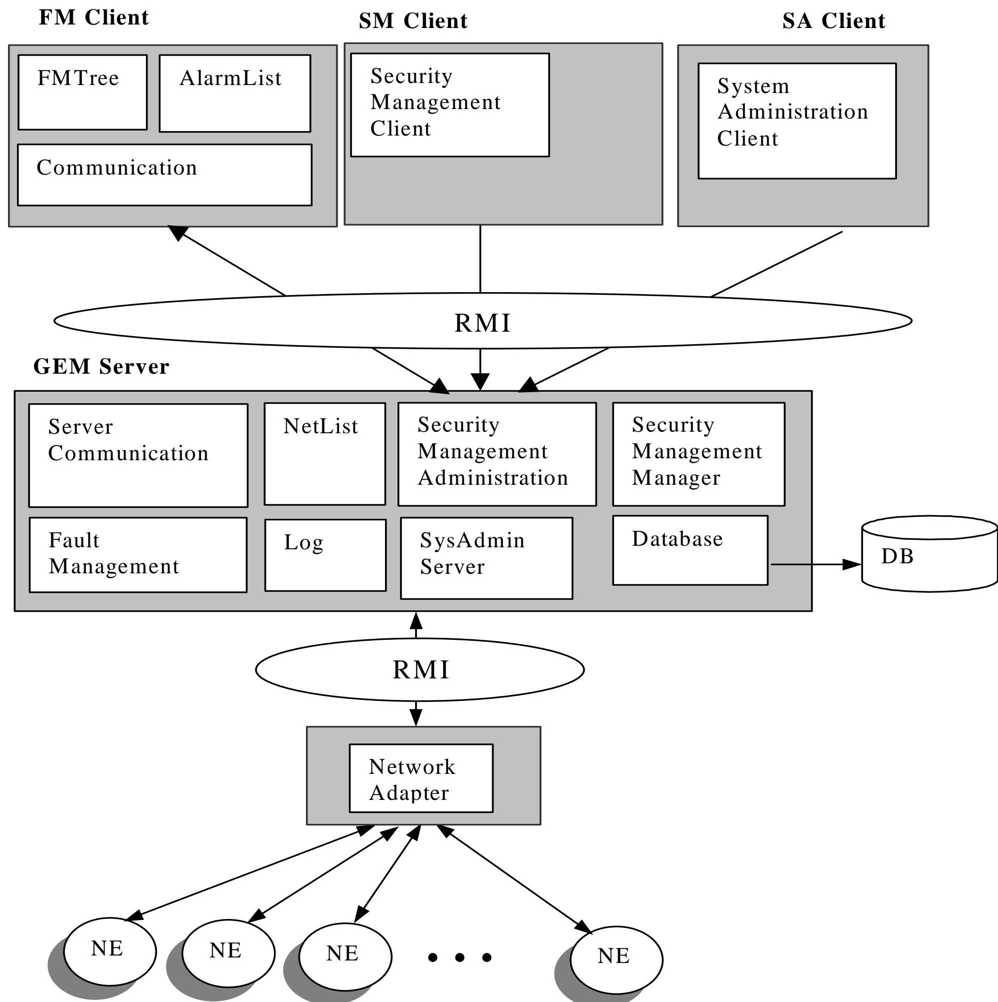

Figure 6. System overview.

GEM source code is packaged with the generic code and the customer specific code separated. The generic code consists of 133 Java files and the customer specific code of 6 files (in a specific project).

2.4.7 User Roles in GEM

There are three roles defined for the GEM user [3]: operator, system administrator and security administrator, see Figure 7.

The *Operator* works in the Fault Management (FM) area and may acknowledge and clear alarms.

The *System Administrator* works in the System Administrations (SA) area and may

- add and remove network elements
- maintain the application log
- perform some server configuration

The System Administrator also has the right to work with Fault Management.

The *Security Administrator* works in the Security Management (SM) area to add, edit, block, activate and remove user.
The Security Administrator also has the right to work with Fault Management and System Administration. Only the Security Administrator is allowed to maintain the security log in the System Administration area.

Note. A user can only be associated with one role at the time.

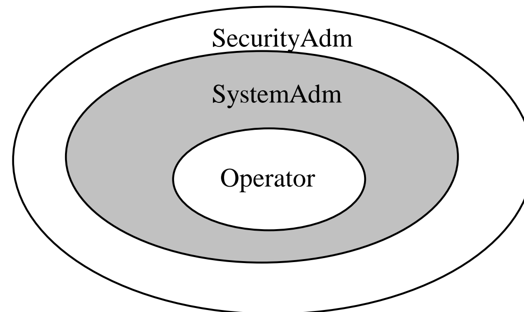

Figure 7. The different user roles. A SecurityAdm is allowed to do the same things as a SystemAdm and an Operator. A SystemAdm the same things as an Operator, but an Operator can only have the role as an Operator.

2.4.8 Start-up and synchronisation

Upon start-up the server, actually the Network Adapter will go through the network element list, establish a contact with every physically available network element, read the element alarm list and synchronise with the alarm list in the database. Such synchronisation will also be performed when the server has gained contact with a previously lost network element.
At a configurable time interval the server will also send a heartbeat request to every network element.

3 Distributed objects

This chapter gives an overview of distributed objects and distributed systems.

3.1 Definition of a distributed object

A distributed object is essentially a component. According to [6], this means it is a blob of self-contained intelligence that can interoperate across operating systems, networks, run on different platforms, languages, applications, tools, and multivendor hardware. One way to explain it is that distributed objects, when packages as *components*, provide the only middleware that can make client/server computing really work at a global level.

Distributed component infrastructure
At the most basic level, a component infrastructure provides of an object software bus - the *Object Request Broker (ORB)* - that lets components interoperate across address spaces, languages, operating systems, and networks. The bus also provides mechanism that lets components exchange metadata and discover each other. At the next level, the infrastructure makes the bus more effective with add-on a system-level service that helps the programmer to create very smart components. Examples of these services include licensing, security, version control, persistence, suite negotiation, semantic messaging, scripting, and many others.

The benefits of distributed objects
A "classical" object (non-distributed), of for example C++ variety is a blob of intelligence that encapsulates code and data. Classical objects provide good code reuse facilities via inheritance and encapsulation. 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 doesn't know about these objects and has no way to access them.

In contrast, distributed object technology is extremely well suited for creating flexible client/server systems, because the data and business logic are encapsulated within objects, allowing them to be located anywhere within a distributed system. The granularity of distribution is greatly improved [6]. Objects should make it possible to manage complex systems by broadcasting instructions and alarms. It should also be possible to modify or change any object without affecting the rest of the components in the system or how they interact.

Disadvantages with distributed object
The opportunities distributed objects provide, such as web-based computing, brings new concerns for interoperability, security, scalability, data integrity, and access to multiple data sources. Some points to consider [6]:

- *Distributed objects can play both client and server roles*. In traditional client/server architecture, it is clear who is a client and who is a server. A server can always be trusted, but not a client. A client can for example trust its database server, but the reverse does not have to be true. In distributed object systems it is not possible to clearly distinguish between clients and servers, as a single object can alternate between these roles.

- *Distributed object interactions are less predictable* because distributed objects are more flexible and granular than other client/server system and may interact in more ad-hoc ways. This is a strength of the distributed object model, but it is also a security risk.
- *Distributed objects can scale without limit*. As every object can be a server it can be millions of servers on the ORB. How to manage access rights for millions of servers?
- *Distributed objects are very dynamic*. A distributed object is inherently anarchistic. Objects come and go. They get created dynamically and self-destructive when they are no longer being used. This dynamism is a great strength of objects, but it could also be a security nightmare.

Good news is that moving the security implementation into the Corba ORB itself can solve many of these problems.

In short
In summary, the object bus and the component infrastructure make it unnecessary to build information systems from scratch. Distributed objects are really independent software components. A component is an object that is not build to a particular program, computer language, or implementation. Objects built as components are well suited for distributed systems. They reduce application complexity, development cost, and time to market. They also improve software reusability, maintainability, platform independence, and client/server distribution. Finally, components provide more freedom of choice and flexibility. All these features, however, introduces "new" aspects that have to be concerned, such as other security threads than for traditional client/server systems.

3.2 Overview of different approaches to distributed computing

There are several different approaches to distributed computing [18], such as:
- *RPC* uses a protocol for implementing the client/server mode. With RPC a specific function is called. The function might be on another host in a network.
- *Java RMI* provides remote communication between programs written in Java. Java RMI can be used on many operating system platforms, as long as there is a Java Virtual Machine (JVM) implementation for the platform.
- *DCOM* is defined by Microsoft. DCOM server components can be written in a number of languages. DCOM will run on any platform, as long as there is a COM Service implementation for the platform.
- *Corba* is an open platform standard, defined by the Object Management Group (OMG). Unlike competing technologies, such as Microsoft's DCOM or Java RMI, Corba is not tied to a specific vendor, platform or programming language. Corba will run on any platform as long as there is a Corba ORB implementation for the platform.

Location transparency is one of the important design issues to be considered for distributed object technology and for Java RMI, DCOM and Corba, location transparency is of great concern.
Location transparency is the ability to access and invoke operations on an object without needing to know where the object resides. The idea is that it should be equally

easy to invoke an operation on an object residing on a remote machine as it is to invoke a method on an object in the same address space.

Corba and Java RMI will be described in more detail as they are of great concern for the Master thesis. For more information about DCOM, see references [6], [18], and [20].

3.3 RMI: Remote Method Invocation

The present GEM system is using Java RMI for the remote communication.
The Java RMI system [10] allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM.

RMI application
RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application.

Distributed object applications need to

- locate remote objects: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the `rmiregistry`, or the application can pass and return remote object references as part of its normal operation.
- communicate with remote objects: Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.
- load class byte-codes for objects that are passed around. Because RMI allows a caller to pass objects to remote objects (copy by value), RMI provides the necessary mechanisms for loading an object's code, as well as for transmitting its data.

In Figure 8[7] an RMI distributed application uses the registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. When the client runs as an applet, the RMI system uses an existing Web server to load class byte-codes, from server to client and from client to server.

[7] [10]

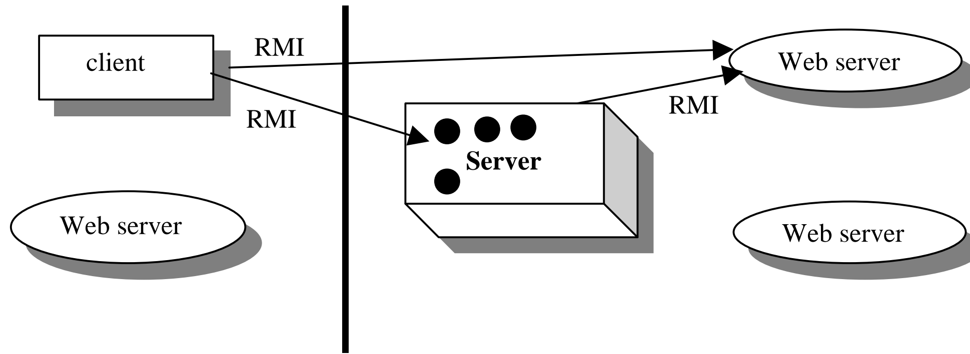

Figure 8. Java RMI distributed application.

Remote Interfaces, Objects, and Methods
Like any other application, a distributed application built using Java RMI consists of interfaces and classes. The interfaces define methods, and the classes implement the methods defined in the interfaces and, perhaps, define additional methods as well. In a distributed application some of the implementations are assumed to reside in different virtual machines. Objects that have methods that can be called across virtual machines are remote objects.

An object becomes remote by implementing a remote interface, which has the following characteristics.

- A remote interface extends the interface java.rmi.Remote.
- Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one virtual machine to another. Rather than making a copy of the implementation object in the receiving virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the caller, the remote reference. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object.
A stub for a remote object implements the same set of remote interfaces that the remote object implements. This allows the stub to be cast to any of the interfaces that the remote object implements. However, this also means that only those methods defined in a remote interface are available to be called.

Creating Distributed Applications Using RMI
When using RMI to develop a distributed application, follow these general steps.

1. Design and implement the components of a distributed application.
2. Compile sources and generate stubs.
3. Make classes network accessible.
4. Start the application.

3.4 Corba Technology

3.4.1 Overview of Corba, ORB and OMG

The abbreviations Corba, ORB and OMG, which are widely used in distributed computing community, stands for:

- Common Objects Request Broker Architecture - the name of the whole architecture
- Object Request Broker - the active component that supplies and handles calls to and from different programs in the network. Depending on the ORB vendor it might be necessary to install on every machine.
- Object Management Group - the consortium that specifies and administrates the Corba standard. Over 800 companies are members.

Corba

Corba is a technology standard developed by OMG, a non-profit group, to allow objects to communicate with each other and develop the architecture to support remote objects. It consists of standards for an Object Request Broker (ORB) and services and facilities supporting a distributed application, see 3.4.7. Corba works in a heterogeneous environment where different operative systems, hardware and programming languages can be used.

Using Corba makes it possible to create an ordinary object and then make it transactional, lockable, and persistent by making the object multiply-inherit from the appropriate services. This means that the programmer can design an ordinary component to provide its regular function and then insert the right middleware mix at run time.

Corba separates interface from implementation and provides language-neutral data types that make it possible to call objects across language and operating system boundaries, see Figure 9[8].

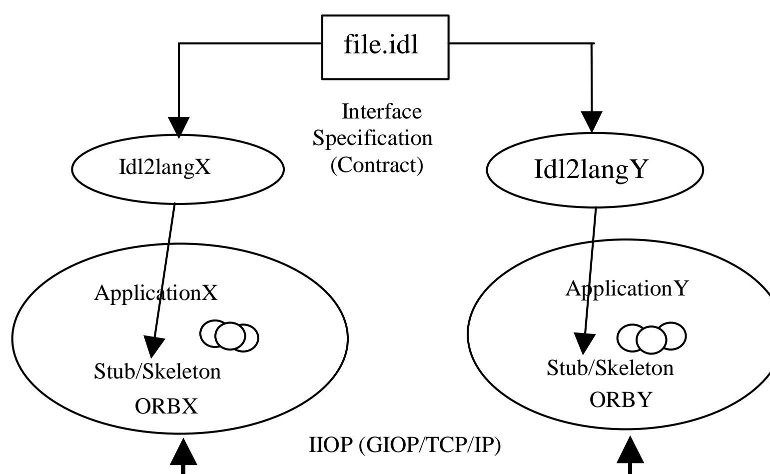

Figure 9. An overview of Corba. An IDL interface acts as a contract between developers of objects and the users of their interfaces.

[8] [21]

Local/remote transparency
In general, a Corba programmer does not have to be concerned with transports, server locations, object activation, byte ordering across dissimilar platforms, or target operating systems - Corba makes it all transparent.

History and versions
- OMG is created 1989 and Corba has been OMG's biggest task.
- Corba 1.0 came 1991, but was during a long time a good thought without implementation.
- With Corba 2.0, which came late 1994, implementations and practical used started to increase.
- From Corba 2.0 IIOP exist, which makes it possible for different ORB to co-operate.
- From Corba 2.3.1 POA exist, which makes the most part of the code ORB vendor independent.
- Corba is under continually development, new specifications are released the whole time.

The Corba Development Process
To build and run an application using Corba the following steps are required, [13]:
1. Write some IDL that describes the interfaces to the object or objects that will be used or implemented.
2. Compile the IDL file. This produces the stub and the skeleton code that implements location transparency, see Figure 12.
3. Identify the IDL compiler-generated interfaces and classes that will be used or specialised in order to invoke or implement operations.
4. Write code to initialise the ORB and inform it of any Corba objects that are created.
5. Compile all the generated code and the application code with a Java compiler, if using Java.
6. Run the distributed application.

As can be seen, it is very similar to how distributed applications are created using Java RMI, see 3.3.

OMG
Since 1989, a consortium of object vendors, the Object Management Group (OMG), has been busy specifying the architecture for an open software bus (the ORB) on which object components written by different vendors can interoperate across networks and operating systems. OMG has today over 800 member companies representing the entire spectrum of the computer industry: 3Com, Canon, Hewlett-Packard, Sun Microsystems. The success of OMG is perhaps that it creates interface specifications, not code. The interfaces it specifies are always derived from demonstrated technology submitted by member companies.

OMG’s primary goal is to create a truly open object infrastructure instead of being controlled by a single company. OMG is definitely doing a great job as it is of great importance to have a standard for objects to interoperate in heterogeneous client/server environment. Unfortunately, when writing Corba programs it is, in some

cases, still necessary to be vendor dependant (in the ORB environment, although, they can co-operate with other Corba implementations). Inprise VisiBroker's SmartAgent is one example.

ORB
The ORB defines a mechanism for objects to communicate with each other. The ORB can be seen as an object bus, which one can plug objects into to communicate, regardless of location and language the object was written. In addition the mechanism for communication is transparent to the programmer. The programmer uses the remote object the same way one would use a regular object. The client object views the remote object as local and the remote object views the client object as local.

Built in security and transaction
The ORB includes context information in its messages to handle security and transactions across machine and ORB boundaries.

The Corba specification provides certain interfaces to components of the ORB, but leaves the interfaces to other components up to the ORB implementer.

3.4.2 Interfaces, IDL and stubs and skeletons

An *interface* is a description of the operations and data types that are offered by an object and can also contain structured type definitions used as parameters to those operations. Interfaces are specified in OMG IDL and are related in an inheritance hierarchy. In Corba, interface types and object types have a one-to-one mapping.

Interface Definition Language (IDL) is a programming *language transparency* that provides the freedom to implement the functionality encapsulated in an object using the most appropriate language. Corba supports this freedom by using OMG IDL for defining the interfaces of Corba objects.

OMG uses IDL contracts to specify a component's boundaries and its contractual interfaces with potential clients. The IDL is purely declarative and therefore has no implementation details. IDL can be used to specify a component's attributes, the parent classes it inherits from, the exception is raises, the typed events it emits, and the methods it interface supports - including the input and output parameters and their data types. IDL-specified methods can be written in and invoked from any language that provides Corba bindings - currently Java, C, C++, Smalltalk, Cobol, Ada, and Object Pascal. It allows client and server objects written in different languages to interoperate across networks and operating systems, see Figure 10[9].

[9] [6], figure 3-1.

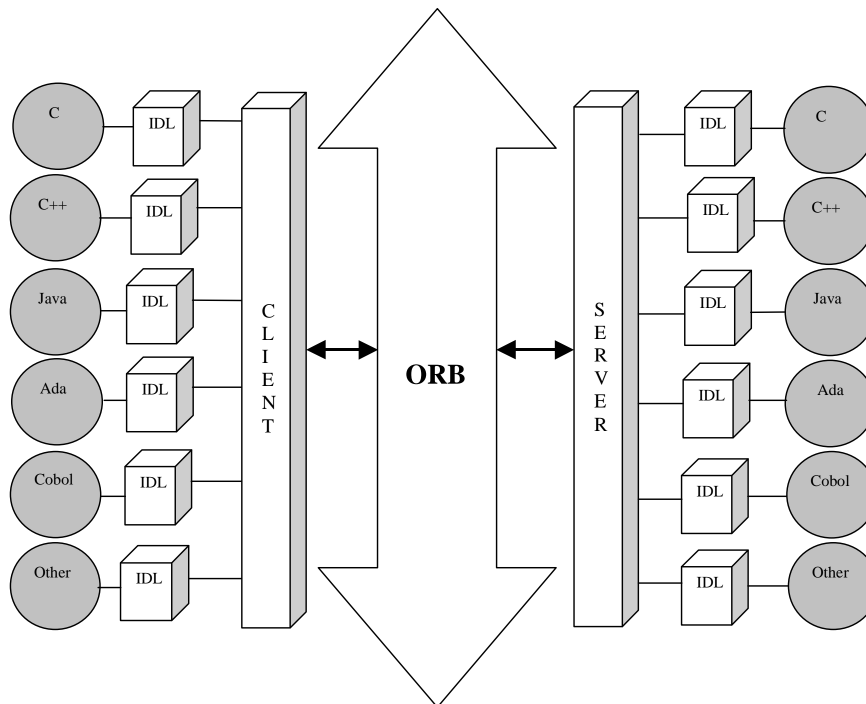

Figure 10. Corba IDL bindings provide client/server interoperability.

The IDL compiler generates *stub code* (e.g. Java classes) for the client and *skeleton code* for the server. The role of the stub code is to provide proxy objects that the client can invoke methods on. The proxy object method implementations invoke operations on the object implementation, which may be located remotely. If the object implementation is at a remote location the proxy objects *marshals* and transmits the invocation request. That is, it takes the operation name and the types and values of its arguments from language-dependent data structures and places them into a *linear representation* suitable for transmitting across a network. The code to marshal programmer-defined data types is an essential part of the stub code. The resulting marshalled form of the request is sent to the object implementation using the particular ORB's infrastructure. This infrastructure involves a network transport mechanism and additional mechanisms to locate the implementation object, and perhaps to activate the Corba server program that provides the implementation.

The *skeleton code* implements the mechanisms by which invocation requests coming into a server can be *un-marshaled* and directed to the right method of the right implementation object. The implementation of those methods is the responsibility of the application programmer. See Figure 11[10] for an illustration of the use of stub, skeleton, and ORB to make a remote invocation.

[10] [13], page 27.

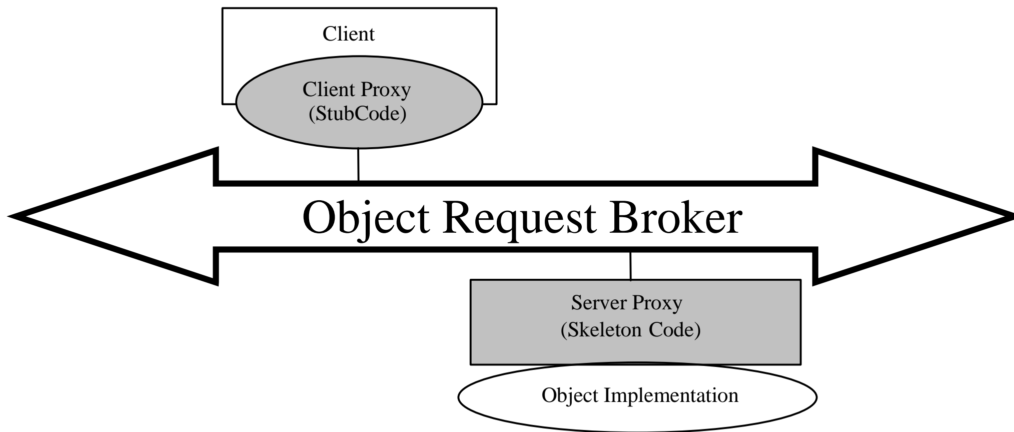

Figure 11. Stub, skeleton and ORB.

The skeleton code provides the glue between an object implementation, a Corba server, and the ORB, in particular the *object adapter,* see *3.4.4*.

A client programmer needs only the IDL to write client code that is ready to invoke operations on a remote object. The client uses the data types defined in IDL through a *language mapping*. This mapping defines the programming language constructs (data types, classes, etc.) that will be generated by the IDL compiler supplied by an ORB vendor, see Figure 12.

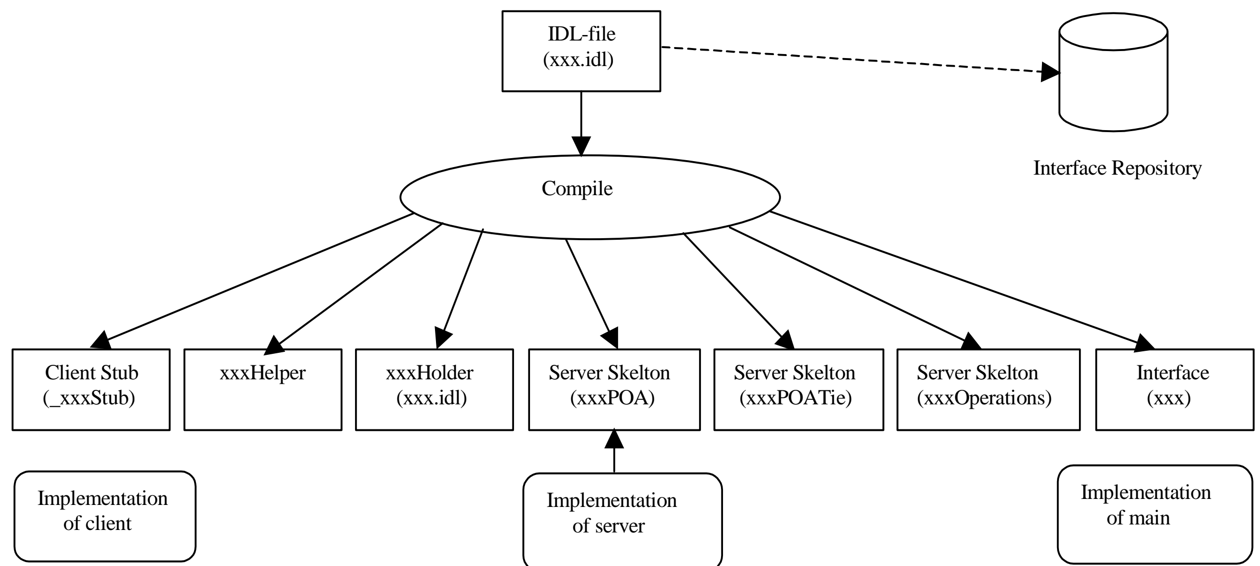

Figure 12. Generated files after compiling an IDL-file (using VisiBroker 4.1).

The IDL grammar is a subset of C++ with additional keyword to support distributed concepts; it also uses the same *pre-processors* as C and C++, see Figure 13 and Figure 14.

```
#ifndef COMMON_IDL
module Common{
        struct PersonalData{
                string firstName;
                string familyName;
                string phoneNumber;
                string address;
        };
};
#endif
```

Figure 13. Example of an IDL-file, Common.idl.

```
#ifndef BANKACCOUNT_IDL
#include <Common.idl>                                      <-- Include file Common.idl
module BankAccount{

        exception toLargeAmount{long maxAmount};           <-- An exception

        interface Account{
                // Attribute
                attribute string account_no;
                attribute long holding_amount;
                // Operations
                boolean insert(in long amount);
                boolean withdraw(in long amount)
                        raises toLargeAmount;
                Owner get_owner();  // Returns another interface
        };
        interface AccountManager{
                Account open(in common::PersonalData id);  <-- Using the included datatype PersonalData
        };
        interface SuperAccount : Account{                  <-- Inherits the interface Account
                // More attributes and operations
        };
};
#endif
```

Figure 14. BankAccount.idl.

The parameters can either be of type in, out or inout. A parameter of type in is for incoming data, out is for outgoing data and inout is for bi-directional data. In Figure 14 all parameters are of type in.

3.4.3 Interface Repository (IR)

The Interface Repository (IR) is a fundamental service in Corba that provides information about the interface types of objects supported in a particular ORB installation. The IR is like a database that contains information about objects. The information includes modules, interfaces, operations and attributes. The repository can be used for a variety of things including a centralised location for information about interfaces available to programmers and to create clients that find and use interfaces at run-time and without prior knowledge of the interface. Essentially the repository contains the information found in the IDL.

3.4.4 **Object Adapter**

An object adapter is the mechanism that connects a request using an object reference with the proper code to service the request. The original OMG specification for binding the object implementation to the ORB was named Basic Object Adapter (BOA). But the BOA was neither robust enough nor precisely enough described. The server side code did for example not port from one vendor's ORB to another. The BOA specification was followed, and replaced, by a Portable Object Adapter (POA) specification that among other things solved this problem. The BOA is anyhow still supported.
POA is used in the Master thesis and will be described in detail in chapter 3.4.9. Here is a short definition [11]: "The POA is a particular type of object adapter specified by OMG to achieve the maximum amount of portability among ORBs that have widely differing design points".

3.4.5 Object Implementation and Object Reference

It is necessary to distinguish between *object implementation* and *object reference*. Object implementation is the code that implements the operations defined by an IDL interface definition, while an object reference is the object's identity, which is used by clients to invoke its operations.

An *object implementation* is the part of a Corba object that is provided by an application developer. It usually includes some internal state, and will often cause side effects on things that are not objects, such as a database, screen display, or telecommunications network elements. The methods of the implementation may be accessed by any mechanism, but in practice most object implementations will be invoked via the skeleton code generated by an IDL compiler.

Object references are handles to objects. A given object reference will always denote a single object, but several distinct object references may denote the same object. Object references can be passed to clients of objects, either as an operation's parameter or result, where the IDL for an operation nominates an interface type, or they can be passed as strings which can be turned into live object references that can have operations invoked on them.
Object references are opaque to their users. That is, they contain enough information for the ORB to send a request to the correct object implementation, but this information is inaccessible to their users. Object references contain information about the location and type of the object denoted, but do so in a sophisticated manner so that if the object has migrated or is not active at the time, the ORB can perform the necessary tasks to redirect the request to a new location or activate an object to receive the request.
Unless an object has been explicitly destroyed, or the underlying network and operating system infrastructure is malfunctioning, the ORB should be able to convey an operation invocation to its target and return results. The ORB also supports operations that interpret the object reference and provide the client with some of the information it contains.

3.4.6 Static and Dynamic Invocation

The ORB lets clients invoke methods on remote objects either statically or dynamically [11]. If a component interface is already defined, the developer can bind

the program to a static stub to call its methods; otherwise, the developer can discover how the interface works at run time by consulting an OMG-specified *Interface Repository,* see chapter 3.4.3.

Dynamic Invocation and Dynamic Skeleton Interfaces
The Dynamic Invocation Interfaces (DII) on the client side and the Dynamic Skeleton Interface (DSI) on the server side is a symmetrical pair of ORB components.
The DII enables a client to invoke operations on an interface for which it has no compiled stub code. That is, it can send the request, do some further processing, and then check for a response. This is useful regardless of whether or not the interface type is known at compile time, as it is not available via a static, or stub-based, invocation.
The DSI is used to accept a request for any operation, regardless of whether it has been defined in IDL or not. The mechanism allows servers to implement a class of generic operations of which it knows the form but not the exact syntax. It helps in writing client code that uses compiled IDL stubs based on an abstract IDL template. The client can then invoke operations on a complied proxy stub in a type-safe manner.

3.4.7 Standard object interfaces

The OMG's object management architecture (OMA) identifies four categories of software interfaces for standardisation [23]: the Corba ORB, the Corba services, the Corba facilities and the Corba domain, see Figure 15[11].

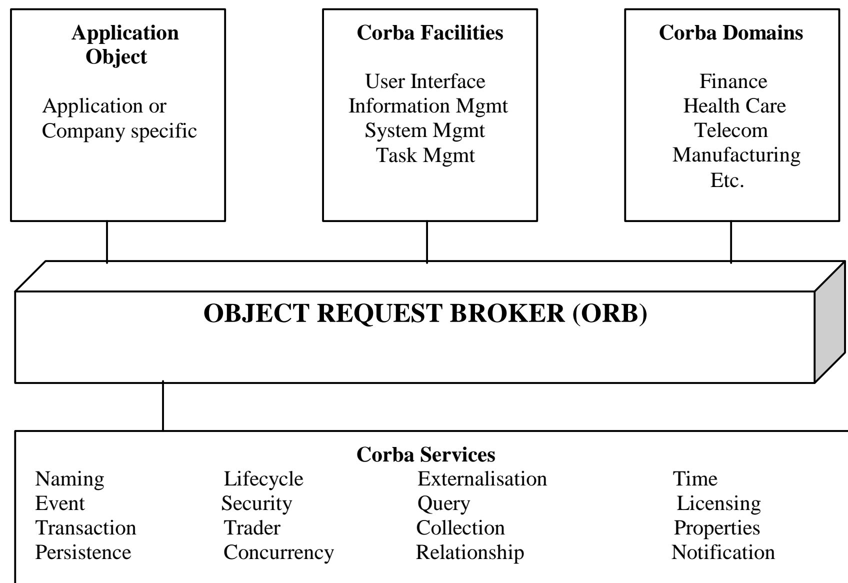

Figure 15. Corba Architecture.

Corba Facilities
Corba Facilities are collections of IDL-defined components that provide services of direct use to application objects.

[11] [21], slide 1-15.

Corba Domains
Corba Domains provides IDL-defined interfaces (or components) for domain specific areas, such as Finance or Telecommunication.

Application objects
Application objects are not standardised by OMG. They are components specific to end-user applications. These objects must be defined using IDL if they are to participate in ORB-mediated exchanges.

ORB and its implementations
The *Object Request Broker* (ORB) is the object bus. It lets objects transparently make request to and receive responses from, other objects located locally or remotely. The client is not aware of the mechanisms used to communicate with, activate, or store the server objects.

A Corba ORB provides a very rich set of distributed middleware services. The ORB lets objects discover each other at run time and invoke each other's services. An ORB is much more sophisticated than alternative forms of client/server middleware, including traditional RPC, Message-Oriented Middleware (MOM), database-stored procedures, and peer-to-peer service [6].

Corba Services
The Corba Services represents services that distributed object systems may need. The remote and local objects to work with the ORB and other objects use these services. According to [11] there are 16 Corba services specified by OMG.

- The **Naming Service** allows objects implementations to be identified by name and is thus a fundamental service for distributed object systems.

- The **Trader Service** is like a directory of available objects.

- The **Event Service** provides channels where supplier objects can input events, and consumer objects can receive them. This is a standard way for components on the bus to register and unregister objects interested in an event. The event object is responsible for notifying event recipients.

- The **Notification Service** works as the Event Service plus using sophisticated event typing and filtering.

- The **Object Transaction Service** provides two-phase commit co-ordination among recoverable components using either flat or nested transaction. Concurrency Control Service supports Object Transaction Service.

- The **Security Service** defines architecture and interfaces for security on an object.

- The **LifeCycle Service** defines operations for creating, copying, moving, and deleting components on the bus.

- The **Relationship Service** allows for creating links between objects that have no direct knowledge of each other.

- The **Persistence Service** provides a single interface for storing components persistently on a variety of storage servers, such as Object Databases (ODBMs).

- The **Externalisation Service** provides a standard way for getting data into and out of component using a stream-like mechanism.

- The **Object Query Service** is a standard for querying objects using a scripting language. It is similar to SQL and based on SQL 2 specification and Object Database Management Group Object Query Language.

- The **Object Properties Service** provides operations to associate named values or properties with any component.

- The **Concurrency Control Service** provides a lock manager that can obtain locks on behalf of either transactions or threads.

- The **Licensing Service** provides operations for measuring the usage of an object.

- The **Secure Time Service** is a standard for providing synchronising time in a distributed environment.

- The **Object Collection Service** provides a standard for creating and manipulating collections of objects.

Of all the services, the Naming Service is the most important and therefore always in use. The Naming Service makes the local transparency possible. As the service is relevant for the Master thesis it will be described in more detail.

Naming Service
The use of object references alone to identify objects has two problems for human users, first, because object references are opaque data types, and second, their string forms is a long sequence of numbers. When a service is restarted, its objects typically have a new object reference. However, in most cases clients want to use the service repeatedly without needing to be aware that the service has been restarted, *object persistence*. The Naming Service solves these problems by providing an extra layer of abstraction for identification of objects.

The Naming Service provides a mapping between a name and an object reference. According to the OMG specification [16] is a name-to-object association called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Contexts are similar to directories in file systems and they can contain name bindings as well as sub-context, see Figure 16[12]. Different names can be bound to an object in the same or different context at the same time. Obtaining an object reference, which is bound to a name in a given context, is known as *resolving the name*. To *bind a name* is to create a name binding in a given context. A naming context can also be bound to a name in a naming context. Binding context in other contexts creates a *naming graph*.

[12] [12], figure 18.2.

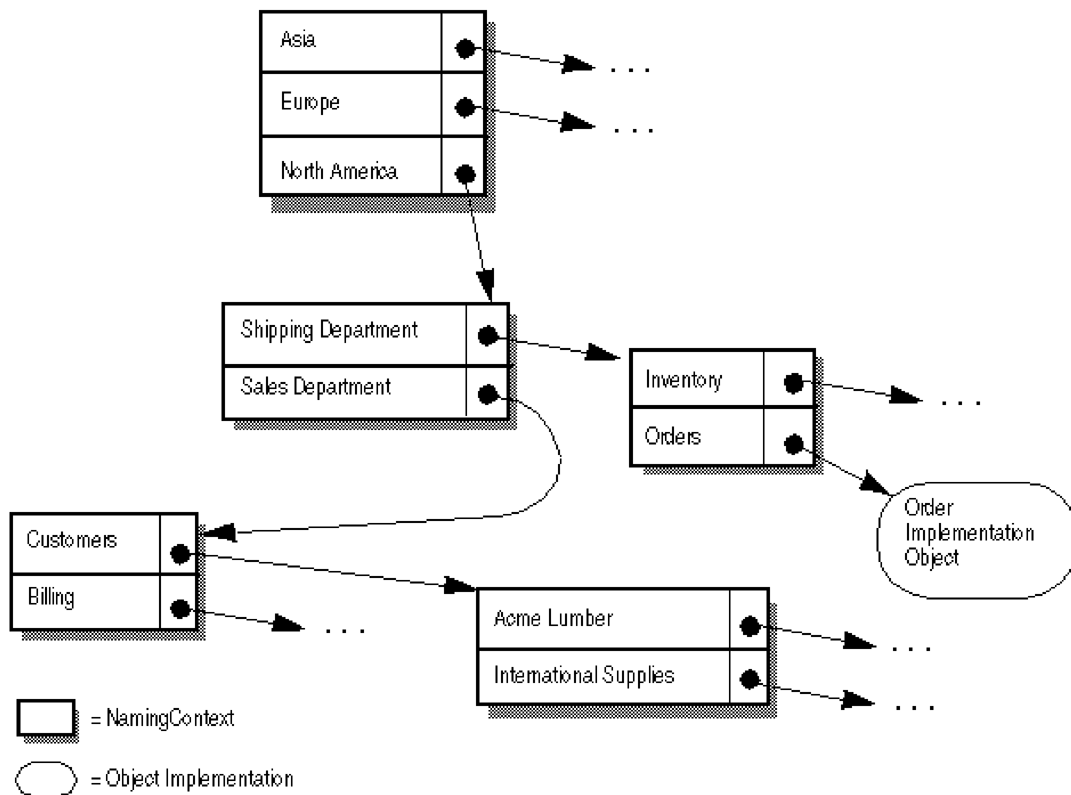

Figure 16. Naming scheme for an order entry system.

The typical use of the Naming Service involves object implementations binding to the Naming Service when they come into existence and unbinding before they terminate. Clients resolve names to objects, on which they subsequently invoke operations. See this usage scenario in Figure 17[13].

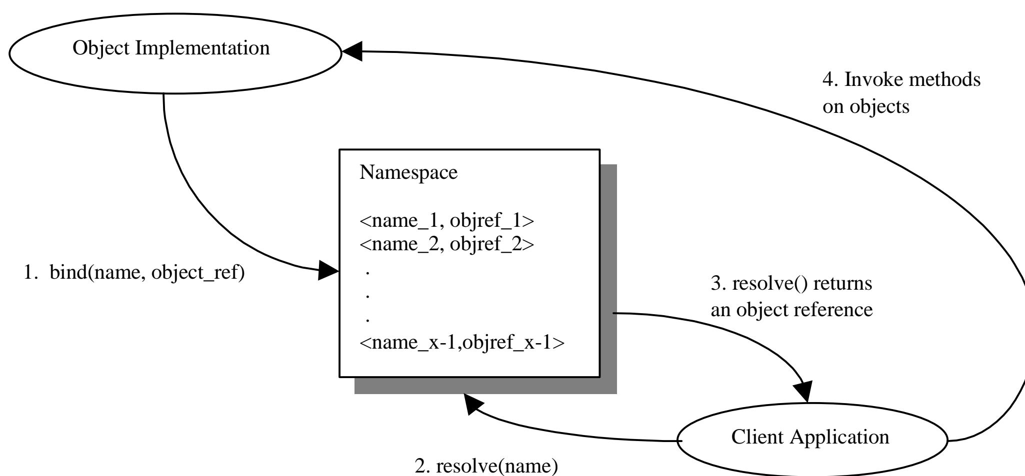

Figure 17. Binding, resolving, and using an object name from a naming context within a namespace.

[13] [12], figure 18.1.

Event Service
The Corba Event Service provides a way of distributing data about an occurrence in a distributed application to a number of interested parties without requiring the originator of the event data to know the receivers and to make several calls to specific objects.
The Event Service's *event channel* takes event data from a *supplier* of events and delivers that data to one or more event *consumers,* see Figure 18[14]. The channel may act as a client to the supplier, *pulling* the event data from the supplier, or it may provide an object interface that allows the supplier to *push* the event data into the channel. When the data is to be delivered to consumers, the same options are available: the channel may push the event data to the consumer, or it may wait until the consumer pulls the event data from the channel. Channels may also use a combination of the push and pull approaches with different clients.
The specification defines the communication interfaces used to push and pull event data to and from suppliers and consumers. It then defines the Event Channel in terms of proxy suppliers and consumers. That is, the channel is an intermediary object between a supplier and a consumer, and it act toward a supplier as a *proxy consumer* and toward a consumer as a *proxy supplier*.

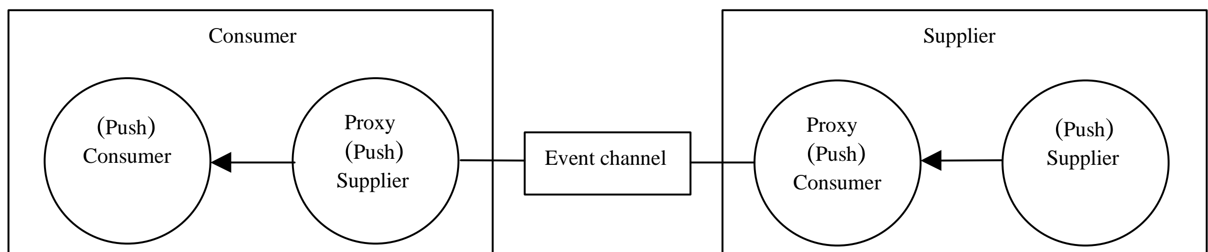

Figure 18. Event channel between Consumer and Supplier. The channel acts as an intermediate object.

3.4.8 Interoperability

Local and remote invocation
Corba 2.0 interoperability is based on ORB-to-ORB-communication [11], see Figure 19[15]. All of the clients connected to ORB 1 can access object implementations in both ORB 1 and ORB 2. The same conditions holds for clients connected to ORB 2. The architecture scales to any number of connected ORBs.
An invocation from a client of ORB 1 passes through its IDL stub into the ORB core. ORB 1 extracts the location of the object instance from the object reference where it is encapsulated. If the target instance is *local*, ORB 1 passes the invocation through the skeleton code to the object for servicing. If the target is remote, ORB 1 marshals arguments for the wire and passes the invocation across the communication pathway to ORB 2, which un-marshals and routes everything to the object. Because the invocation must come into the implementation via either the skeleton or the DSI (Dynamic Skeleton Interface), the object implementation (like the client) has no way of knowing whether the client is local or remote, nor does it care.

[14] [21], slide 9-7.
[15] [11], page 163.

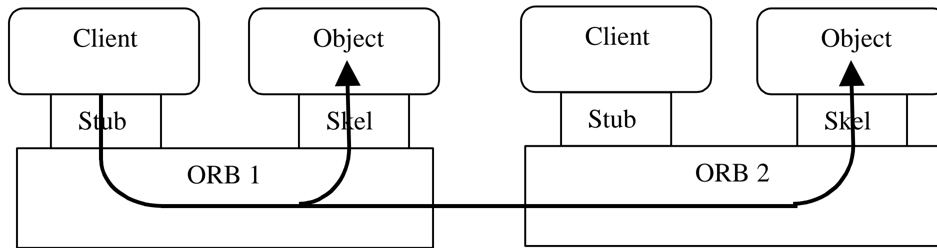

Figure 19. Interoperability via ORB-to-ORB communication.

For invocations that stay local, Corba only adds a few extra instructions and the servicing is extremely fast. For remote invocation, most of the processing is local and only the actual invocation goes out over the wire unless the target object has moved; in this case, an extra round-trip or two are necessary to find its new location.

Note that client and object implementations are not involved in the communication step. In Corba, the communication always goes from one ORB to another.

Remote invocation works regardless of platform, protocol, and format differences that might exist between ORB 1 and ORB 2. That made it necessary for both ORBs to have enough information about the invocation and response to allow them to translate data where necessary as they transfer the requests from Platform 1 to Platform 2, and back. IDL interfaces are the key to this: Interface definitions are encoded in the stubs and skeletons, where they control marshalling and un-marshalling; for dynamic invocations, these details come from the *Interface Repository*, see 3.4.3.

Interoperable Object References (IORs)
OMG has specified a standard object reference format, as part of the architecture. It is named Interoperability Object Reference (IOR) and contains the same information as a single domain object reference, but adds a list of protocol profiles indicating which communication protocols the domain of origin can accept requests in.
IOR is used in inter-ORB invocations, so it is emitted and accepted by ORBs speaking to the network, and used by the bridges between them.

Protocols
Some protocols are official OMG standards. So far there are two protocols: the IIOP, and the DCE-ESIOP [11]. See [21][16] for the relationships between these protocols.

[16] [13], page 68.

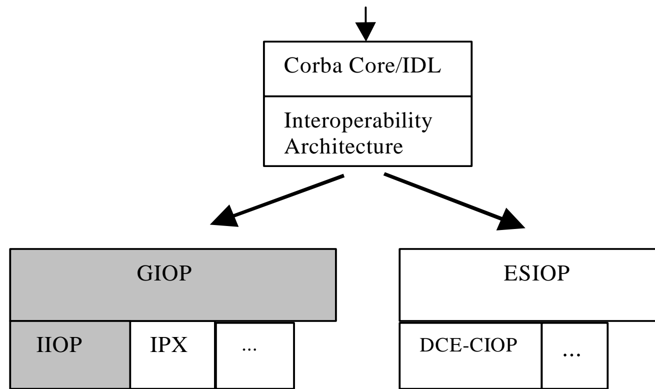

Figure 20. *Structure of the Corba Interoperability Specification.* Shaded components are mandatory for Corba Interoperability compliance. *Interoperability Architecture* contains the basic architecture built on bridging; the Interoperable Object Reference (IOR); the interoperability interfaces including the DSI; and the provision for context-specific services.

General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP)
The *GIOP* defines a linear format for the transmission of Corba requests and replies without requiring a particular network transport protocol.

IIOP, which is the GIOP over TCP/IP, is the one protocol *mandatory* for Corba Interoperability compliance, see Figure 21[17]. It defines some primitives to assist in the establishment of TCP connections. This protocol is required for compliance to Corba 2.0 and is intended to provide a base-level interoperability between all ORB vendors' products, even though some vendors will continue to support proprietary protocols.

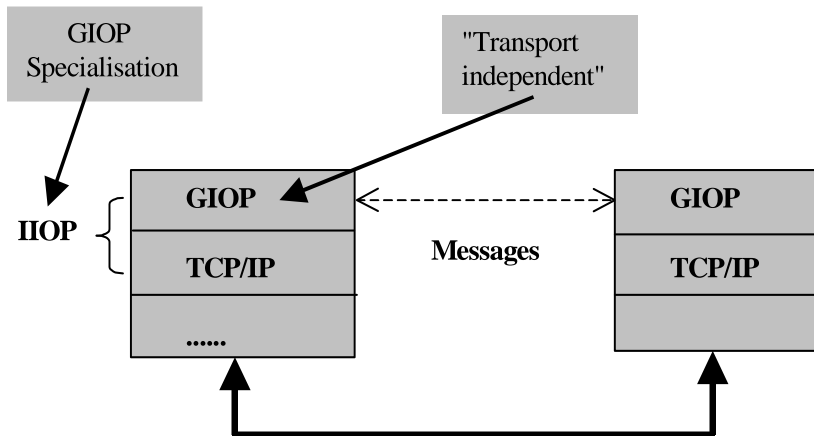

Figure 21. IIOP is GIOP over TCP/IP.

Environment-Specific Inter-ORB Protocols (ESIOPs)
A non-GIOP protocol is an ESIOP if it is based on the interoperability architecture including domains, bridging, the IOR, and the interoperability interfaces including DSI. A Distributed Computing Environment (DCE)-based protocol adopted by OMG,

[17] [21], slide 1-23.

as a part of Corba 2, is an ESIOP. Even though it is not mandatory for all ORBs is it an OMG standard, and using it is the only way to implement a DCE-based protocol in Corba.
The DCE-CIOP is a protocol for ORB-to-ORB communications, there it plays the same role as IIOP.

3.4.9 The Portable Object Adapter (POA)

A POA is the intermediary between the implementation of an object and the ORB.

Definitions
Definitions of some key concepts used in the POA specification:

- *Servant***.** An implementation/programming object that provides the run-time semantics of one or more Corba objects. A servant is not a Corba object.
- *Object ID.* An identifier, unique with respect to a POA, that the POA uses to associate a Corba object identity with a servant.
- *Incarnate.* The action of providing a running servant to serve requests associated with a particular object ID. A POA will keep this association in its active object map.
- *Etherealise.* The action of destroying a servant associated with an object ID, so that the object ID no longer identifies a Corba object with respect to a particular POA.
- *Default Servant.* An object to which all incoming requests for object IDs not in the Active Object Map are dispatched.

In its role as an intermediary, a POA route requests to *servants* and, as a result may cause servants to run and create child POAs if necessary.
Servers can support multiple POAs. One POA is created automatically, the rootPOA. The set of POAs is hierarchical; all POAs have the rootPOA as their ancestor. *Servant managers* locate and assign servants to objects for the POA. When an abstract object is assigned to a servant, it is called an active object and the servant is said to *incarnate* the active object. Every POA has one Active Object Map, which keeps track of the object IDs of active objects and their associated active servants. See Figure 22[18] for an overview of the POA.

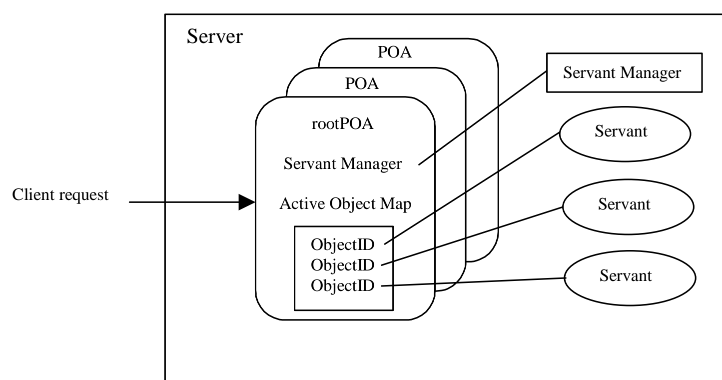

Figure 22. Overview of the POA.

[18][12], Figure 7.1.

A POA object is *locality constraint*. This means that references to objects defined in POA may not be passed outside of a server's address space. Its job is to deal with requests on a particular computer.

Servants and servant managers
Servant manager performs two types of operations: find and return a servant, and deactivate a servant. They allow the POA to activate objects when a request for an inactive object is received. Servant managers are *optional*. For example, servant managers are not needed when the server loads all objects at start-up. Servant managers may also inform clients to forward requests to another object using ForwardRequest.

A *servant* is an active instance of an implementation. The POA maintains a map of the active servants and the object IDs of the servants. When a client request is received, the POA first checks this map to see if the object ID (embedded in the client request) has been recorded. If it exists, then the POA forwards the request to the servant. If the object ID is not found in the map, the servant manager is asked to locate and activate the appropriate servant. See Figure 23[19] of the example scenario. Scenarios can be different depending on the POA polices.

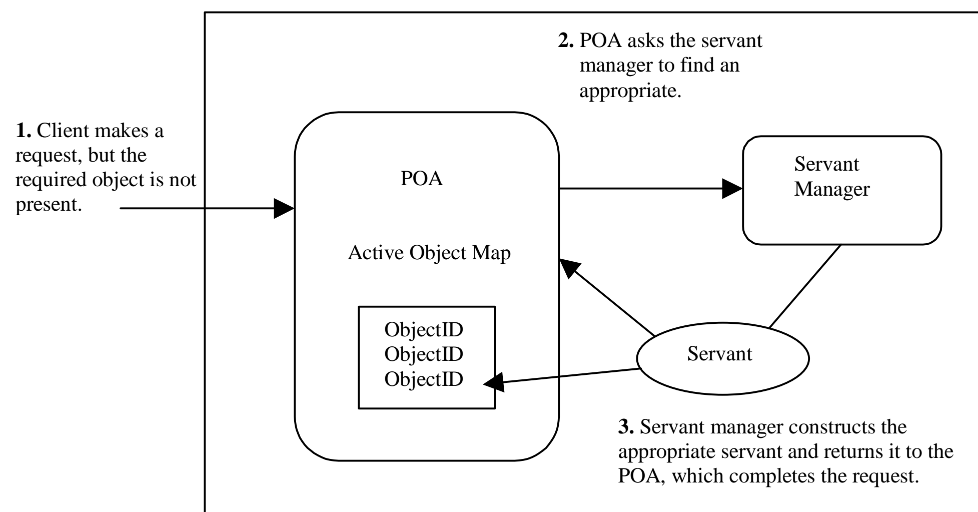

Figure 23. Servant manager function.

Setting up a POA with a servant
The steps to follow are the following:
- Obtaining a reference to the rootPOA
- Defining the POA policies
- Creating a POA as a child of the rootPOA
- Creating a servant and activating it
- Activating the POA through its manager

These steps can be seen in Figure 25.

[19] [12], Figure 7.2.

Example code: The example code is from for the Master thesis[20] and shows the FaultManagement part. The FaultManagement interface is specified in Figure 24. When compiling the IDL file creates, among other, a FaultManagementPOA.java file and this serves as the skeleton code for the FaultManagement object implementation on the server side, see Figure 25.

```
#ifndef INTERFACES_IDL
#include <CommonDeclare.idl>

module interfaces{

  /**
  * The interface is used by any component that wants alarms from the database.
  */
  interface FaultManagement {
   /**
   * Returns all active alarms from database.
   */
   commonDeclare::alarmList getAlarms()
     raises (commonDeclare::DbException);
   /**
   * Called from clients in order to update active alarms.
   */
   void updateAlarm(in commonDeclare::Client client, in commonDeclare::alarmList alarms, in long updateData)
     raises (commonDeclare::DbException);
   commonDeclare::alarmList getSelectedAlarms(in long from, in long to)
     raises (commonDeclare::DbException);
  /**
   * Returns the number of cleared alarms in the database.
   */
   long long getAlarmHistoryCount(in commonDeclare::Date from, in commonDeclare::Date to)
     raises (commonDeclare::DbException);
   /**
   * Returns a vector of cleared alarms.
   */
  commonDeclare::alarmList getAlarmHistory(in long size, in long order)
     raises (commonDeclare::DbException);
   /**
   * Returns a vector of cleared alarms.
   */
   commonDeclare::alarmList getAlarmHistoryDate(in long size, in long order, in commonDeclare::Date from, in
commonDeclare::Date to)
     raises (commonDeclare::DbException);
  };
```

Figure 24. The FaultManagement interface in the IDL file Interfaces.idl. Just a part of the file is shown.

[20] As VisiBroker 4.1 is used it might be a difference when using another ORB.

```
/** The start menu item has been clicked.
   * Construct all the Corba servers and add them in to the NameService.
   */
   void jMenuItem2_actionPerformed(ActionEvent e){
        ...
      try
      {
        // Init ORB
        ORB orb = ORB.init(m_args, null);
        // Init root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        // Init root POA policies, make PERSISTENT
        org.omg.CORBA.Policy[] policies = {
                                   rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
        };
        //Init NameService
        org.omg.CORBA.Object nameObj = orb.resolve_initial_references("NameService");
        NamingContextExt root = NamingContextExtHelper.narrow(nameObj);
        //Instansiate object implementation
             ...
        //Fault Management
        //Create a fmPOA with the same policies as the rootPOA
        POA fmPOA = rootPOA.create_POA("FaultManagementPOA",rootPOA.the_POAManager(),policies);
        //Create a servant
        FaultManagementImpl fm = new FaultManagementImpl((DistributionProxyAlarms)dist ,conf);
        //Bind the servant to an ID
        byte[] faultManagementID = "FaultManagement".getBytes();        The name may always be the same
        //Activate servant with the objectID on fmPOA
        fmPOA.activate_object_with_id(faultManagementID, fm);
        //Activate the POA Manager
        rootPOA.the_POAManager().activate();
        //Bind object to name
        NameComponent faultManager = new NameComponent("FaultManagement","");
        root.rebind(new NameComponent[] {faultManager}, fmPOA.servant_to_reference(fm));
        //Wait for incoming requests
        orb.run();
      }
       catch (org.omg.CORBA.SystemException se){
         System.out.println("ERROR: CORBA SYSTEMException" + se.getMessage());
         se.printStackTrace();
       }
       catch (org.omg.CORBA.ORBPackage.InvalidName exc){
         System.out.println("ERROR: The ORB could not resolve a reference to the NameService");
         exc.printStackTrace();
       }
      }
   }
```

Figure 25. The server implementation of the code. Note: This shows just the instantiation of FaultManagement part of the code.

In the example code the FaultManagement is activated with activate_object_with_id (see Figure 25), which passes the objectID to the Active Object Map where it is recorded. This approach ensures that this object is always available when the POA is active and is called *explicit object activation*. The Corba NameService is in use, see Naming Service in chapter 3.4.7. A logical name is associated with an object reference and the logical name is stored in a namespace. In this example an object reference is named *FaultManagement*. This allows a client application to use the Naming Service to obtain an object reference by using the logical name assigned to that object, see Figure 26.

```
public ServerProxyImpl(boolean alarmSubscriber, boolean netlistSubscriber, String hostName, String[] args)
{
  this.hostName = hostName;
  isAlarmSubscriber = alarmSubscriber;
  isNetlistSubscriber = netlistSubscriber;

  try{
    // Init ORB
    ORB orb = ORB.init(args,null);

    // Init NameService
    Object nameService = orb.resolve_initial_references("NameService");
    nc = NamingContextHelper.narrow(nameService);
          ...
    //Resolve a logical name to an object reference.
    NameComponent faultManagementName = new NameComponent("FaultManagement","");
    org.omg.CORBA.Object faultMObj = nc.resolve(new NameComponent[] {faultManagementName});
    faultManagement = FaultManagementHelper.narrow(faultMObj);
          ...
    alarms = new FaultManagementProxyImpl(faultManagement);

   // Create callback objects and activate them on rootPOA
   POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
   regAlarmClient = new ClientImpl(alarms,true,false);
   regNetlistClient = new ClientImpl(netlist,false,true);
   rootPOA.activate_object(regAlarmClient);
   rootPOA.activate_object(regNetlistClient);
   rootPOA.the_POAManager().activate();
   regAlarmCl = ClientHelper.narrow(rootPOA.servant_to_reference(regAlarmClient));
   regNetlistCl = ClientHelper.narrow(rootPOA.servant_to_reference(regNetlistClient));
   this.registerSubscriber();
   }
   catch (org.omg.CORBA.ORBPackage.InvalidName ex){
     System.out.println("Can't find nameservice!");
   }
   catch (Exception e)
   {
     System.out.println(e);
     e.printStackTrace();
   }
}
```

Figure 26. The client implementation of the code. Note, only the implementation of FaultManagement is shown.

3.4.10 Tie Mechanism

The inheritance approach has some shortcomings. Since Java only supports *single class inheritance*, an object implementation cannot extend any application specific class as it already extends the skeleton class, see Figure 27. Some times it might be necessary for one Java object to implement multiple IDL interfaces, for example, an application specific interface and a general management interface. This cannot be achieved via Java extension, as the implementation object in that case needs to extend two or more skeletons, see Figure 29.

```
public class FaultManagementImpl extends FaultManagementPOA{

      public CorbaAlarms[] getAlarms throws DBException{ <-- The specification is in the interface FaultManagement
            ...
      }
}
```

Figure 27.   Single class inheritance.

A solution to these problems is to use delegation instead of inheritance. This is achieved by generating a *pseudo-implementation* or Tie-class, which inherits the skeleton. However, rather than implementing the operations, this pseudo-implementation class delegates all calls to the actual implementation, see Figure 28. The Tie-class acts like a proxy that takes care of the net communication.

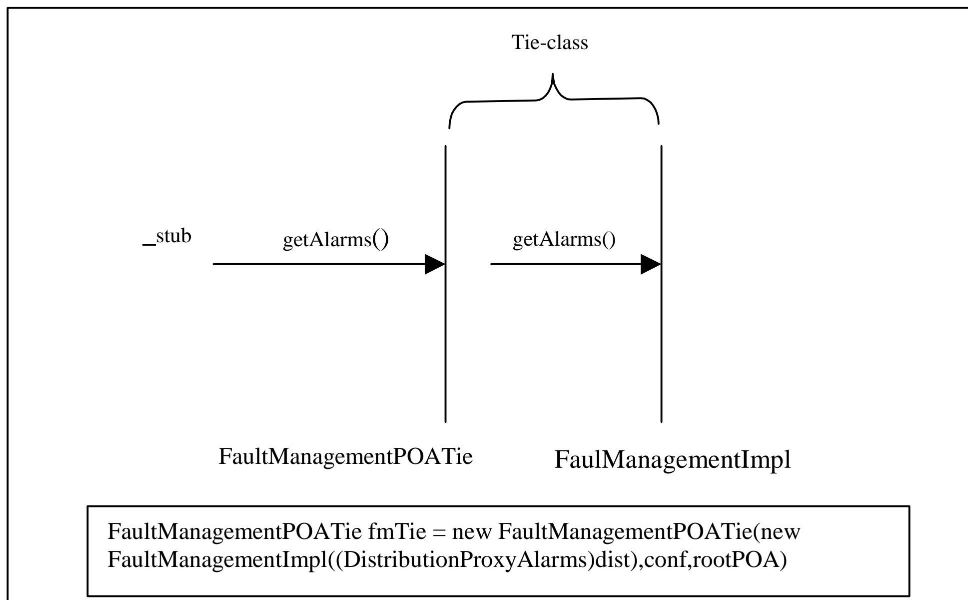

Figure 28.   A schematic picture of the Tie-mechanism.

When compiling an IDL file, see Figure 12, two additional files are generated: <InterfaceName>POATie and <InterfaceName>Operations, and those are used when using the Tie-mechanism, see Figure 29.

```
/**
* This class handles all operations regarding alarms.
*/
public class FaultManagementImpl implements FaultManagementOperations,FaultManagementNAOperations {

        public CorbaAlarm[] getAlarms() throws DbException{          <-- The specification is in the
            ...                                                          interface FaultManagement
        }

        public string newAlarm(CorbaAlarm alarmCorba) throws DbException{   <-- The specification is in the
            ...                                                              interface FaultManagementNA
        }
}
```

Figure 29. The implementation class only needs to implement <InterfaceName>Operations and can still inherit another class.

Changes in the server code example in Figure 25 when using the Tie-mechanism is (changed lines in bold):

```
/** The start menu item has been clicked.
  * Construct all the Corba servers and add them in to the NameService.
  */
  void jMenuItem2_actionPerformed(ActionEvent e){
        ...
     try
     {
       // Init ORB
       ORB orb = ORB.init(m_args, null);
       // Init root POA
       POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
       // Init root POA policies, make PERSISTENT
       org.omg.CORBA.Policy[] policies = {
                                   rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
       };
       //Init NameService
       org.omg.CORBA.Object nameObj = orb.resolve_initial_references("NameService");
       NamingContextExt root = NamingContextExtHelper.narrow(nameObj);
       //Instansiate object implementation
         ...
       //Fault Management
       POA fmPOA = rootPOA.create_POA("FaultManagementPOA",rootPOA.the_POAManager(),policies);
       FaultManagementPOATie fmTie = new FaultManagementPOATie(new
FaultManagementImpl((DistributionProxyAlarms)dist ,conf,rootPOA));
       // FaultManagement: Bind it to an ID              <-- The name may always be the same
       byte[] faultManagementID = "FaultManagement".getBytes();
       fmPOA.activate_object_with_id(faultManagementID, fmTie);
         ...
       //Activate the POA Manager
       rootPOA.the_POAManager().activate();
       //Bind object to name
       NameComponent faultManager = new NameComponent("FaultManagement","");
       root.rebind(new NameComponent[] {faultManager}, fmPOA.servant_to_reference(fmTie));
       //Wait for incoming requests
       orb.run();
    }
     catch (org.omg.CORBA.SystemException se){
       System.out.println("ERROR: CORBA SYSTEMException" + se.getMessage());
       se.printStackTrace();
     }
      catch (org.omg.CORBA.ORBPackage.InvalidName exc){
       System.out.println("ERROR: The ORB could not resolve a reference to the NameService");
       exc.printStackTrace();
      }
         ...
    }
  }
```

Figure 30. Changes in the server code example in Figure 25 when using Tie-mechanism.

3.5 Corba compared with Java RMI

Both Corba and Java RMI defines a programming model for distributed object computing. The biggest difference is that Java RMI is only for use for objects written in Java. Corba objects can be written in a variety of languages including C++, Cobol, Java, Ada, Smalltalk and Object Pascal.

Protocol

Java RMI defines APIs, that enable invocations of methods on Java objects across JVM and machine boundaries. RMI uses its own proprietary protocol, Java Remote Method Invocation Protocol (JRMP) over TCP/IP and does not define a specific protocol, but can be implemented with IIOP (including object-by-value). RMI-over-IIOP natively supports the propagation of transaction and security contexts and allows the integration of legacy systems through IDL interfaces implemented in other languages [24]. Corba is using the two official OMG standard protocols: IIOP and DCE-ESIOP.

Call-by-value

RMI provides call-by-value mechanisms for Java objects. For example, RMI makes it possible to send a Hashtable object from one Java machine to another. From Corba version 2.3 is it also possible to use Object-by-value (e.g. passing of objects by value), but it is not as simple as in RMI. In RMI is it not necessary to add code. In Corba is it necessary to use the IDL data type *valuetype* to make a mapping to a public Java class with the same name, see 5.3.2 for more information.

Naming Service

In RMI, for a client to locate a server object for the first time, RMI depends on a naming mechanism called a RMIRegistry that runs on the Server machine and holds information about available Server Objects, see Figure 31. A Java RMI client acquires an object reference to a Java RMI server object by doing a lookup for a Server Object reference and invoke methods on the Server Object as if the Java RMI server object resides in the client's address space [18], see Figure 32.
For Corba the client and server parts are described in chapter 3.4.9. To compare with RMI the server side part is in Figure 33 and the client side part is in Figure 34. As can be seen, Corba requires more coding.

```
try
{
	LocateRegistry.createRegistry(1099);
	//LogManager
	LogManagerImpl lmi = new LogManagerImpl(Database.getDbLogInstance());
	Naming.rebind("LogManagerServer",lmi);
	//Netlist
	NetListHandlerImpl nl = new NetListHandlerImpl((NetListSubscriber) dp);
	Naming.rebind("NetList",nl);
}
```

Figure 31. Server side, using RMI. LocateRegistry.createRegistry creates and exports a Registry on the local host that accepts requests on the specified port (1099). Naming.rebind rebinds the specified name to a new remote object. Any existing binding for the name is replaced.

```
public SysAdmProxy (String hostName){
	this.hostName = hostName;
	try{
		// NetworkElements
		String rmiHost = "//" + this.hostName + ":1099/NetList";
		NetListHandler NL = (NetListHandler)Naming.lookup(rmiHost);
		NetListHandlerNLW NLAdmin = (NetListHandlerNLW)Naming.lookup(rmiHost);
		// ActionLog
		rmiHost = "//" + this.hostName + ":1099/LogManagerServer";
		Log log = (Log)Naming.loopup(rmiHost);
	}
	...
}
```

Figure 32. Client side, using RMI. Naming.lookup returns a reference, a stub, for the remote object associated with the specified name (rmiHost).

```
try{
	//Init ORB
	ORB orb = ORB.init(m_args,null);
	//Init rootPOA
	POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
	//Init rootPOA policies, make PERSISTENT
	Policy[] policies = {
		RootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
	};
	//Init NameService
	Object nameObj = orb.resolve_initial_references("NameService");
	NamingContextExt root = NamingContextExtHelper.narrow(nameObj);
	//Instatiate objects
	//LogManager
	POA logMPOA = rootPOA.create_POA("LogPOA",rootPOA.the_POAManager(),policies);
	LogPOATie logMTie = new LogPOATie( new LogManagerImpl(Database.getDbLogInstance(), rootPOA));
	byte[] logManagerServerID = "LogManagerServer".getBytes();
	logMPOA.activate_object_with_id(logManagerServerID,logMTie);
	//Netlist
	POA netlPOA = rootPOA.create_POA("NetListHandlerPOA",rootPOA.the_POAManager(),policies);
	NetListHandlerPOATie netlTie = new NetListHandlerPOATie(new NetListHandlerImpl((NetListSubscriber)
dist,rootPOA));
	byte[] netListHanderID = "NetListHandler".getBytes();
	netlPOA.activate_object_with_id(netListHandlerID,netlTie);
	//NetlistNLW
	POA netlNLWPOA =
rootPOA.create_POA("NetListHandlerNLWPOA",rootPOA.the_POAManager(),policies);
	NetListHandlerNLWPOATie netlNLWTie = new NetListHandlerNLWPOATie(new
NetListHandlerImpl((NetListSubscriber) dist, rootPOA));
	byte[] netListHandlerNLWID = "NetListHandlerNLW".getBytes();
	netlNLWPOA.activate_object_with_id(netListHandlerNLWID,netlNLWTie);
	//Activate the POA Manager
	rootPOA.the_POAManager().activate();
	//Bind object to name
	NameComponent logManager = new NameComponent("LogManagerServer","");
	root.rebind(new NameComponent[]{logManager},logMPOA.servant_to_reference(logMTie));
	NameCompontent netListH = new NameComponent("NetListHandler","");
	root.rebind(new NameComponent[]{netListH},netlPOA.servant_to_reference(netlist));
	NameComponent netListNLWH = new NameComponent("NetListHandlerNLW","");
	root.rebind(new NameComponent[]{netListNLWH},netlNLWPOA.servant_to_reference(netlNLWTie));
	//Wait for incoming requests
	orb.run();
```

Figure 33. Server side, using Corba.

```
public SysAdmProxy(String[] args){
	try{
		//Init ORB
		ORB orb = ORB.init(args,null);
		//Init NameService
		Object nameService = orb.resolve_initial_references("NameService");
		NamingContext nc = NamingContextHelper.narrow(nameService);
		//Network elements
		NameComponent netlistName = new NameComponent("NetListHandler","");
		Object netlistObj = nc.resolve(new NameComponent[]{netlistName});
		NetListHandler NL = NetListHandlerHelper.narrow(netlistObj);
		NameComponent netlistNLWName = new NameComponent("NetListHandlerNLW","");
		Object netlistNLWObj = nc.resolve(new NameComponent[]{netlistNLWName});
		NetListHandlerNLW NLAdmin = NetListHandlerNLWHelper.narrow(netlistNLWObj);
		//Action logs
		NameComponent logManagerServerName = new NameComponent("LogManagerServer","");
		Object logMObj = nc.resolve(new NameComponent[]{logManagerServerName});
		Log log = LogHelper.narrow(logMObj);
	}
	...
}
```

Figure 34. Client side, using Corba.

Corba has several benefits over traditional two, three, and N-tier client/server systems.

- The programmer uses a familiar method for using a remote object.
- Can find, query and invoke a method without knowledge of the actual behaviour.
- Integration of legacy systems can be accomplished by hiding them behind an interface.
- Corba allows objects to seamlessly interact with each other without regard to the location, language or platform (vendor independent).
- Corba can scale from a stand-alone palmtop to the enterprise.
- Corba hides the locations of object.
- The specification is separate from the implementation. The interface is defined by a general language, IDL.

(Corba) ORBs and Java RMI have been viewed as competing technologies. SUN and OMG are, however, working very close with Java RMI and Corba. In mid-1999 a new version of Java RMI, supporting IIOP, was official released. The version was developed jointly by IBM and Sun [26]. Additionally the OMG aligned its object-by-value specification so that it is fully compatible with RMI.

Finally some word from [27][21]:

- Start with Corba if you need to build to an architecture.
- Start with RMI if you are programming a simple all-Java prototype (not for deployment).

[21] Chapter 9, slide 23.

4 Security in distributed objects

Security is an important issue for most distributed applications, in particular when developed over the Internet. Distributed object systems are for example more vulnerable to security breaches than the more traditional systems, as there are more places where the system can be attached. Therefore, security is needed in Corba systems, which takes account of their inherent distributed nature.

4.1 Threats in a distributed object system

There are several threats in a distributed object system [15]:

- An authorised user of the system gain access to information that should be hidden from him/her.
- A user masquerading as someone else, (in other words: obtaining access to whatever that user is authorised to do) so that actions are being attributed to wrong person. In a distributed system, a user may delegate his/her rights to other objects, so they can act on his/her behalf. This adds the threat of rights being delegated to widely, again causing a threat of unauthorised access.
- Security being bypassed.
- Eavesdropping on a communication channel, so gaining access to confidential data.
- Tampering with communication between objects - modifying, inserting, and deleting items.
- Lack of accountability due, for example, to inadequate identification of users.

4.2 Overview of security in distributed objects

4.2.1 Low level

At a low-level, at the OSI sessions level, the industry standard Secure Socket Layer (SSL) can be used to establish secure connections between clients and servers. The SSL protocol allows sensitive data to be transmitted over an insecure network, like the Internet, by providing the security features: authentication, privacy and integrity [17].

SSL is a protocol on top of TCP/IP, which adds security capabilities [24]. The SSL API is an extension to the TCP/IP socket API. SSL's security capabilities include encryption of the message sent through an SSL communication channel, authentication of the server based on *digital certificates* and *signatures*, and optional authentication of the client. Note, in SSL the client is the program that initiates an SSL connection and the server is the program that accepts the connection. The client and server participating in an SSL connection are also known as *peers*.

4.2.2 High level

SSL is only for use at the session level and not for higher levels, such as the application level. SSL can not secure "soft parameters", such as that the user is who he/she claims to be. SSL can only secure that the client and server machines are what they claim to be.

4.3 Security features

4.3.1 Authentication

Authentication is asking the question "Are You Who You Claim To Be?". A client can determine a server's identity and be certain that the server is not an impostor. Optionally, a server can also authenticate the identity of its client applications [17].

In SSL authorisation is based on *digital certificates* ***[24].*** A digital certificate is issued by a certificate authority. The certificate contains the name of the certificate authority, the name of the party who owns the certificate (and which is identified by it), the public key of this party, and time stamps. All this data is *signed* by the certificate authority, which means that the certificate can not be modified without this being noticed.

The information a certificate gives is that a name and a public key belong together. As that information can be public is it necessary to prove that whoever presents the certificates also has the private key corresponding to the public one contained by the certificate. *Digital signatures* are used to prove this, see Figure 35. The mechanism behind the digital signature involves creating random message and encrypting it with the private key. If the server wishes to authenticate itself with the client who holds its certificate, the server sends the message in clear text as well as the encrypted message to the client. The client can compare the clear text message with the result of decrypting the encrypted message using the public key it obtained from the certificate. If they match, the client knows that the public and the private key of the server match and has finally established the identity of the server.

4.3.2 Privacy

Data passed between the client and server is encrypted so that if a third party intercepts their messages, it will not be able to unscramble the data.

4.3.3 Integrity

The recipient of encrypted data will know if a third party has corrupted or modified that data.

4.3.4 Authorisation

Authorisation is asking the question "Are You Allowed to Use This Resource?". Once a client are authenticated, the server objects are responsible for verifying which operations the clients are permitted to perform on the information they try to access [6]. **Note**. SSL does not support security on this level.

4.4 Public-key encryption

SSL authentication is based on *public key cryptography*. Public key technologies use a pair of asymmetric keys for encryption and description. This means that a message encrypted with one key can only be decrypted using the other key of the pair, see Figure 35. Each user generates a *private key* and a *public key* and it is not possible to derive a user's private key from their public key. If someone wants to send sensitive data to a user, they acquire the users public key and use it to encrypt that data. Once encrypted, the data can only be decrypted by the person holding the users private key - the user himself. Not even the sender of the data will be able to decrypt the data.

One of the most popular public-key encryption algorithms is *RS*A [www.rsa.com], named after its inventors Rivest, Shamir and Adleman (1977). Data that has not been encrypted is often referred to as *clear-tex*t, while data that has been encrypted is called *cipher-tex*t.

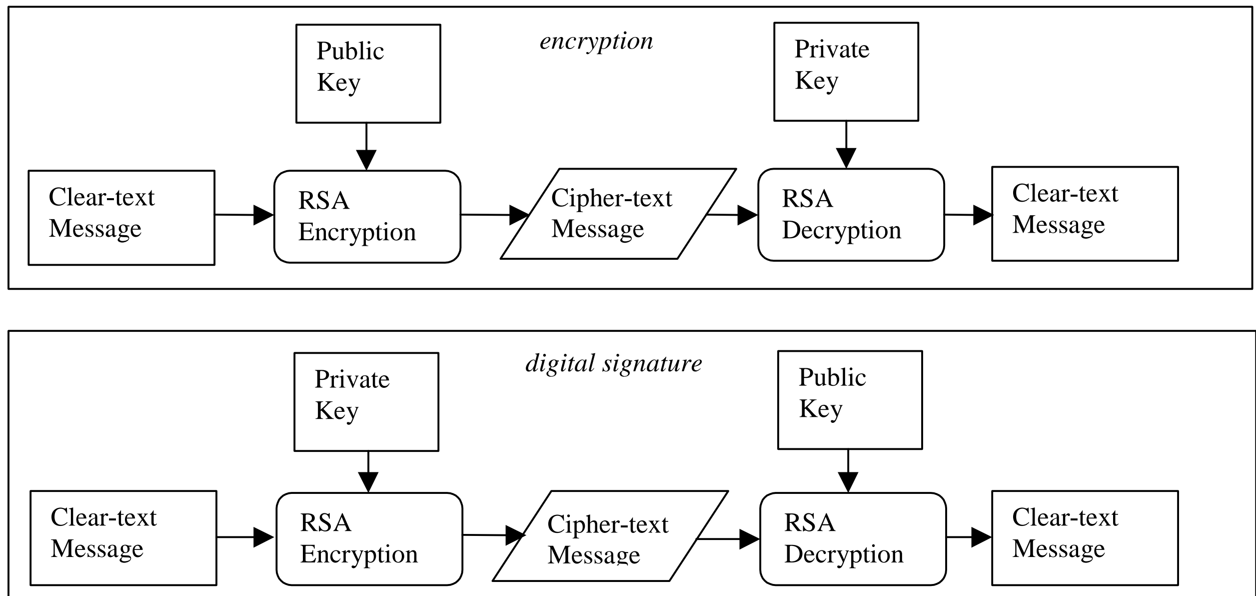

Figure 35. Public and private keys performing inverse function of one another.

4.5 Auditing

Security auditing assists in the detection of actual or attempted security violations [15]. This is achieved by recording details of security relevant events in the system, such as who has logged into the system, who transferred money to whom, authentication of principals, etc. Depending on implementation, recording an audit event may involve writing event information to a log, generating an alarm, or some other action. Audit policies specify which events should be audited under what circumstances. Auditing is not a part of the Master thesis.

4.6 Investigation of Corba/SSL products

One minor part of the first part of the Master Thesis was to investigate which Corba/SSL product to use. The investigation is found in Appendix B. Anyhow, the result of the investigation was to use Inprise VisiBroker with the SSL package.

4.6.1 VisiBroker SSL package

The VisiBroker SSL feature [17] is designed to be easy to use and transparent to developers of client and server applications. An existing VisiBroker client or server can be modified to use SSL by simply adding a few lines of initialization code. The presence or absence of SSL does not affect a client application's core code that invokes an object's methods and the server code that implements those methods. VisiBroker's SSL feature is also compatible with the use of threads and other VisiBroker facilities.

If writing a client application or server that is to use SSL, the programmer is responsible for understanding and obtaining the certificate chain and private key. If writing a client or server that inspects the certificates sent by a peer, the programmer must understand the syntax and semantics of certificate chains and the programmer must hold, in the program or in a file, the certificate chain and private key.

5 Migration of GEM from Java RMI to Corba/SSL

5.1 General method of porting

It should be very easy if there was an intelligent program that could do the porting. That is, unfortunately, not the case. There are, however, some general steps to follow to make the porting more smooth.

1. Locate all the remote interfaces and find out how and where Java RMI is used.
2. Analyse methods in a remote interface, this includes looking for in/out parameters, names, data types, etc.
3. Find out how the interfaces interoperate with each other. Look in a programmer's guide, if one is available, and in the code.
4. Write the remote interfaces in an IDL-file (see Figure 14) and compile the IDL-file. The generated files, per interface, can be seen in Figure 12.
5. Find out if a Java RMI class implements more than one interface. If that is the case it is necessary to use the Tie-mechanism in Corba (see 3.4.10), as <interface>POA is a class and Java does not allow to extend more than one class. When using the Tie-mechanism <interface>Operations is implemented.
6. Change all throws and RMI exceptions to Corba exceptions.
7. If objects need to be send by value decide to choose either Corba object-by-value or send the object parameters in an IDL struct or if possible, use a call-back routine.
8. Look at the server example code in Figure 25 or if using the Tie-mechanism, see Figure 30 for how to implement the Server side.
9. For implementing the Client side, see Figure 26.

5.2 The Analyse Phase

The GEM System consists of functional components that will be used in a framework designed for a specific system. That means that the components will be reusable in a new context for a new system. See Figure 6 in 2.4.6 for a view of all components in the GEM System.
The analyse phase is important for getting a view of how the GEM System is build and how it works. The code in the GEM System is written by using JBuilder and is divided into different projects, with different functionality. As the GEM System is a system for supervision the remote invocation is quite at lot, as can be seen in Figure 6 and Figure 36. The GEM source code is packaged with the generic code and the customer specific code separated. The generic code consists of 133 Java files and the customer specific code of 6 files (in a specific project). Many files need to be changed and it will also take some time to get a picture of the system. In the end it will be worth it because of the features that Corba has compared with Java RMI.

5.2.1 Client-Server communication in GEM

The communication between the client and the server is by using distributed interfaces. In Figure 36[22] the component *Communication* on the client-side makes it

---
[22] [1], Figure 4.

possible for the client components to communicate with the server components, without knowing that they (the client components) are communicating with a distributed object. The component *Communication* supplies proxy classes, which gets data from the components on the server-side. The client components can also subscribe to alarms by register themselves. The alarms will be received from the proxy classes. *Communication* is using Java RMI, but will - as one part of the Master thesis - be changed to Corba.

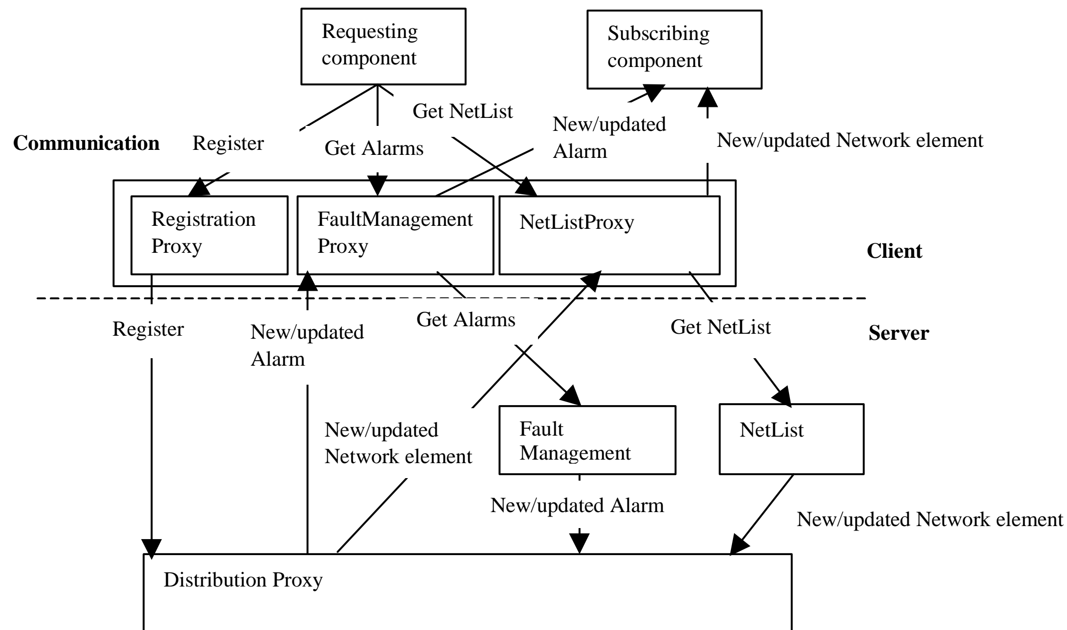

Figure 36. Client-Server communication

5.2.2 GEM Clients

At present there are three clients in GEM: Fault Management (FM), Security Management (SM) and System Administration (SA) [2], see Figure 6 in 2.4.6.

FM Client

FM Client consists of components used for storing and updating all alarms and network elements and presenting alarms and network elements in a tree view. A user can change status on an alarm (e.g. that an alarm has been corrected) and look at correlated alarms in the tree view.

Example of components:
se.ausys.gem.fm.gui.alarmdetails
se.ausys.gem.fm.gui.alarmdetails.interfaces
se.ausys.gem.fm.gui.alarmhistory
se.ausys.gem.fm.gui.alarmlist
se.ausys.gem.fm.gui.alarmlist.interfaces
se.ausys.gem.fm.gui.buttonpanel
se.ausys.gem.fm.gui.filterwindow
se.ausys.gem.fm.gui.fmtree
se.ausys.gem.fm.gui.fmtree.interfaces

The FM GUI components above communicate with the server components by using *Communication,* see 5.2.1.

SM Client
The SM Client component's responsibility is to present a GUI where a Security Administrator can add, edit and remove users in the system.

SA Client
The SA Client component's responsibility is to present a set of GUI:s where a System Administrator can view and update logs, cleared alarms and network elements.

5.2.3 GEM Server

The server is a standalone Java application encapsulating a variable number of server components. The server is designed depending on the needs of the specific customer. Each of the server components can easily be included or excluded in the final server. At present there are 8 server components [2], see Figure 6 in 2.4.6.

FM Component
The component *FaultManagement* contains the most of the FM functionality. New, deleted and updated alarms are send to the clients via *Communication.*

Netlist Component
This component controls the list of Network Elements.

Server communication Component
This component handles call-backs to registered clients. At present only the FM Clients uses this. Call-backs are made when new, deleted or updated alarms and networks elements needs to be distributed to registered clients.

SM Administrator
This component is responsible for the server part in the security management administration part of the system. This server handles all administration of users, passwords and roles.

SM Manager
This component is responsible for the server part in the security management manager part in the system. This server handles the login and access control in GEM.

SA Server
This component is responsible for the server part in the system administration part of the system. This server handles all administration of logs, cleared alarms, the network list and configuration of polling interval.

Database Component
The database component is responsible for implementing all the interfaces that are used by the different components in GEM that wants to save and/or retrieve information from the GEM database.

Log Component
This component is responsible for the server part in the log part of the system. The log consists of three different types, namely Alarm log (i.e. logs all alarm events, such as new alarm and update alarm), Security log (i.e. logs all the security events such as login, password change etc.) and System Administration log (i.e. removal of log entries etc.).

5.3 Design and Implementation

The design and implementation phases are written in the same chapter as they are very closely related because of updating an existing system.
The creation of the new GEM System starts in the design phase. It turned out that it definitely was not easy to do the migration of GEM from Java RMI to Corba. As mentioned in the Analyse phase the remote invocation is quiet a lot. Another thing is that the clients should be running as applet, but by some reason it never worked out initialising Corba with the applet (this is for further investigation). Therefore, the clients are running as application.

5.3.1 The Corba Development Process

To build and run an application using Corba the following steps are required [13]:
1. Write some IDL that describes the interfaces to the object or objects that will be used or implemented.
2. Compile the IDL file. This produces the stub and the skeleton code that implements location transparency, see Figure 12.
3. Identify the IDL compiler-generated interfaces and classes that will be used or specialised in order to invoke or implement operations.
4. Write code to initialise the ORB and inform it of any Corba objects that are created.
5. Compile all the generated code and the application code with a Java compiler, if using Java.
6. Run the distributed application.

5.3.2 Write and generate IDL files

The code style is rather similar between Java RMI and Corba. Both are using interfaces and both generate stub and skeleton code.
The main disadvantage by using Corba, instead of Java RMI, is that Corba indeed allow passing of objects by value (from Corba 2.3). But with RMI is it not necessary to do something special in the coding. In Corba is it necessary to use an IDL type named *valuetype* to make a mapping to a public Java class with the same name. A Java class supporting a *valuetype* must implement the java.io.Serializable interface. When implementing a *valuetype* the Tie-mechanism must be used [11].
In the Master thesis it is chosen not to use the object-by-value facility, but to pass the object parameters in a struct defined in an IDL file named CommonDeclare.idl. In Figure 37 a struct CorbaAlarm and a struct CorbaNetworkElement are declared for this reason.

```
#ifndef COMMONDECLARE_IDL

module commonDeclare {

  exception DbException {string msg;};

  struct CorbaAlarm{
    long alarmid;
    string distingueshedName;
    string probableCause;
    long long eventTime;
    long percievedSeverity;
    long state;
    string additionalText;
    string eventType;
    string comment;
    string ackSignature;
    long long ackTime;
    long long clearTime;
  };

  struct CorbaNetworkElement{
    string distinguishedName;
    string adress;
    string comment;
    boolean activated;
};
            ...

  typedef sequence <CorbaAlarm> alarmList;
  typedef sequence <CorbaNetworkElement> networkElementList;
  typedef long long Date;

  interface Client{
    boolean isAlarmSubscriber();
    boolean isNetlistSubscriber();
  };
};
#endif
```

Figure 37. A part of the new IDL file CommonDeclare.idl.

The "difficulty"with sending objects is illustrated in Figure 38.

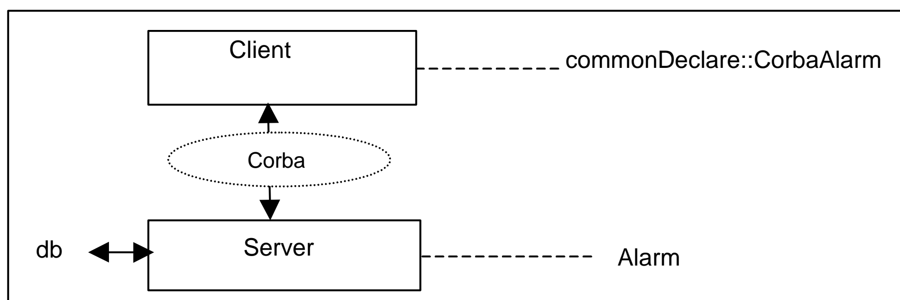

Figure 38. The Server side is using an Alarm object and the parameters in the object receive its data from the database (db). Corba is used between the Client and the Server. When sending alarm information between the Client and the Server is it necessary to place the object parameters in a struct CorbaAlarm that is defined in the file CommonDeclare.idl.

In the Master thesis four help classes are used for passing the object parameters: GetCorbaAlarm.java, GetAlarm.java, GetCorbaNetworkElement.java and GetNetworkElement.java. In Figure 39 is the help class GetCorbaAlarm.

```
package se.ausys.gem.common;
  import se.ausys.gem.common.commonDeclare.*;
  import se.ausys.gem.common.fm.Alarm;

  public class GetCorbaAlarm {
  static Alarm al = null;

  public CorbaAlarm getCorbaAlarm(Alarm alarm){
     al = alarm;

     // To make sure that no empty string is transmitted, as Corba does not allow that.
     if (al.getAckSignature() == null)
       al.setAckSignature("   ");
     if (al.getAdditionalText() == null)
       al.setAdditionalText("   ");
     if (al.getComment() == null)
       al.setComment("   ");
     if (al.getDistinguishedName() == null)
       al.setDistinguishedName("  ");
     if (al.getEventType() == null)
       al.setEventType("  ");
     if (al.getProbableCause() == null)
       al.setProbableCause("  ");
     if (al.getClearTime() == null)
       al.setClearTime(0);
     if (al.getAckTime() == null)
       al.setAckTime(0);

     CorbaAlarm cAlarm = new CorbaAlarm(al.getAlarmId(),al.getDistinguishedName(),
                                        al.getProbableCause(),al.getEventTime().getTime(),
                                        al.getPercievedSeverity(),al.getState(),
                                        al.getAdditionalText(),al.getEventType(),al.getComment(),
                                        al.getAckSignature(),al.getAckTime().getTime(),
                                        al.getClearTime().getTime());
     return cAlarm;
  }
}
```

Figure 39. Help class, GetCorbaAlarm, to pass parameters from Alarm to CorbaAlarm.

Valuetype

If *valuetype* was used the IDL file should look like this [11]:

```
ValueType AlarmType{
    long alarmid;
    string dName;
        void setDistinguishedName(in string dName);
    factory create(in long alarmid, in string dName, in string pCause, in long long eventTime,
                in long perceivedSeverity, in long state, in string addText, in string eventType,
                in string comment, in string ackSign, in long long ackTime,
                in long long clearTime);
     ...
};
```

Implement the *valuetype* by inheriting the *valuetype* base class:

```
public AlarmTypeImpl extends AlarmType{
    ...
    public void setDistinguishedName(String dName){
            /* User code. */
    }
};
```

It is also necessary to implement a Factory class to implement any factory methods defined in IDL[23]:

```
class AlarmTypeDefaultFactory implements AlarmTypeFactory{
    public AlarmType create(int alarmId,String dName,String pCause,long eventTime,
                            int percievedSeverity,int state,String addText,
                            String eventType,String comment,String ackSign,
                            long ackTime,long clearTime){
        /* Just an example */
        AlarmType obj = new AlarmType();
        obj.setDistinguishedName(dName);
        obj.setAdditionalText(addText);
                ...
        return obj;
    }
};
```

Change name of methods
In IDL is it not possible to have methods with the same name and with different numbers of parameters. As it is possible in Java, some methods had to change name (i.e. change from getClearedAlarms(int size,int order,Date from, Date to) to getClearedAlarmsDate(int size,int order,Date from, Date to)).

5.3.3 Using Name Service

VisiBroker provides a Smart Agent architecture that provides high availability and fault tolerance needed in mission critical applications [11]. When a server binds an object to a particular name the name, location, and reference to the object is maintained by Smart Agent. Anyhow, Smart Agent may not be use in the Master thesis as the GEM System should be as vendor independent as possible. Instead Corba Name Service is used, see example code in 3.4.9. As Smart Agent should not be used is it necessary to set it to disable, -Dvbroker.agent.enableLocator=false. Otherwise the Smart Agent (osagent) must be running to bootstrap the Naming Service[24]. When starting the Name Service it can be a good idea to start it on a particular port, for example port 20000.

[23] A factory object is a server object that handles creation and destroy of other server objects (instance the object, initialise it, register or unregister it at POA and releases it) [21], slide 5-8.
[24] A Naming Server needs to register itself with the Smart Agent during starting up (at least with VisiBroker 4.1). This allows clients to retrieve the initial root context by calling the *resolve_initial_references* method [12], Chapter 18, Using the Naming Service.

Example when starting the Name Service:
vbj -Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=20000
com.inprise.vbroker.naming.ExtFactory -Dvbroker.agent.enableLocator=false

When running the program in JBuilder is it also necessary to add information to the Application Parameter:
-ORBInitRef NameService=corbaloc::localhost:20000/NameService

When running the program from shell:
%java.home%\bin\java se.ausys.custspec.main.server.MainServerFrame -ORBInitRef NameService=corbaloc::localhost:20000/NameService

This information is what will be send to main(String[] args). All clients need to get this information, so that they can use the Name Service for finding required objects.

5.3.4 Using the Tie-mechanism

As Java does not allow multiple class inheritance and that is needed when migrate GEM to Corba, the Corba Tie-mechanism is used. See 3.4.10 for more information.

5.3.5 Using Callback

Although, it is more common that a client just act as a client (e.g. calls a server) and a server acts as a server (e.g. receive calls from client) the opposite is sometimes necessary. The technique, where the server calls the client is named call-back. The client creates a Corba object and sends the object's reference to the server.

Call-backs, in general, are normally used when:

- a client don't have the time to wait for a reply
- the client subscribes to data from the server (i.e. updates of alarms)
- a kind of asynchronous transfer/reply is required

This technique is in use in GEM. A client informs the server that it will subscribe on either the network element list or the alarm list.
Instead of using call-back a Corba service can be used, either Corba Notification Service or Corba Event Service, see 3.4.7. Corba use a synchronous remote procedure call, but on the application level the named services are set as standards for asynchronous communication.

5.3.6 SSL Security Service Package

The certificate, see 4.3 and 4.4, is obtained by constructing a certificate request and sending it to a Certificate Authority (CA). In the Master thesis the CA is from Microsoft. The public and private key pair and the certificate request (i.e. cert_req) are generated in the tool Certificate Request Generator in Inprise Application Server Console.

5.4 Test procedure of GEM with Corba/SSL

During, and after, the migration of GEM from Java/RMI to Corba/SSL the functionality of GEM was tested by doing the operations named in the User Manual [3].

First, GEM was installed by following the instructions in the Installation manual [4] and the compliment to the Master thesis [5]. It includes starting the database (i.e. InterBase) and the Naming Service. Some data about the system was already stored in the database, such as the user role Security Administrator.
Second, the server was started and third, the client was started and a login window pops up. The first user role, see 2.4.7, to test was the Security Administrator. A Security Administrator has the right to work with the three main functions in GEM: Fault Management (e.g. acknowledge and clear alarms), Security Management (e.g. add, edit, block, activate, remove user) and System Administration (e.g. add/remove network elements, maintain application log).
Theses main steps were followed:

1. Change password for this Security Administrator.
2. Start a Security Management client.
3. Start a System Administration client.
4. Start a Fault Management client.

When knowing that all operations was working fine the other two user roles were tested. Primarily to see that the system did not let them do anything they were not suppose to do, such as an operator is not allowed to start a Security Management client (i.e. insert a user).

These main steps were followed for a System Administrator:

1. Change password for this System Administrator.
2. Start a System Administrator client.
3. Start a Fault Management client.

These main steps were followed for an Operator:

1. Change password for this Operator.
2. Start a Fault Management client.

Some data had to be inserted manually into the database, such as alarms. It is normally the Network Adapter, see Figure 6, that generates alarms. But as the Network Adapter is not a part of the Master thesis is it not running during this test.

The result of testing the functionality is that the modified version of GEM has exactly the same functionality as the previous version of GEM.

6 Design of GEM Security Model

This is the second part of the Master thesis. GEM Security Model consists of the part in GEM that deals with authentication and authorisation. The security model contains rules about what is allowed in the system. All data in GEM (e.g. network elements, alarms, users, logs) are stored in a database, see Figure 6, and to prevent that anyone can get access to any data is it necessary to have rules, i.e. policies. The database is a relational database, which consists of relations between the stored data.

Implementation of GEM Security Model is out of scope of this Master thesis.

6.1 Security Management

The typical telecom solutions have been running in closed network environments, with preparatory protocols and no access to such external networks as the Internet, and with each element manager in the network having its own user database and access control system.

With an increasing portion of operation and maintenance traffic using the Internet as a backbone, security management has become more important [22].

6.2 Design processes and tools

6.2.1 Rational Unified Process

During the design of the Security Model the Rational Unified Process (RUP) where followed. RUP is a software engineering process that enhances team productivity and delivers software best practices via guidelines, templates, and tool guidance for all critical software development activities. Its goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget [7]. RUP consists of four phases, see Figure 40:

- *Inception* – Define the scope of project.
- *Elaboration* – Plan the project, specify features, baseline architecture.
- *Construction* – Build the project.
- *Transition* – Transition the product into end user community

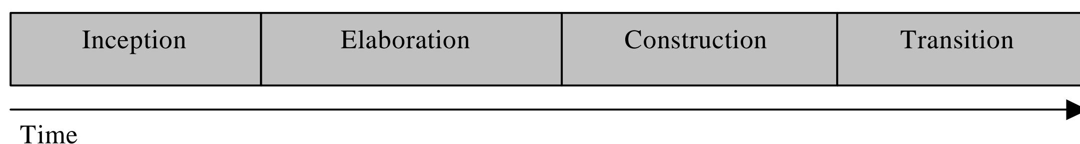

Figure 40. Process Structure – Lifecycle Phases

Use-case model
The *requirements* of the system are captured in a use-case model, see Figure 41.

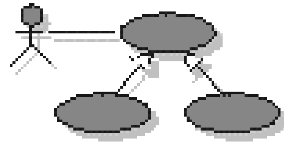

Figure 41. Use-case model.

The use-case model is a model of the system's intended functions and its environment, and serves as a contract between the customer and the developers. The use-case model is used as an essential input to activities in analysis, design, and test.

Following is a set of questions that are useful when identifying use cases:

- For each identified actor, what are the tasks in which the system would be involved?
- Does the actor need to be informed about certain occurrences in the system?
- Will the actor need to inform the system about sudden, external changes?
- Does the system supply the business with the correct behaviour?
- Can all features be performed by the identified use cases?
- What use cases will support and maintain the system?
- What information must be modified or created in the system?

Iteration plan
Each phase in RUP can be further broken down into iterations. An iteration is a complete development loop resulting in a release (internal or external) of an executable product, a subset of the final product under development, which grows incrementally from iteration to iteration to become the final system.

Benefits of an Iterative Approach
Compared to the traditional waterfall process, the iterative process has the following advantages:

- Risks are mitigated earlier
- Change is more manageable
- Higher level of reuse
- The project team can learn along the way
- Better overall quality

6.2.2 Rational Rose

Rational Rose [9], by Rational, is a very useful visual modelling tool. The modelling is based on the Unified Modelling Language™ (UML) [8]. The UML is the standard notation for software architecture, which means that it is possible by using Rational Rose to communicate with one language and one tool.

6.3 Design phases

When designing the GEM Security Model following design phases are used:

1. Identifying use cases, see section 6.2.1.
2. Look in the existing GEM Security Model how authentication and authorisation are done.
3. Research for what have already been done at AU-System, such as another security model.
4. Database design.

During *research* it was found that a security model already had been designed in another project at AU-System. That project had also followed RUP and there were several Use Case Specifications in requirement specification documents, and a Rational Rose model. Some of the use cases were ideal for use in the GEM Security Model and they are copied into Appendix D, which contains all use cases and scenarios in more detail. Auditing (i.e. UC 52, UC 53, UC 63), is not a scope of the Master thesis, but of importance in a Security Model so therefore are they mentioned in the appendix.

6.4 The existing GEM Security Model

In the existing GEM Security Model authentication and authorisation is very simple. The *authentication* is done by using Java Authentication and Authorisation Service (JAAS) [19], which is plugged-in to the code. Implementing Corba/SSL is definitely an improvement.
The *authorisation* is very rudimentary:

- it is only based on the user's role (i.e. operator, system administrator or security administrator). "Is an *operator* allowed to do this operation?"
- the authorisation check is done in the code by using *if* statement:
  if *user* equals to "*Security Administrator*" or "*System Administrator*"
  delete selected *network element*

In a (large) system for supervision this authorisation is rather inflexible and complex and makes authorisation more difficult to administrate, for example is *every* System Administrator allowed to do all System Administrator operations (e.g. delete network elements) in the system. To prevent disorder in the system, such as that someone deletes an important network element, there must always be a consensus between the actors in the system before a more severe operation can be done.

There are some steps that can improve the administration and the security in the existing GEM Security Model:

- Administration will be easier by dividing users into different *user groups* and connect each user group (and user role) to a *policy rule*, see 6.5. By using policy rules the security in the system will increase, see 6.5. The policy rule contains all network elements that the user group are allowed to make either a write or a read operation on. If some conditions changes, such as a user group is not allowed to read a network element, the user group will be connected to another policy rule. In this case the question is "Is the Liljeholms group allowed to delete this network element?"
- Instead of using more or less complex compare statements in the code, is it better to do the authorisation check by using a relational database, see 6.8.1 and SQL questions. This makes changes in the system more easy and flexible.
  If users are grouped into user groups these changes are necessary to do in the code, example:

if (*user* in *usergroup)* and (*usergroup* is allowed to *add network element*)
add network element

6.5 Security policies

GEM has three different kinds of user roles: Security Administrator, System Administrator and Operator. A user is, depending of its role, allowed to do certain operations, according to the Policy Rule.

6.5.1 Access policy architecture

The use cases *Configure Network Policies*, *Access Control* and *Modify Security View,* see 6.6, all have to do with access (security) policy.

Problems to solve
There are lots of different access policies that must be possible to implement in the security model. Examples:

- John, who is a network administrator, is allowed to upgrade all routers in his own region.
- Ben, who is a network administrator, is allowed to do anything in the office network.
- Elisabeth, who works at customer support, is allowed to browse the network status.
- All network operators are allowed to do non-critical operations on all network elements in their own region.

How to find the correct policy
A user may have many policies, depending on what role he assumes and what he tries to do.
The approach in the Master thesis is to connect policies to users, user groups and/or user roles. To use groups or roles is more scalable, but, for example, with few users a user connected security policy may be useful.

The approach for finding a *user connected security policy*:

1. Check if the user has any security policies connected to him. If none is attached, proceed with finding group connected or role security policies. If found proceed to 2, otherwise deny the request.
2. Find if any of the connected security policies satisfies the current conditions. If no security policy fulfils the conditions, deny the request. Several security policies may be fulfilled, but the request is granted when the first fulfilled policy is found.

The approach for finding *a group* or *role based security policy*:

1. Check if the user belongs to any groups or roles by checking the "group/role-membership" attribute. If the user does not belong to any deny the request, otherwise proceed to 2.
2. Find if any of the connected security policies satisfies the current conditions. If no security policy fulfils the conditions, deny the request. Several security policies may be fulfilled, but the request is granted when the first fulfilled policy is found.

The search for user policies is carried out before the search for group/role-connected policies. User connected policies overrides group/role-connected policies.

6.6 Use cases and scenarios in GEM Security Model

Use cases in the scope of the Master thesis for GEM Security model are listed in Table 1. They are classified in three different groups, according to their functionality:

1. Basic functionality (UC24, UC25,UC28,UC29 and UC30)
2. Grouping (UC64,UC65 and UC66)
3. Authorisation (UC31,UC62 and UC67)

Group 1 is already defined and implemented in GEM, and Group 2 and 3 have to do with security policy.

Use Case	Status
UC 24 Login to the System	Already in GEM
UC 25 Change own password	Already in GEM
UC 28 Add a User	Already in GEM
UC 29 Remove a User	Already in GEM
UC 30 Modify a User	Already in GEM
UC 31 Configure Network Policy	Will be designed
UC 62 Access Control	Will be designed
UC 67 Modify Security View	Will be designed
UC 64 Add User Group	Will be designed
UC 65 Modify User Group	Will be designed
UC 66 Remove User Group	Will be designed

Table 1. Use cases for GEM Security Model.

6.6.1 A representative use case

UC 31 Configure Network Policy

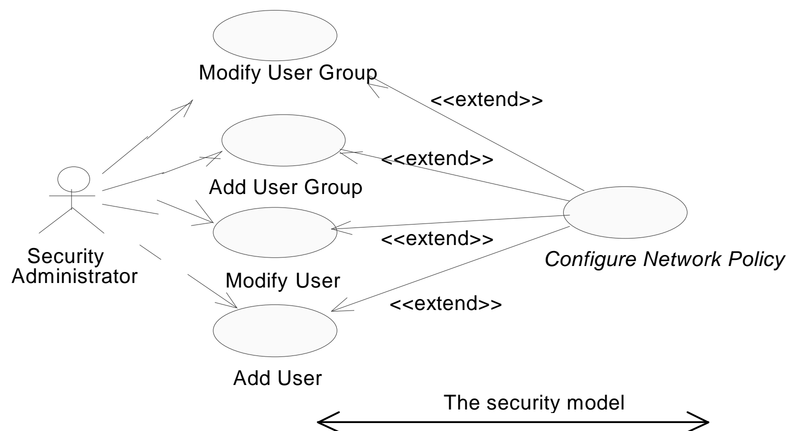

Figure 42. Use case diagram for Configure Network Policy.

This use case is extended from UC 28 Add User, UC 30 Modify User, UC 64 Add User Group and UC 65 Modify User Group. The *goal* for this use case is to define the access rights on security view[25] groups for users or user groups.

A *network policy configuration* is a collection of rules that determines the user's, or the user group's, access rights on groups of network elements. The Security Administrator defines the network policy configuration for every user and user group. The Security Administrator may copy an already existing network policy from another user or user group. Every user and user group has a unique network policy configuration.

The network elements are grouped in the security view. Each such group has access rights, which can be either "None", "View" or "Control". A new user or user group added to the system has the access rights "None" for all security view groups.

A user that is a member of a user group can have "None" access rights for all security view groups and instead use the access rights set in the network policy configuration for her user group.

Flow of Events
This use case begins when it is invoked from one of its base use case.

A. Basic Flow

1. The system displays the configure network policy dialog for the user or user group. See Figure 43 for an example of the user interface.

[25] The Security Administrators use the security view to set access rights on the security view groups for users and user groups. The security view is structured like a tree with security view groups as branch nodes and network elements as leaf nodes. A security view group can not consist of both sub-level groups and network elements, i.e. groups and network elements can not be siblings in the security view tree.

2. The security view for the owner of the network policy is shown with the access rights.
3. The Security Administrator sets the access rights on security view groups by selecting and giving them the wanted access rights.
4. When the Security Administrator commits the changes, if any, the network policy is stored for the user / user group and the dialog shuts down. The use case ends.

B. Alternative Flows

Copy Network Policy from another User or User Group

This alternative flow is followed when the Security Administrator wants to copy an already existing network policy from another user or user group.

1. Step 1 – 2 from the basic flow above.
2. The Security Administrator activates the copy network policy function.
3. The system displays a new dialog that lists all users and user groups. See Figure 44 for an example of the user interface of the copy network policy dialog.
4. The Security Administrator selects one of the users or user groups to copy from.
5. The copy network policy dialog shuts down and in the configure network policy dialog the access rights are set on the security view groups just like the copied network policy.
6. Step 3 – 4 in the basic flow.

Special Requirements for Access Rights

[UC31-req.1]	All security view groups have the access rights “None” as default. That is, a new user has “None” access rights for all groups in the security view before any configurations of the network policy have been made.
[UC31-req.2]	In the security view all security view groups are noted with the policy owners access rights. When a security view group is noted with, i.e. “Control”, you know that all child groups of that also have “Control” access rights.
[UC31-req.3]	If security view groups with the same parent group, i.e. sibling groups, have different access rights, the parent group gets the notation “Mixed”. That means that you must expand the group to find out the access rights for the child groups.
[UC31-req.4]	Access rights can only be set for groups in the security view, not on separate network elements.
[UC31-req.5]	**Highest permission rules** When network policy conflicts arise, the network policy configuration with the highest access rights is valid. Since a user can be a member of several user groups, and every user and user group has its own network policy configuration, these conflicts are likely to arise. For instance, the user *u1* is member of the user groups *ug1* and *ug2*. If *u1* wants to perform an operation on some object, it is enough that one of the policies of *u1*, *ug1* and *ug2* has the access rights to perform the operation.

Picture of the User Interface

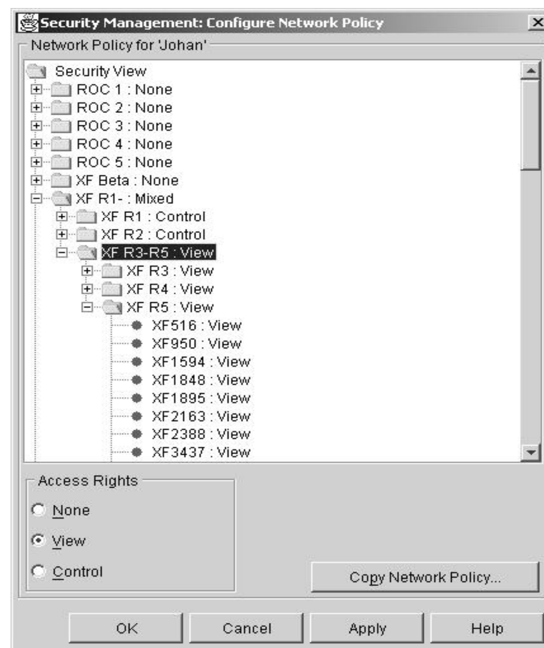

Figure 43. Example of Configure Network Policy dialog.

The above example shows the network policy for the user Johan. The Security Administrator has given “Johan” the access rights “Control” for the groups “XF R1” and “XF R2”. “XF R3-R5” has been set to “View” which entails that the child groups “XF R3”, “XF R4” and “XF R5” also has been set to “View”. Since Johan has different access rights for “XF R1” and “XF R2” than “XF R3-R5” the parent group of those get the notation “Mixed”.

Note. The access rights are the same for all the network elements in a security view group since there is not possible to set the access rights for separate network elements.

Figure 44. Example of Copy Network Policy dialog.

6.7 Iteration plan

The Iteration plan describes the plan for design of the security model use cases.

Iterations

The use cases are divided in 2 iterations.

The plan

Use Case	Iteration 1	Iteration 2	Time (days)
UC 24 Login to the System	Ready		-
UC 25 Change own password	Ready		-
UC 28 Add a User	Ready		-
UC 29 Remove a User	Ready		-
UC 30 Modify a User	Ready		-
UC 31 Configure Network Policy		Start/Ready	2
UC 62 Access Control		Start/ready	2
UC 64 Add User Group	Start/Ready		1
UC 65 Modify User Group	Start/Ready		1
UC 66 Remove User Group	Start/Ready		1
UC 67 Modify Security View		Start/Ready	2

Table 2. The iteration plan for design of security model use cases.

The number of expected working days are 9.

6.8 Relation Database Design

6.8.1 Relation Model Concepts

All data in the system (e.g. network elements, users) are stored in a relation database [28]. The relation model represents the database as a collection of *relations*. Informally, each relation resembles a table of values and each row in the table represents a collection of related data values.
The table name and column names are used to help in interpreting the meaning of the values in each row, see Table 3.

UserMembership	GroupName	UserName
	Liljeholmen	KAAL
	Liljeholmen	KAAN

Table 3. Table with table name UserMembership and column names GroupName and UserName.

In the formal relation model terminology a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*.

6.8.2 Conceptual Schema

The new security model for authentication and authorisation in GEM is structured in a conceptual scheme [29], see Figure 45. The scheme describes use cases *UC31 Configure Network Policy*, *UC 62 Access Control, UC 64 Add User Group, UC 65 Modify User Group, UC 66 Remove User Group* and *UC 67 Modify Security View*.

Ideas who to structure:
Read the use cases careful and take notes. Three use cases are more difficult to structure: UC 31 Configure Network Policy, UC 62 Access Control and UC 67 Modify Security View.

UC 31 Configure Network Policy

- Every user and user group has a unique policy configuration.
- Network elements are grouped in the security view. The access rights are set for each group in the security view.
- Different types of access rights: "None", "View" and "Control". "None" is default for new user and user group.
- The Security Administrator defines the network policy configuration for every user and user group. He may also copy an already existing network policy from another user or user group.
- A user that is a member of a user group can have "None" access rights for all security view groups and instead use the access rights set in the network policy configuration for her user group.

UC 67 Modify Security View

- The security view consists of groups, which consists of sub-level groups or network elements.
- The security view is defined and configured by the Security Administrator.
- The Security Administrator use the security view to set access rights on the view groups for users and user groups.
- A security view group can not consist of both sub-level groups and network elements, i.e. groups and network elements can not be siblings in the security view tree.

UC 62 Access Control
This use case requires that both UC 31 and UC 32 have been structured.

1. Check the users role policy.
2. Check the users network policy.

The notation for *mapping constraints* between the objects is using the quadruple (m|1,m|1,t|p,t|p). The first component of the quadruple indicates whether the object's attribute is *single valued* or not. The second component indicates whether the attribute is *injective*. The third component indicates the *totality* of the attributes. The fourth component indicates the *surjectivity* of the attribute.

Example: A *User* has exactly one *Role* (component 1 and 3), several *Users* may have the same *Role* (component 2), and there are *Roles* that no one has (component 4). This gives the quadruple (1,m,t,p).

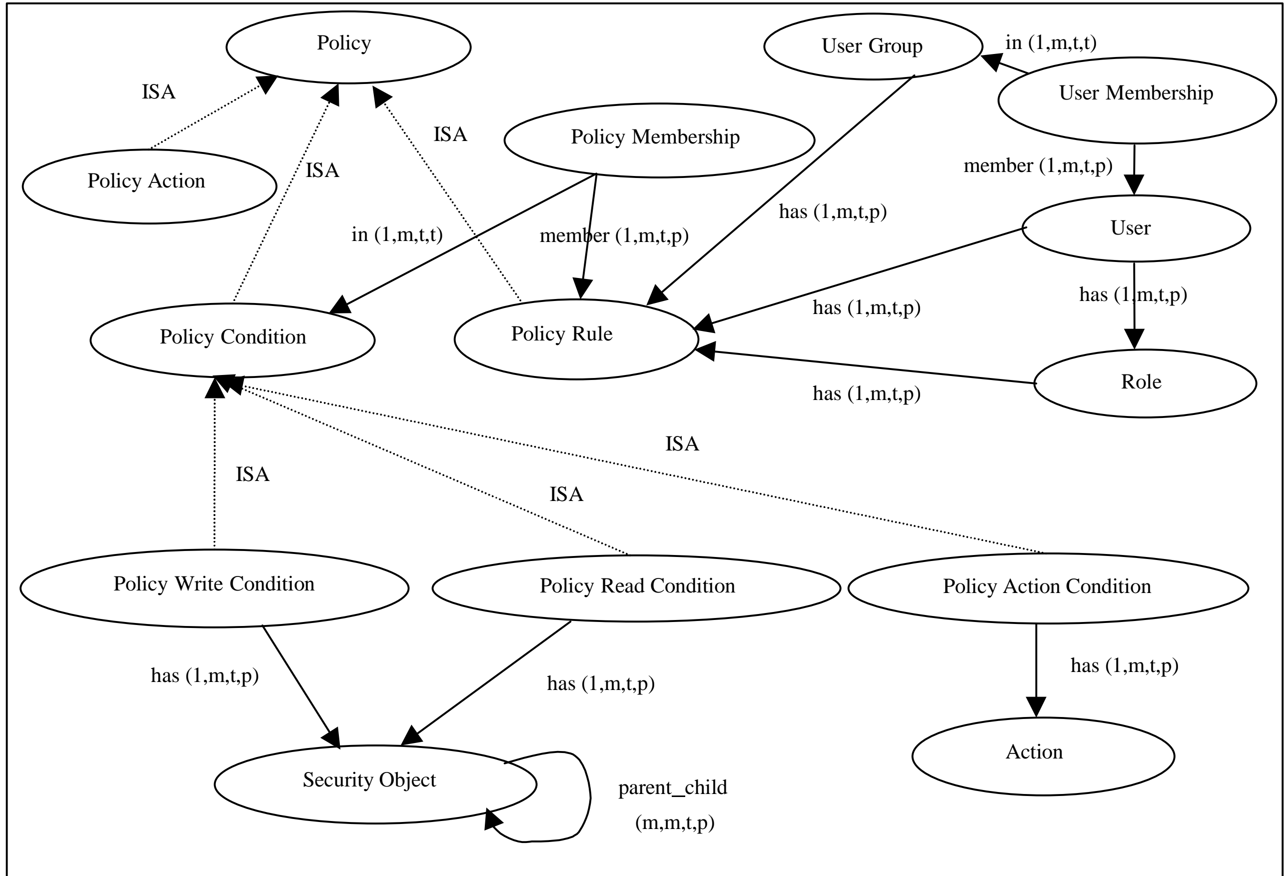

Figure 45. Conceptual schema for the GEM Security model.

Table 4 gives a description for every object in the conceptual schema.

Objects	**Description**
Policy	Base class for policy related classes.
Policy Action	Represents a policy action, i.e. an action to be taken *after* one or several policy conditions has been fulfilled.
Policy Condition	Base class for different policy conditions.
Policy Membership	Represents "connection" between Policy Condition and Policy Rule.
Policy Rule	Points out all conditions and actions for a user, a user group or a role.
Policy Write Condition	Represents a policy condition for a user or a user group. This policy condition points out all NE:s to which the user or group has control access.
Policy Read Condition	Represents a policy condition for a user or a user group. This policy condition points out all NE:s to which the user or role has access.
Security Object	Represents a NE in the EM.
Policy Action Condition	Represents *role related actions*, i.e. actions that are not related to a NE.
Action	Represents role-related actions, i.e. each role has an instance of this class, which points out the role actions

	this role can do.
User Group	Represents a user group in the EM.
User Membership	Represents "connection" between User and User Group.
User	Represents a user in the EM.
Role	Represents a role in the EM.

Table 4. Description for every object in the conceptual schema.

6.8.3 Relation Database Schema

A relation schema is used to describe a relation. A relation schema **R**, denoted by R(A1,A2,..., An), is made up of a *relation* R and a list of *attributes* A1,A2,...,An.

Keys are used to access the table, see Table 3. *Primary keys* uniquely identify a row in a table, while *foreign keys* access data in other related tables.

Following are two tables, with each object in Figure 45 written as a relation schema. For example:
UserGroup((g_name),p_role), there UserGroup is the name of the *relation* and g_name and p_role are *attributes*. g_name is the *primary key*.

Tables (Primary keys in parentheses)

--

Policy base class and its three different policies:
Policy((p_type))
PolicyAction((p_action),p_type)
PolicyRule((p_role),p_type)
PolicyCondition((p_cond),p_type)
PolicyMembership((p_role,p_cond))

User, user groups and their role and policy:
User((u_name),setting_id,full_name,activated,pwd,p_role,r_name)
UserGroup((g_name),p_role)
UserMembership((g_name,u_name))
Role((r_name),p_role)

The three different policy conditions:
PolicyReadCondition((p_cond),obj_name)
PolicyWriteCondition((p_cond),obj_name)
PolicyActionCondition((p_cond),action)

The security views sub-level groups and network elements, see also Figure 46:
SecurityObject((name),type)
ParentChild((parent,child))
NetworkElement[26]((ne_name),dist_name,comment,adress,activated)

[26] This relation is not in the conceptual schema.

Security Object	name	type
	ROC1	false
	XF1222	true

Parent-Child	parent	child
	ROC1	XF122

Network Element	ne_name	dist_name	comment	address	activated
	XF1222	ROC1-XF1222	Important	192.167.122.45	true

Figure 46. Graphic view of some tables.

Foreign Keys

PolicyAction.p_type << Policy.p_type
PolicyCondition.p_type << Policy.p_type
PolicyRule.p_type << Policy.p_type

UserMembership.g_name << UserGroup.g_name
UserMembership.u_name << User.u_name
User.r_name << Role.r_name
UserGroup.p_role << PolicyRule.p_role
User.p_role << PolicyRole.p_role
Rule.p_role << PolicyRole.p_role

PolicyMembership.p_role << PolicyRole.p_role
PolicyMembership.p_cond << PolicyCondition.p_cond

PolicyReadCondition.p_cond << PolicyCondition.p_cond
PolicyWriteCondition.p_cond << PolicyCondition.p_cond
PolicyActionCondition.p_cond << PolicyCondition.p_cond

ParentChild.parent << SecurityObject.name
ParentChild.child << SecurityObject.name

NetworkElement.ne_name << ParentChild.child

7 Conclusions and Future work

Conclusion

In the Master thesis is presented how to migrate a system that were using Java RMI for the remote communication, in a distributed system, to use Corba instead. It turned out that Corba is more complex and needs more coding than Java RMI, but as Corba is, among other things, language and platform independent is it possible to create more flexible infrastructures using Corba. That is definitely a demand from vendors/companies, as it makes the infrastructure more cost effective and possible to do rapid changes.
Security in an open system is very important and the difficulty of designing a security model has been shown in the Master thesis. In the existed version of GEM Security Model authorisation was very rudimentary. Authorisation was only based on the user's role and authorisation check was done in the code, by using if-statement. In the modified version of the security model users can belong to a group and this user group has a policy rule. Authorisation check is done by using a relational database. These modifications makes administration easier and more flexible and increases the security in the system. During design the Rational Unified Process has been followed.

As a request from their customers, AU-System has got a modified system for supervision there Corba is used for remote communication. They will use this modified version in their further development of GEM.

Future work

A future work in the Security Model is to add audit functionality. Audit functionality is very important in a network, for example to detect intruders.

Problems

During the Master thesis there were several problems, some was quite easily fixed and some took longer time. Here are some of the last mentioned:

- When testing the modified GEM version, a Corba exception error sometimes appeared when reading data from the database. It turned out that this happened when a field in the database was empty and data was send to a client. Corba does not allow empty data, and as object parameters are passed in a struct, see 5.3.2, an exception was generated. This is mentioned in Corba books!
- The clients should be running as applets, but by some reason it never worked out. This Corba exception was generated:
  org.omg.CORBA.INITIALIZE: can't instantiate default ORB implementation se.vbroker.orb.ORB minor code: 0 completed: No
  at org.omg.CORBA.ORB.create_impl(ORB.java:286)
  at org.omg.CORBA.ORB.init(ORB.java:365)
  ...
  According to books and the Internet all configurations and the parameters send to initialise the applet in Corba is correct. However, there could be an environmental problem. A question was send to borland.public.visibroker newsgroup, but no answer. Someone else had the same problem, and had just got an already known answer. This is under further investigation. The clients are currently running as applications.

- Got the wrong version of SSL Security Service Package from LinSoft. It was not possible to install it, and the generated error code was not helpful in this case. Borland were very helpful and had lots of different suggestions on the environmental configuration. The error was that the version was not compatible with the used version of Visibroker, which was mentioned when ordering.

Abbreviations

API	Application Programming Interface
Corba	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interfaces
DSI	Dynamic Skeleton Interfaces
ESIOP	Environment Specific Interoperability Protocol
GEM	General Element Manager
GIOP	General Inter-ORB Protocol
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IIOP	Internet Interoperability Protocol
IOR	Interoperable Object Reference
IR	Interface Repository
ITU	International Telecommunication Union
J2SE	Java 2 Standard Edition
JAAS	Java Authentication & Authorization Specification
JVM	Java Virtual Machine
O&M	Operation and Maintenance
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
PKI	Public Key Infrastructure
POA	Portable Object Adapter
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RUP	Rational Unified Process
SQL	Structured Query Language
SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modelling Language

References

[1] AU-System. Generic Element Manager. Version PA1.
[2] AU-System.1st of June 2000. *Programmers Guide - Generic Element Manager*. Document number: 21908065. Revision A.
[3] AU-System. 9th of April 2000. *GEM User Manual*. System Release 1.0.
[4] AU-System. 20th of April 2000. *Installation av GEM*. System Release 1.0.
[5] AU-System. 4th of June 2001. *Configuration, installation and start of GEM(Generic Element Manager) Corba*. Revision A.
[6] Orfali, Robert. Harkey, Dan. Edwards, Jeri. 1996. *The Essential Distributed Objects Survival Guide.* ISBN 0-471-12993-3, United States of America. John Wiley & Sons, Inc.
[7] Rational. 2001. *Rational Unified Process*. URL:http://www.rational.com/products/whitepapers/100420.jsp [2001, March]
[8] Rational. 2001. The UML and Data Modeling. URL:http://www.rational.com/products/whitepapers/101516.jsp [2001, March]
[9] Rational. 2001. *A Rational Approach to Software Development Using Rational Rose 4.0*. URL: http://www.rational.com/products/whitepapers/293.jsp [2001, March]
[10] Sun Microsystems. Java RMI Tutorial. URL:http://www.javasoft.com/docs/books/tutorial/rmi/index.html [2001,February]
[11] Siegel, Jon. 2000. 2nd edition *CORBA 3 Fundamentals and programming*. ISBN: 0-471-29518-3. United States of America. John Wiley & Sons, Inc.
[12] Inprise. *VisiBroker for Java 4.5 Programmers guide* URL:http://www.inprise.com/techpubs/books/vbj/vbj45/programmersguide/contents.html [2001, spring]
[13] Vogel, Andreas. Duddy, Keith. 1998. 2nd edition. *Java Programming with CORBA.* ISBN 0-471-24765-0. United States of America. John Wiley & Sons, Inc.
[14] Object Management Group. 1999. *The Common Object Request Broker: Architecture and Specification* Revision 2.3.1. URL:ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf [2001, spring]
[15] Object Management Group. May 2000. *Security Services Specification* URL:ftp://ftp.omg.org/pub/docs/formal/00-06-25.pdf [2001, spring]
[16] Object Management Group. November 2000. *Interoperable Naming Service Specification*. URL: ftp://ftp.omg.org/pub/docs/formal/00-11-01.pdf [2001, spring]
[17] Inprise. VisiBroker SSL Pack. *Programmer's Guide*. Version 3.3. URL:http://www.borland.com/techpubs/books/addons/vbssl/vbssl33/pdf_index.html [2001, spring]
[18] Suresh Raj, Gopalan. 1998. *A Detailed Comparison of CORBA, DCOM and Java/RMI* URL: http://www.execpc.com/~gopalan/misc/compare.html [2001, spring]
[19] Sun Microsystems. Java™ Authentication and Authorization Service (JAAS) 1.0 Developer's Guide. URL: http://java.sun.com/security/jaas/doc/api.html [2001, spring]
[20] Chung, P. Emerald. Huang, Yennun. Yajnik, Shalini. Liang, Deron. Shih, Joanne C. Wang, Chung-Yih. Wang, Yi-Min. (No Date) *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer* URL:http://www.cs.wustl.edu/%7Eschmidt/submit/Paper.html [2001, spring]
[21] Init, Course binder 29/1-2/2-01,*Distribuerade objekt med CORBA*.
[22] AU-System, May 2001, *Security Management*.
[23] MowBray Thomas J. Malveau Raphael C. 1997. *CORBA Design Patterns.* ISBN 0-471-15882-8. United States of America. John Wiley & Sons, Inc.

[24] Vogel Andreas. (No Date). *Building Enterprise Applications for the Net with EJB, CORBA and XML*. http://www.borland.com/appserver/papers/ejb/ejb.doc [2001, May]
[25] AU-System, May 2001, *We build your Element Manager*.
[26] Java World. Dec 1999.*RMI over IIOP*. http://www.javaworld.com/javaworld/jw-12-1999/jw-12-iiop.html [2001,May]
[27] KTH, Spring 1999, *Course compendium 2G1118: Java Network Programming, IT/KTH.*
[28] Elsmasri, Navathe. 2000. *Fundamentals of Database Systems*. 3rd edition. ISBN 0-201-54263-3. United States of America. Addison Wesley.
[29] Boman, Bubenko Jr, Johannesson, Wangler. 1997. *Conceptual Modelling.* ISBN 0-13-514879-0. Great Britain. Prentice Hall.

Other useful resources, used in the Master Thesis

Internet

- Borland. *JBuilder 4 Team Development Feature Matrix* URL:http://www.borland.com/jbuilder/jb4/feamatrix/teamdev.html [2001, March]
- Inprise. *VisiBroker for Java 4.5 Reference* URL:http://www.inprise.com/techpubs/books/vbj/vbj45/java-reference/contents.html [2001, spring]
- Object Management Group. 1999. *IDL/Java Language Mapping*. Revision 2.3.1. URL: ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf [2001, spring]
- Sun Microsystem. *Java™ 2 Platform, Standard Edition, v 1.3 API Specification* URL: http://java.sun.com/j2se/1.3/docs/api/ [2001, spring]
- KTH, Master Thesis, *Corba vs. DCOM* http://hem.passagen.se/freddas/ [2001,spring]