Université d'Ottawa · University of Ottawa

# CORBA Switching

# A Network Approach

By

Hadi Nasrallah

A thesis submitted to the

School of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Masters of Applied Science

School of Information Technology and Engineering

Faculty of Engineering

University of Ottawa

May 2001

# Table Of Contents

# Abstract

This thesis introduces the concept of CORBA switching. CORBA switching is a network approach to increase the reliability of the current CORBA standard, CORBA 2.3. The CORBA Switch implements the concepts of fault-tolerance, load balancing, versioning, network security and quality of service into a distributed CORBA application. By introducing a CORBA switch into the network, these concepts can be introduced without any modification to the clients or servers. A prototype CORBA Switch was designed and implemented. A series of tests were conducted to evaluate the impact of a CORBA Switch on a distributed CORBA application. The results obtained showed that the CORBA Switch increased the reliability of a distributed CORBA application while adding a very negligible amount of delay into the system. Unlike previous approaches, this thesis shows that the CORBA Switch can deal with all of the following concepts: fault-tolerance, load balancing, versioning, network security and quality of service, while not requiring any modification to existing CORBA clients and server.

# List of Figures

## List of Tables

# Glossary of Terms

| | |
|---|---|
| OODB | Object-oriented database |
| CORBA | OMG standard for distributed computing applications. |
| DII | Dynamic Invocation Interface. Runtime operation requests used by clients. |
| DSI | Dynamic Skeleton Interface for server implementation. |
| GIOP | General Inter-ORB Protocol |
| IDL | Interface Definition Language defined by the OMG for CORBA. |
| IFR | Interface Repository that registers IDL specifications. |
| IIOP | Internet Inter-ORB Protocol |
| IOR | Interoperable Object Reference for objects in separate CORBA processes. |
| OMA | Object Management Architecture |
| OMG | Object Management Group. |
| ORB | Object Request Broker, the CORBA process that provides communications between client and server processes. |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| UML | Unified Modeling Language. |

# 1 Introduction

## 1.1 Motivation and Research Objectives

As the world enters the 21st Century, the Information Revolution is taking place. At the center of this revolution is the Internet. This huge network that spans the globe allows users connected to it to access vast amounts of information in a matter of seconds. Not only does the Internet give a wealth of information to consumers but it also allows corporations to conduct their business in a more effective and productive fashion. There is no doubt that the Internet has become a vital part of how business is conducted today. But behind its ease of use exists very large and very complex distributed systems built upon heterogeneous technologies. Two major problems exist in developing software for such a complex system.

First, in this Internet world, business is no longer run from 9 to 5 o'clock but 24 hours a day, 7 days a week. In a 24x7 world, any loss of service will result in a loss of business. Higher reliability is therefore a must for any business trying to be competitive.

Second, the process of developing distributed applications to run on these complex heterogeneous networks is very slow and complex.

To increase productivity, major software companies joined together to form a consortium to set standards for software development for distributed applications in heterogeneous networks. This consortium, the Object Management Group (OMG), produced in 1991 a standard for distributed computing called CORBA (the Common Object Request Broker) [1]. The aim of this standard is to remove some of the complexity of software development for heterogeneous networks. CORBA allows for seamless communication

between software elements implemented on different operating systems and written in different programming languages. With the development of the Internet, CORBA has found its niche allowing clients and servers implemented on different hardware and software systems to communicate seamlessly across the world. The complexity of dealing with the network is delegated to a software component called Object Request Broker (ORB) [1]. The ORB is present at each client and each server, simplifying the software development stage by freeing the programmer from dealing with the communication process.

The early CORBA standards did not deal with the problem of reliability. The adoption of CORBA may have been delayed by corporations due to the lack of such facilities. These facilities are very important in keeping a distributed system up and running at all time. In a current distributed CORBA application, loss of service may be caused by the following circumstances:

- A fault, which causes a server or servers to crash or to become unavailable. These faults may be caused by a software bug, hardware bug, loss of power, or a loss of communication. When the server or servers become unavailable due to a fault, they are therefore not accessible by the clients who then experience a loss of service.
- A server upgrade. When a server must be upgraded, this may cause a loss of service. The upgrade may be a software upgrade where the operating system is upgraded or the CORBA server itself is upgraded with a newer version. The upgrade may also be a hardware upgrade, where the server may need more

memory (RAM) or a faster central processing unit (CPU). Any one of these server upgrades will force the server to be temporarily shut down, therefore causing all users to experience a loss of service.

- A surge in demand. During peak hours, the demand on the servers may be too large and therefore cause the server to be overwhelmed. Response time of the overwhelmed server may be very long, or the server may simply crash. This will cause a loss of service for all clients.
- A hacker attack such as a denial-of-service attack. In Internet world, most of the servers are vulnerable to hacker attacks. Hackers may cause the server to crash in many different ways including denial-of-service attacks where the hackers flood the servers with requests, causing them to be overwhelmed.

CORBA switching alleviates these concerns by increasing the reliability of a distributed CORBA application by implementing the following concepts: fault-tolerance, versioning, load balancing, quality of service and network security. Some of these concepts are currently available for CORBA but fall short when compared to CORBA switching. A CORBA switch works at layer 7 of the OSI model since CORBA is a layer 7 protocol. The CORBA switch also includes a component that works at layer 2 or 3 of the OSI model, which is responsible for forwarding the CORBA requests on to the layer 7 component of the CORBA switch. It has the advantage of introducing the previously stated concepts to CORBA without any modifications to the client or server applications. The CORBA Switch can be installed in an environment where a distributed CORBA application already exists and increasing reliability while the clients and servers are still

running. The main advantage of the CORBA switch is that it is non-intrusive. It can be installed and uninstalled without affecting the distributed CORBA application. By using the innovative concept of passing all the traffic through a CORBA switch, the server administrator is able to implement these different essential policies:

- Fault-tolerance: If a CORBA server crashes, one of the backup servers will take its load.
- Versioning: The administrator can redirect all requests to the new version of a server without any loss of service.
- Load Balancing: The CORBA switch can distribute all the requests among several servers.
- Quality of service: During peak time, the CORBA switch can selectively drop low priority requests, if its internal request queue is full, so as not to overload the servers.
- Network security: The CORBA switch can detect hacker attacks including denial-of-service attacks and filter them out by monitoring the flow of CORBA requests for flooding.

The main goal of CORBA switching is to introduce the above stated concepts to CORBA with the following restrictions:

- The clients and servers must not be recoded or recompiled. Already deployed application can immediately make use of CORBA switching without any lengthy upgrade cycle.
- There must be no single point of failure in the system

- The CORBA switch must be completely transparent to the clients and servers.

CORBA switching is not suited for every situation. The limitations on CORBA switching are as follows:

- CORBA switching can only version complete servers and not individual objects. When a CORBA object residing in a process or CORBA server needs to be versioned to another CORBA object, the complete CORBA server must be versioned to another CORBA server.
- CORBA switching does not allow for inter-server communication. There can be no inter-server communication because that traffic would not physically pass through the CORBA switch and therefore cannot be switched.
- Ownership of the network is required so that the physical introduction of the CORBA switch can be made.

This thesis presents CORBA switching that increases the reliability of distributed CORBA applications. A framework is presented for implementing the CORBA switch. An implementation of a simplified framework is constructed to demonstrate CORBA switching. Results are presented on the impact to the performance of a distributed CORBA application due to the introduction of a CORBA switch.

## 1.2 Organization of the Thesis and Contributions

This thesis is organized to provide an incremental understanding of the problem of high reliability CORBA in a distributed environment. A design for the CORBA switch is proposed, and implementation results are presented.

Chapter 2 describes the need for and the problems of CORBA switching. Recent work in fault-tolerance, versioning and load balancing for CORBA are reviewed. The chapter also includes a review of CORBA mechanism needed for CORBA switching

Chapter 3 describes the new concept of CORBA switching.

Chapter 4 describes a framework for adding quality of service and network security extensions to the CORBA switch

Chapter 5 describes an implementation of a prototype CORBA switch.

In Chapter 6 a series of experiments are run to accomplish two goals:

- Prove the concept of CORBA switching
- Measure the performance penalty of using a CORBA switch.

Finally, in Chapter 7 conclusions are presented on the costs and benefits of the CORBA switch. The possible extensions of the CORBA switching concepts to other protocols are also explored.

This thesis contains several research contributions, which can be summarized as follows:

1. An innovative concept of CORBA switching is presented for the first time. A new methodology for CORBA fault tolerance, versioning and load balancing is introduced. A new approach to CORBA quality of service and network security is described.

2. A fully functional prototype of a CORBA switch is constructed. A test application is produced and tested. The impact of CORBA switching on the performance of a CORBA distributed application is explored.

3. Recommendations are made for future research in CORBA switching.

# 2 CORBA Switching for CORBA Applications

Several different middleware technologies exist today such as Java RMI, CORBA, SOAP, DCOM and so on. Middleware environments simplify the process of developing applications for a distributed environment and increase interoperability between different components. One such object-oriented middleware is CORBA. CORBA is becoming increasingly popular due to the fact that it is standard based and technology independent (Operating system, hardware, programming language, network). Most of the existing middleware, including CORBA, use point-to-point communication between clients and servers. Deficiencies of the point-point communication model become evident in the context of fault tolerance, versioning and load balancing. The aim of the thesis is to study and propose a new approach for converting CORBA's point-to-point communication into a group communication without any modification to existing CORBA standards. Through the introduction of a CORBA switch, CORBA's point-to-point communication becomes a hybrid group/point-to-point communication allowing the implementations of following concepts: fault-tolerance, versioning and load balancing.

This chapter describes the problems of high reliability in distributed systems especially CORBA-based systems and the need for CORBA switching. Related work in the fields of fault-tolerance, versioning and load balancing in a distributed CORBA environment are discussed. The requirements for a high-reliability CORBA system through CORBA switching are defined.

## 2.1 Related Work

### 2.1.1 Background

#### 2.1.1.1 The Common Object Request Broker Architecture

In 1989, a consortium of software and infrastructure companies founded the Object Management Group (OMG) to promote the adoption of standards for software development in a distributed environment. Today the consortium counts more than 800 members. The first CORBA standard, CORBA 1.1, was introduced in 1991. In July 1998, the current CORBA standard, CORBA 2.3 was ratified and currently the specifications for CORBA 3 are being finalized. A CORBA is an open, vendor-independent middleware architecture and infrastructure that allows applications to easily communicate amongst each other without the programmer dealing with low-level network functions. Using the standard protocol IIOP [6] (Internet Inter-ORB protocol) developed by OMG, a CORBA-based program from any vendor, on almost any hardware using any operating system implemented in any programming language, and using any type of network, can communicate with a CORBA-based program from the same or another vendor, on almost any other hardware using any operating system implemented in any programming language, and using any network anywhere in the world.

#### 2.1.1.2 The Object Management Architecture

The Object Management Architecture (OMA) provides a definition of object concepts but leaves the implementation to the different vendors. This allows for the architecture to

be implemented in different manners but ensures interoperability. The OMA defines five components, of which the following three are relevant to this thesis:

- *Object Request Broker:* The object request broker (ORB) is core of the architecture. It allows requests from the four components to be exchanged using standardized interfaces. CORBA uses the interface definition language (IDL) to define interfaces for its objects.
- *Application Interfaces:* These interfaces are specific to each application and are developed during system construction.
- *Object Services:* Also known as CORBA Services, which extends the ORB's support for applications.

#### 2.1.1.3 Smart Proxies

Using CORBA as the communication environment and mechanism, a client object and a server object do not communicate directly. On the client side, a *proxy object* (or just *proxy*) is instantiated in the client's address space. The proxy provides the client an interface to the server object. When the client invokes an operation on the proxy, the proxy then transmits the requests to the remote *skeleton*. The request is transmitted over the network using IIOP [1][6]. The job of the transmission of CORBA requests and reception of the CORBA replies over the network is therefore delegated from the client object to the proxy. This frees the client object from network specific function. The skeleton thereafter forwards the call to the server object. This process is shown in Figure 2.2.

The proxy object is implemented in the same programming language as used to code the ORB and is typically automatically generated by tools provided with the ORB by the vendor. The proxy instance provides an interface to a specific type of object. This means that the client must be aware of the server IDL interface at compile time. Other mechanism exists for situations where the server IDL interface is not known at compile time.

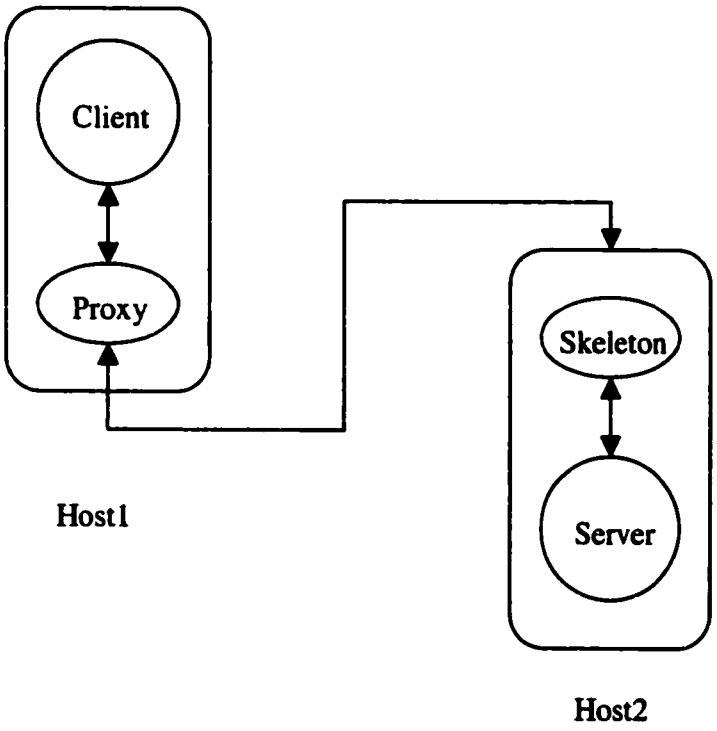


**Figure 2-1 CORBA communication using proxies and skeletons**

Some vendors allow the proxy classes to be manually implemented. The programmer is able to augment the functionality of the tool-generated proxy. The manually implemented proxy inherits from the tool-generated proxy and is called *smart proxy*. This approach can be used to implement more efficient proxies that can do caching and load balancing.

#### 2.1.1.4 General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP)

OMG defined GIOP[1][6] as the mechanism through which ORBs may communicate amongst each other. GIOP is intended to run over any connection-oriented transport protocol. The most commonly used implementation of GIOP is IIOP. IIOP is a specialization of GIOP designed to work over TCP/IP[7][8]. Most current implementations of CORBA use IIOP because the majority of networks today are IP based. ORBs from different vendor can easily communicate amongst each other as long as they are fully compliant with IIOP or any subset of GIOP.

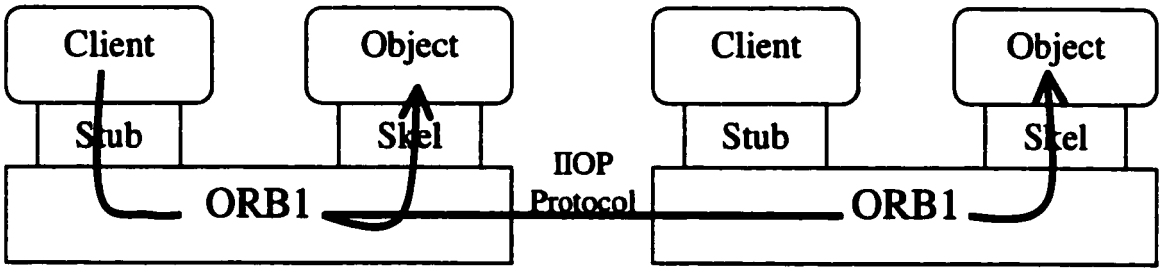


**Figure 2-2 IIOP for inter-ORB communication**

The GIOP standard defines 8 messages to be used by clients and servers, as shown in Figure 2.4. The Request and Reply messages are straightforward. A client sends a request to a server, which in turn sends a reply. A client may send a CancelRequest messages if the reply from the server is no longer needed. A LocateRequest message is sent by the client if more information about the server is needed such as:

- Is an object reference valid?

- Can the current server receive the request from an object reference?
- What is the address to which to send an object request?

The server would then respond with a LocateReply message. A CloseConnection message informs the client not to expect any further messages from the server. Both the client and server can send MessageError and Fragment. MessageError is sent when an error condition is detected as a result of a message and Fragment is sent when a Request or Reply message has to be fragmented.

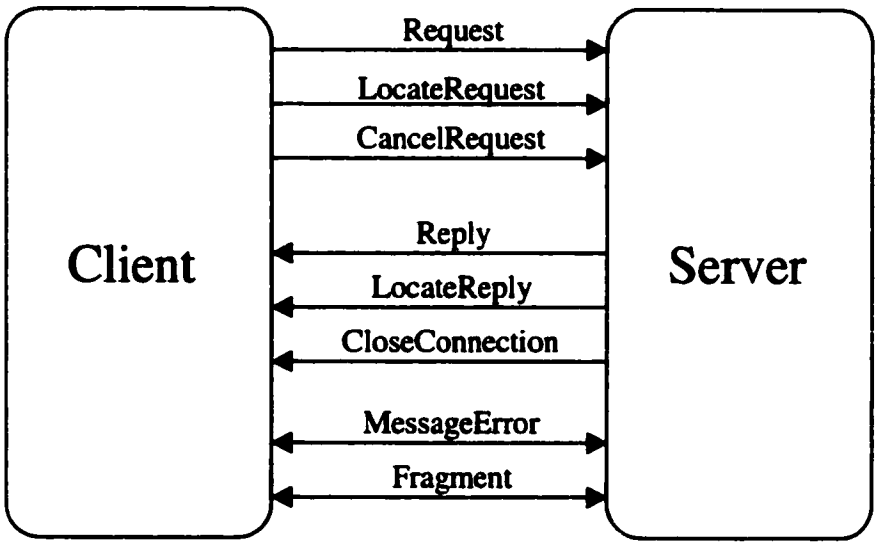


**Figure 2-3 GIOP messaging**

### 2.1.1.5 Interoperable Object References (IOR)

When CORBA objects reside on different hosts in a network, there must exist a way a CORBA object can uniquely identify itself to the rest of the world. The representation of a CORBA object is done using an Interoperable Object Reference (IOR).

In the IOR, the address of the host is found in the variable *host*. This address may be the IP address or the DNS name[21] of the host. The TCP port number of the CORBA server is also included in the IOR in the variable *port*. The *object_key* uniquely identifies the CORBA object within a CORBA server. There can be many CORBA objects running on the same CORBA server, the *object_key* allows the IOR to uniquely identify the different CORBA objects. The IIOP's IOR definition is shown in Figure 2-5.

```
Module IIOP {                       //IDL extended for version 1.1
        Struct Version {
                Octet               major;
                Octet               minor;
        };
        structProfileBody_1_ {      //Version 1 of IIOP
                Version             iiop_version;
                String              host;
                Unsigned short      port;
                Sequence <octet>    object_key;
        };
        structProfileBody_1_1 {     //Version 1.1 of IIOP
                Version             iiop_version;
                String              host;
                Unsigned short      port;
                Sequence <octet>    object_key;

                Sequence >IOP::TaggedComponent> components;
        };
```

**Figure 2-4 IIOP IOR Structure**

#### 2.1.1.6 Naming Service

In a distributed environment such as CORBA, clients must be able to retrieve object references to remote servers. In some instances, the object references may already be located on the client side, e.g. looked up in a table or saved in a file. But in most cases the client will have to retrieve the object references from a remote repository like the OMG CORBA Naming Service [9]. In the CORBA Naming Service, object references are mapped to a name provided by the user. The name provided by the server consists of two strings, an Id and a Kind. For a client to retrieve the object reference from the CORBA Naming Service, the client must be aware of the name used by the server to map the object reference in the CORBA Naming Service.

#### 2.1.1.7 Trading Service

The limitation of the CORBA Naming Service is the fact that the client must be aware of the name that the server used to bind the object reference in the CORBA Naming Service. The Trading Service [10] was designed to overcome this limitation by allowing clients to find object references based upon the type of service. Clients can request references to all objects that match a certain criteria. The Trading Service would then perform a search and return all references matching the given criteria. It is then up to the client to narrow the search or select one of the returned object references.

### 2.1.2 Group communication toolkits

Group communication involves gathering together a set of processes or objects into a logical group, and providing an interface through which a user may send messages to all members of the group at the same time. Group communication also provides different guarantees on the order by which the messages will be received by the servers. Messages are addressed to the group using a logical addressing facility. Clients need not be aware of the location and number of servers belonging to a group. These group communication toolkits provide higher-reliability through replication of the servers.

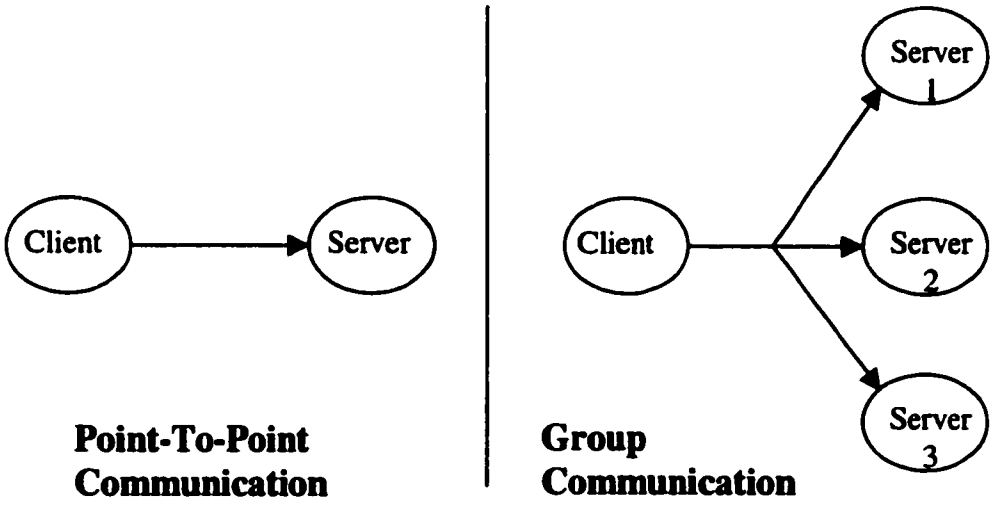


**Figure 2-5 Point-to-Point and Group Communication**

Figure 2.6 shows how the client makes a single remote call through the interface of the group communication toolkit, which in turn distributes the message to all servers

belonging to a group. Instead of addressing every server in the group, the client need only address a logical group.

Many of these group communications toolkits exist today providing support for managing groups in distributed environment. One of the toolkits relevant to this thesis is Isis [11]. The Isis is one of the first group communication toolkits and was developed by Cornell University. Other toolkits available today include Phoenix[12], Horus[13], Ensemble/Maestro[14][15], Totem[16] and Transis [17]. Some research done in the area of fault tolerance CORBA have made use of group communication toolkits.

### 2.1.3 Fault-Tolerance through Replication

Fault-tolerance can be achieved through replication of the servers. Two different replication techniques exist depending on the intended application: *active replication* and *passive replication* [26][27].

Active replication [26], as shown in Figure 2.7, involves the client sending the request to all replicated servers. Each one of the servers will process the request, update their state and then send a reply back to the client. At the client, a decision will be made on which reply to use. This can be done in several different ways:

- First come, first serve: The first reply to be received will be used by the client
- Voting: All replies received will be compared. If one of the messages differs then a fault may have occurred at that server, and the message is ignored.

To ensure that the states of all the servers are the same, messages received by the servers must be in the same order. This can be guaranteed by a group communication toolkit.

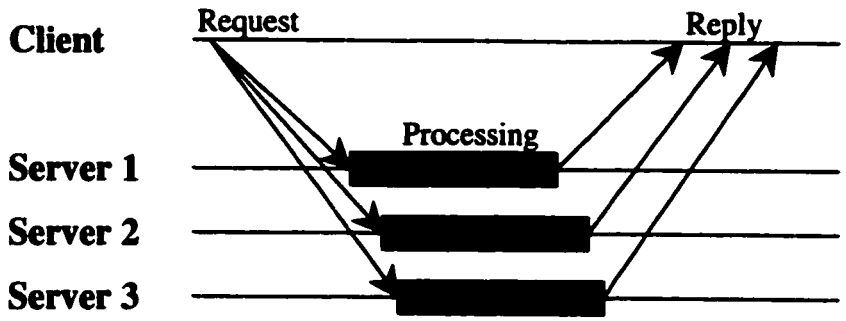


**Figure 2-6 Active Replication**

Passive Replication [26], also known as primary-backup replication, involves the client communicating with one primary or master server at first. In case of a fault at the primary or master server, the client would then switch over to one of the backup servers. To keep the state of all the servers the same, a mechanism for updating the backup servers is needed and is shown in Figure 2.8. The state of the backup servers is updated after each state modifying requests made on the master server. The primary server is responsible for propagating the state change to the backup servers. This mechanism is not needed in the case of stateless servers.

Unlike the active replication technique, the passive replication does not waste resources through redundant processing but does involve some overhead in the transfer of states between the master and backup servers. There will be an additional delay in the case of a failure, which is not present in the active replication technique. The added delay involves the detection of the failure and the retransmission of the requests to one of the backup servers.

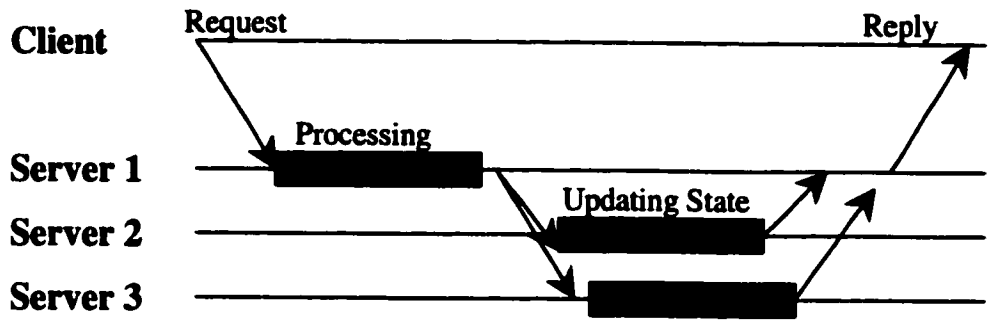


**Figure 2-7 Passive Replication**

## 2.2 The Problem of a high-reliability CORBA system through CORBA switching

In a typical CORBA-based distributed environment multiple client processes invoke operations on remote objects residing on remote servers. The clients retrieve and hold CORBA references to these remote objects. These references are kept in the client application and used when an invocation is to be made. If one of the references held by a client application becomes invalid, the client will generate an exception and will become unworkable. The CORBA references mentioned above, become invalid in the following situations:

- Fault: A fault occurs when the server crashes and the objects residing in that server no longer reside in the memory.
- Versioning: The server is versioned to a different version. This involves shutting down the old server and launching the new server. The references now point to the old objects in the old server who may or may not still be running.

When a server crashes, all references to CORBA objects residing the server become invalid, and clients with these references generate errors and become unworkable. For example in Figure 2-8, the reference held by the client to CORBA Object "A1" and "A2" become invalid when the server crashes. Any invocations made by the client to these two CORBA objects will generate errors in the client application.

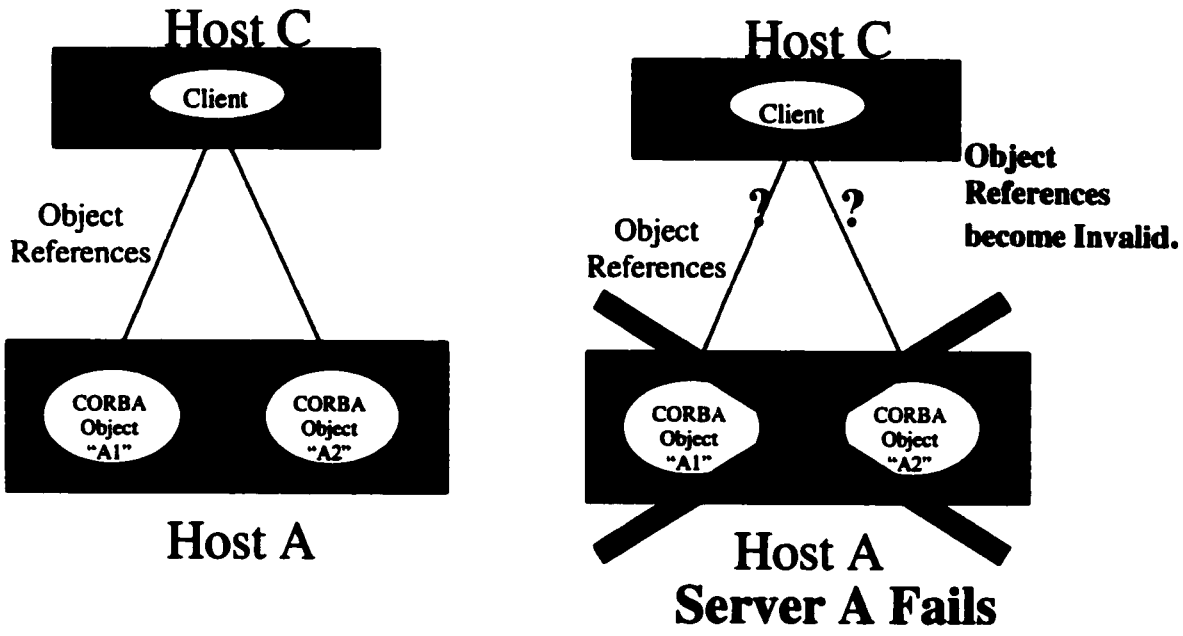


**Figure 2-8 Object References in a Distribution Application in case of a server crash.**

When a server is versioned as in Figure 2-9, the client still holds references to the old Server A. If the server is still running then the client will still be using the old server, Server A, and therefore return potentially outdated results. If the administrator were to shutdown the old server, Server A, then all clients still holding references to Server A will generate errors when invoking requests on the server.

Versioning a server entails replacing the currently implementation of that server with a newer one. The new version may be on a different host, programmed in a different

programming language, return different results for the same inputs but must provide the same interface as the older version. In the case of CORBA, the new server will have the same IDL interface as the old server.

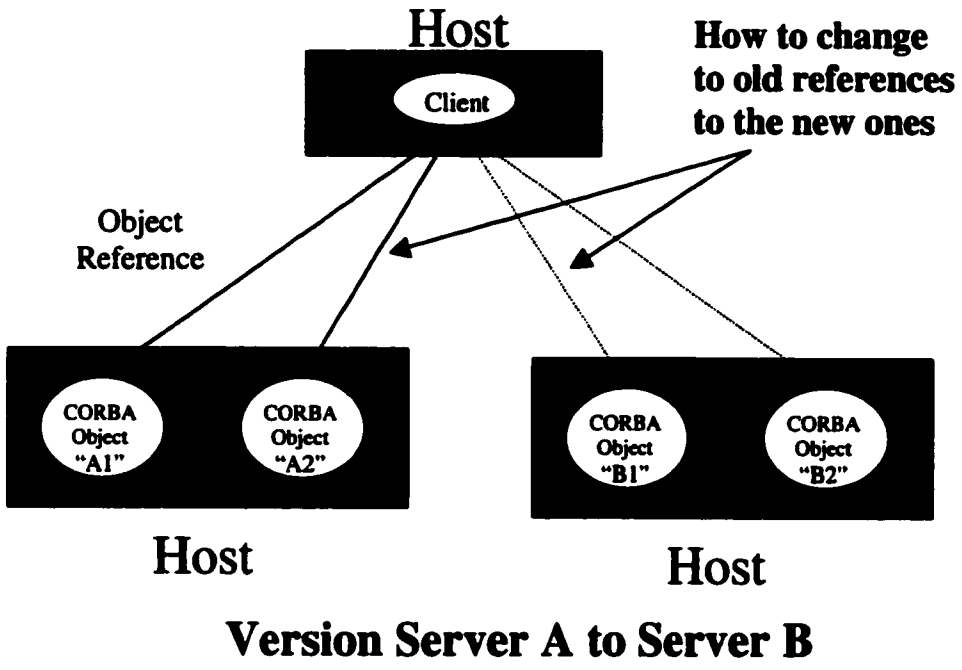


**Figure 2-9 Object References in a Distribution Application in case of a versioned server.**

Another cause of loss of service occurs when a server becomes overloaded. This can occur for different reasons:

- During peak demand, the load on the servers can be too large to handle. This demand can cause very large delays at the server or servers, or can cause the server or servers to simply crash.
- Hacker attacks sometimes involve flooding servers with "dummy" requests, overloading the server or servers. The overloaded server or servers will not be able to respond to legitimate requests in a timely fashion.

## 2.3 Introduction to CORBA Switching

CORBA switching increases the reliability of distributed application by introducing the concepts of fault-tolerance, load balancing, versioning, quality of service and network security. The principals of CORBA switching are as follows: First, the CORBA switch must be physically placed between the CORBA clients and the CORBA servers. This forces all the messages between the clients and servers to pass through the CORBA switch. When a client initiates a TCP connection with a CORBA server, the packets are intercepted by the CORBA switch, which accepts the TCP connection on behalf of server. The client then generates CORBA requests, which are accepted by the CORBA switch, which then forwards them to the appropriate server or servers depending on the setup of the CORBA switch.

## 2.4 Related Works

In recent years a lot of research has been conducted to increase the reliability of CORBA. All works can be grouped in three categories according to their approach:

1. The *integration approach* involves modifying the ORB itself and to integrate a group communication toolkit into it [26][27].
2. The *interception approach* involves interception the CORBA calls and redirecting them to a group communication toolkit [26][27].
3. The *service approach* involves augmenting CORBA by adding new CORBA services or modifying existing ones [26][27].

In this thesis, a forth approach is proposed, a *soft network approach*, where all the intelligence is placed in a network element, hidden from the user.

### 2.4.1 Integration Approach

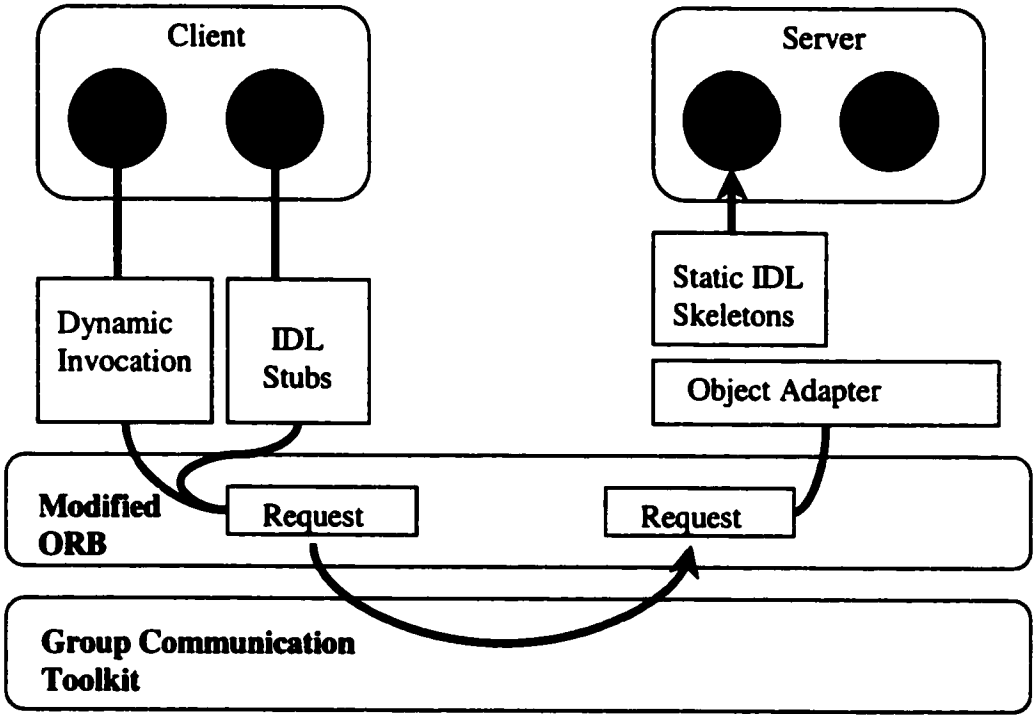


**Figure 2-10 Integration Approach**

The integration approach involves modifying the ORB itself to include fault-tolerance directly into it. The approach taken is for the ORB to intercept requests on object groups and to pass the request on to a group communication toolkit. The toolkit will, in turn, multicast the request from the client to all the replicated servers using proprietary protocols and mechanisms. This is the approach taken by Electra[18][19] and Orbix+Isis [19] as well as by OMG with their specifications [20] that is was ratified February 2001. The integration approach is shown in Figure 2-11.

#### 2.4.1.1 Orbix + Isis

Orbix[3][4] is a commercially available CORBA implementation from Iona Technologies. Isis [11] is a group communication toolkit available from Isis Distributed Systems. By collaborating, Orbix+Isis is able to map replicated CORBA object to Isis groups and provide group communication extensions to Orbix. In Orbix+Isis, point-to-point communication is handled by Orbix, while multicast are delegated to the Isis toolkit.

In this approach, servers must inherit from an abstract class that will therefore determine the role of the server at compile time. The programmer has two options to choose from: *Active Replica* and *Event Stream.*

The active replica execution style provides three different options for the clients to choose form. These three options control how requests are handled by Isis:

- The *multicast* option tells Isis to multicast the request to all objects in the group.
- The *client's choice* option allows the user to invoke a request on a specific server in the group. The target server is chosen by the user. This option is usually used for read-only methods. These methods do not affect the state of the servers.
- The *coordinator/cohort* option resembles the primary-backup approach. Requests are only sent to one server, the master server, and are only sent to the backups if the master fails.

The event stream execution style supports an asynchronous approach to message passing. This style is used for requests not requiring a reply. The clients register to the event stream as senders of messages, and the servers as receivers. When a client invokes a

request, it is sent as an event to all the servers. The clients and servers are therefore decoupled from each other. This execution style is shown in Figure 2-12

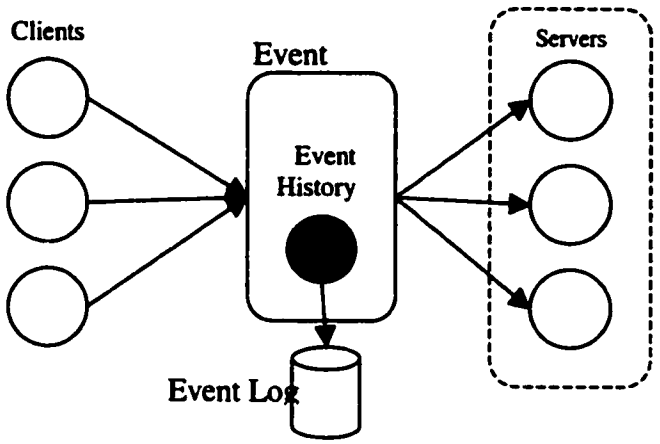


**Figure 2-11 Orbix+Isis Event Stream Execution Style**

#### 2.4.1.2 Electra

The approach taken by the authors of Electra is to extend the *Basic Object Adapter* (BOA)[1], which is the base class of most CORBA objects in CORBA 2.2, adding the ability to *create, destroy, join* and *leave* a group. Since the modifications are done at the BOA level, every CORBA object can potentially become a member of a group. As with Orbix+Isis, the requests are then forwarded to a group communication toolkit. Unlike Orbix+Isis, which exclusively uses Isis, Electra allows for the reimplementation of its adaptation layer allowing Electra to work with different group communication toolkits like Isis or Horus.

One of the main advantages of the integration approaches, like Orbix+Isis and Electra, is the transparency to the client. The clients cannot distinguish between a group and a singleton object. But this transparency comes at a cost. The ORB itself has to be modified. This involves recompiling and reimplementing all clients and servers in the distributed system. Any old client or server still running will not be able to take advantage of such a system.

Another disadvantage that comes with this system is portability. Any client or server implemented in Orbix+Isis or Electra cannot communicate with other CORBA clients or servers because they are very ORB-dependent. This is a huge disadvantage because CORBA is built upon the principle of multi-vendor solutions.

Orbix+Isis and Electra do provide very high efficiency due to the integration of group communication toolkits directly into the ORB but this also has the added disadvantage of relying on a third-party tool and not completely open to the software developer.

Electra extends the *Basic Object Adapter* (BOA)[1], which is the base class of most CORBA object in CORBA 2.2 but this class has been replaced with the *Portable Object Adapter* (POA) [1] in later versions of CORBA. It is unclear if Electra can be adapted to this new class.

Compared to the CORBA switch, these two approaches do not address the problems of load balancing and versioning and cannot address the problem of quality of service and Network Security of the servers.

#### 2.4.1.3 OMG's Fault-tolerant standard

In February 2001, the Object Management Group (OMG) ratified a proposal for fault-tolerant CORBA [20]. OMG set out to modify the CORBA standard to incorporate fault-tolerant properties into the ORB. The main objectives of the standard are to make CORBA more reliable through a range of fault tolerance strategies, including active and passive backup, providing as much transparency to the clients as possible and provide no single point of failure. The standard also provides support for fault detection, notification and analysis for the replicated objects.

Fault-tolerance is achieved through replication of server objects, which will be grouped into *object groups*. All objects in a group still have their own object references (IOR), but the new standard introduces a reference to the whole group termed *interoperable object group reference* (IGOR). This reference will be published for use by all clients. Clients will therefore hold a single reference to the group making it transparent to clients.
The standard specifies that *object groups* will be grouped together into *fault tolerance domains* to be managed by a single Replication Manager. The Replication Manager is also replicated in case of failure. The Replication Manager is in charge of creating and managing all the servers it is in charge of.

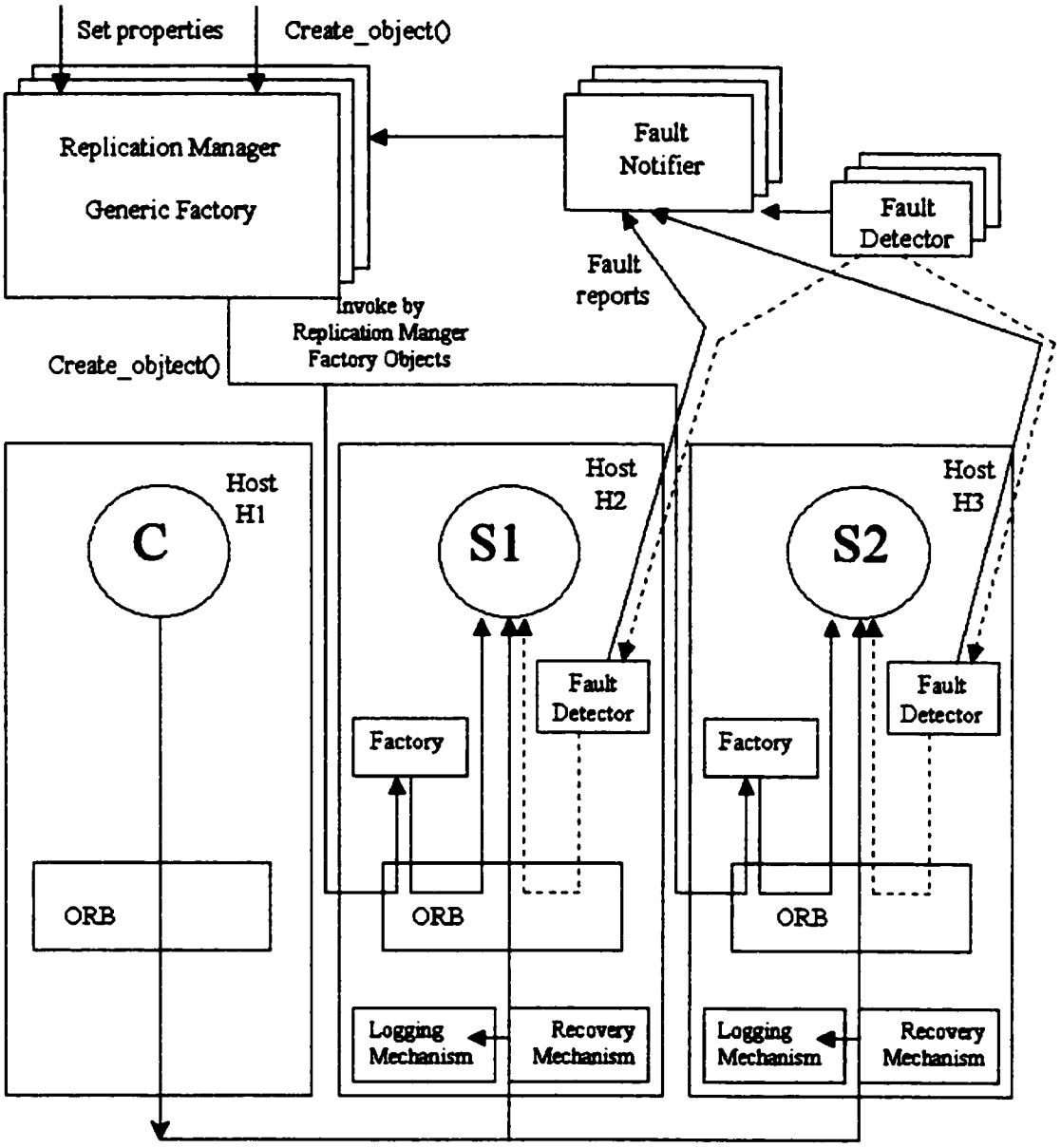


**Figure 2-12 Architecture of OMG's Fault Tolerant CORBA**

### Architectural Overview

Figure 2-12 shows the architecture of a fault-tolerant system. The fault tolerant Infrastructure is composed of the 3 components shown at the top of the Figure: Replication Manager, Fault Notifier, Fault Detector. These three components are

implemented as CORBA objects. Logically there is only one instance of each of these components but physically they are replicated to protect against faults, just as are the servers. The figure also shows a client C on host H1 invoking a request on server S1 on host H2 and on server S2 on host H3. The figure also shows Factory and Fault Detector objects located in the server. These objects are not replicated. The new ORB now includes a Message Handler, and a Logging and Recovery Mechanism.

The standard allows for two types of fault tolerance strategies:

- Active Replication. The Active Replication Style requires that all requests be sent in the same order to all members of an object group. Each object will execute the request, and will therefore maintain the same state. The standard only states that a proprietary group communication system may be used but must remain invisible to the clients and servers. Details are left up to the different vendors.
- Passive Replication. The Passive Replication Style requires the request be sent to the primary server. The state of the primary server and the sequence of requests are logged. In the case of a fault, the Replication Manager attempts to restart the server with the saved state or brings up a backup server with the same state as the failed primary server.

The architecture also describes a mechanism of Fault Detection and Notification. Fault Detectors periodically invoke the *is_alive*() function that all fault tolerant objects inherit. For efficiency, Fault Detectors are located on the same host as the server. There also is a global Fault Detectors that monitors the individual Fault Detectors. The Fault Notifier allows for clients to register and to receive events in case of the failure of a server.

**Interoperable Object Group References**

The standard describes a new type of object reference, which no longer references a single object but a group of objects. Inside this reference will be contained many references to different objects. In the case of Passive Replication, the IOGR will include the IOR of the primary server and the secondary servers. In the case that the primary backup has changed and the reference the client is holding is outdated two situations are possible:

- The old primary has failed and is not currently running: The client will detect a fault and therefore switch over to a backup server.
- The old primary has failed and was launched again as a backup server: The client will invoke the request on the server, which will redirect the client to the appropriate primary server.

The OMG approach suffers from the same disadvantages as Orbix+Isis and Electra. Clients or servers running legacy ORB cannot take advantages of the new fault-tolerant properties of CORBA. This, therefore, involves recoding, reimplementing, and redeploying all the clients and servers in the distributed system.

In a fault tolerant domain, all servers must be implemented using the same ORB. This means that a single CORBA vendor has to be used to implement all the servers in a fault tolerant domain. In the case of the CORBA switch, this is not the case. If for example, a system administrator wanted to make the CORBA Naming Service fault-tolerant using the CORBA switch, the administrator would run a Naming Service implemented by

Orbix, one implemented by Mico[25] (a free C++ ORB) and one from JacOrb [22][23][24] (a free Java ORB). In the case that a software bug exists in any of the three implementations, the system would still continue to operate using the remaining two implementations. With the OMG approach, all three servers must be from the same vendor, i.e. the three CORBA Naming Service servers must be from only one of the three vendors. This does not protect against software bugs in the implementation and limits the choices that a potential user has.

The OMG approach does specify the use of group communication toolkits for multicast communication but all other information is left up to the specific vendor. Different vendors will rely on different toolkits causing the system to be much less open to the user.

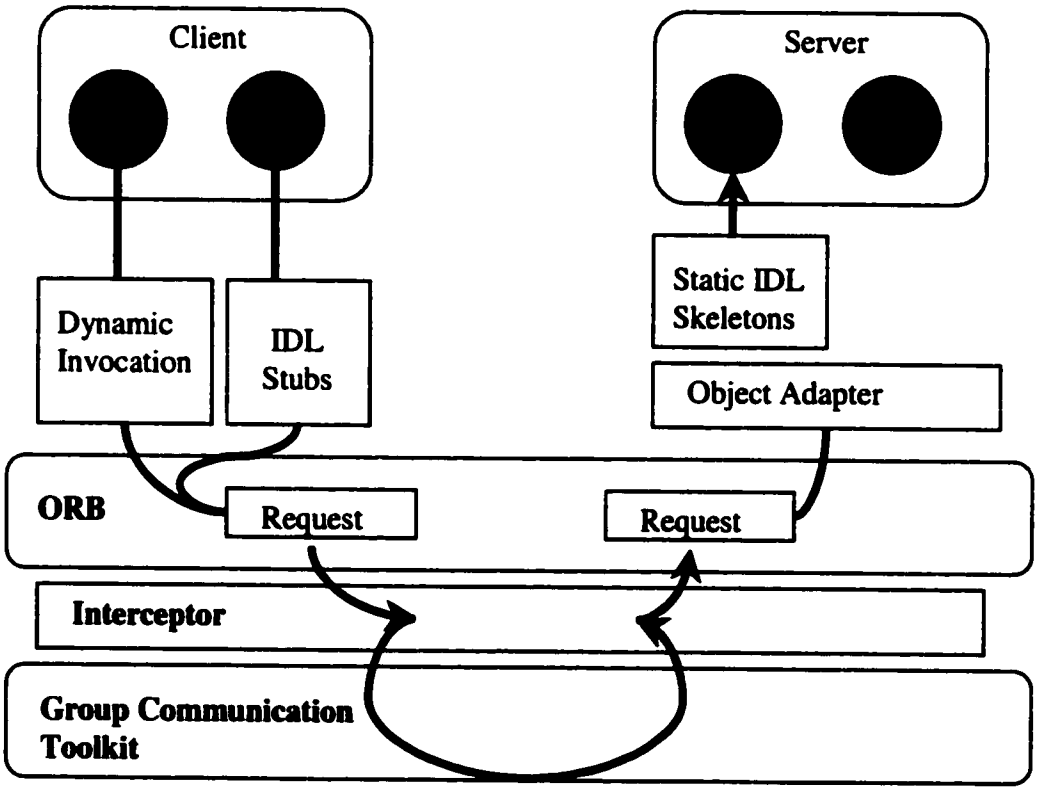


**Figure 2-13 Interception Approach**

### 2.4.2 Interception Approach

The Interception approach involves intercepting all CORBA requests after the ORB but before reaching the OS. The intercepted requests are then forwarded to a group communication toolkit. The interception approach is shown in Figure 2-14.

#### 2.4.2.1 Eternal

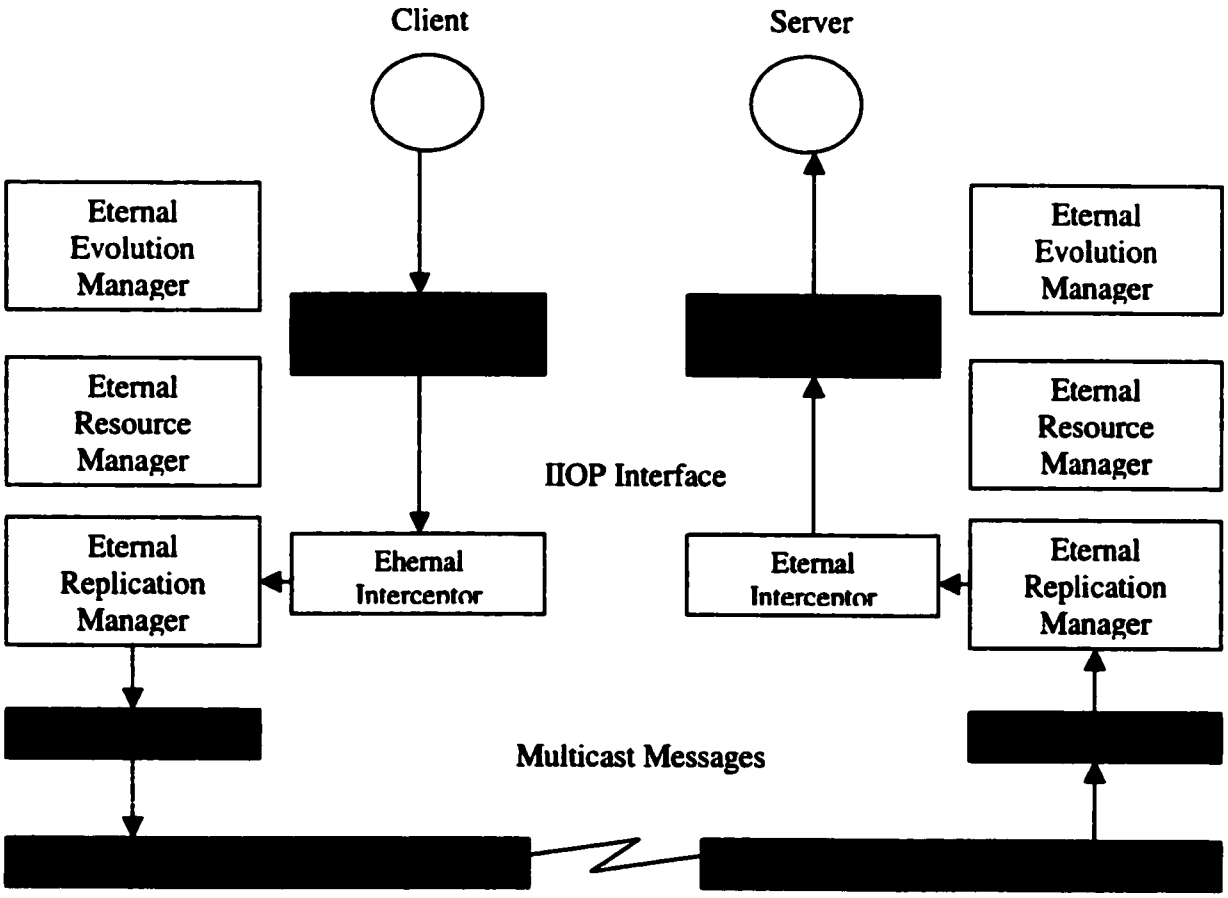


**Figure 2-14 Eternal Architecture**

Eternal implements the interception approach for CORBA. Eternal is responsible for the creation and the maintaining of the replicated objects. The object group abstraction hides the location of the server objects as well as the type of replication used. Eternal was designed to operate on a Unix operating system and with standard CORBA implementations. The Unix operating system was used to take advantage of the Unix */proc* interface to monitor operating system calls made by CORBA objects while establishing IIOP connection over TCP/IP. IIOP messages are intercepted by Eternal and then forwarded to Totem, a group communication toolkit. Totem will in turn multicast the requests to all the servers. According to the authors, Eternal could be implemented on a different operating system, and may be configured to use any other group communication toolkit like Isis.

Figure 2-15 shows the different components of Eternal which include:

-*Interceptor*: The interceptor intercepts standard CORBA requests.

-*Replication Manager*: The Replication Manager handles communication with copies of the replicated objects.

-*Resource Manager*: The Resource Manager manages the system configuration (the replication style as well as the servers used).

-*Evolution Manager*: The Evolution Manager allows for the number of servers used to be changed at runtime.

The disadvantages of the interception approach and Eternal are as follows:

- Eternal is operating system dependent. A different implementation will have to be coded for each operation system.
- Eternal relies on a third party group communication toolkit. Even though the toolkit to be used can be changed by rewriting the adaptation layer, this approach still relies on non-Eternal software. Eternal systems coded for different group communication toolkit will not be able to communicate amongst each other.
- Eternal requires the reimplementation of all the servers in the system and requires the installation of Eternal on all the client machines. If on a certain machine, a client is running without Eternal installed, the client would not be able to take advantage of the fault tolerant capabilities of the server.
- Eternal does not currently deal with the problems of load balancing and versioning but might be modified to do so. On the other hand, Eternal cannot address the problem of quality of service and network security.

### 2.4.3 Service Approach

The Service Approach attempts to improve CORBA reliability by providing the fault-tolerance through a CORBA service. This approach therefore does not modify the ORB and would work with any CORBA implementation. The Service Approach is shown in Figure 2-15.

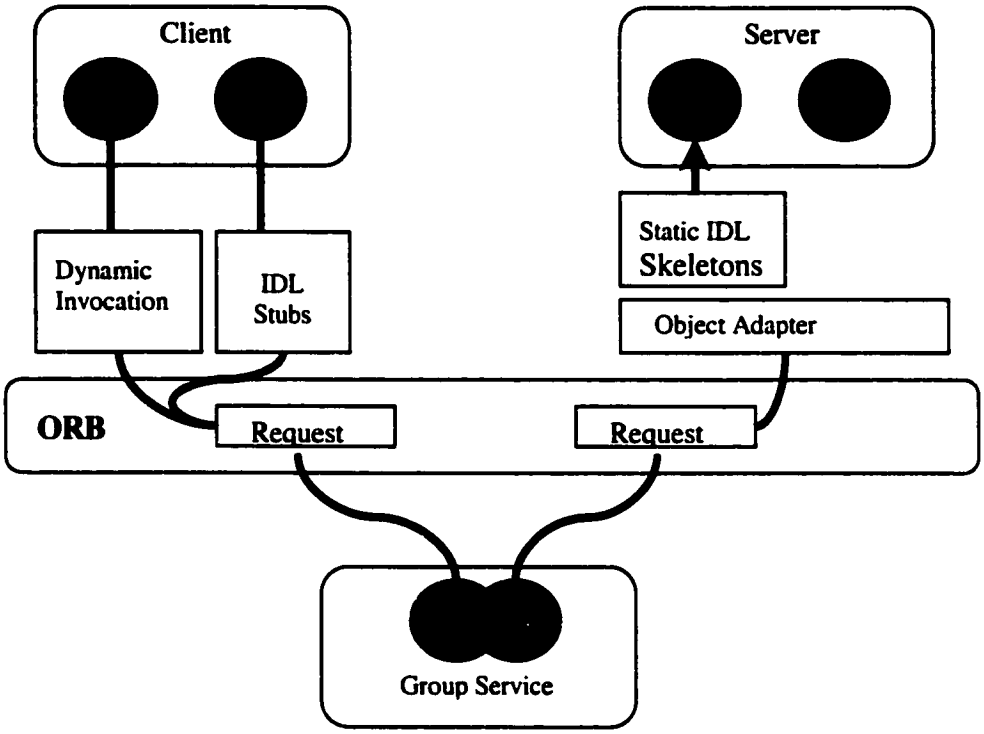


**Figure 2-15 Service Approach**

#### 2.4.3.1 Object Group Services

Object Group Service (OGS) [26][27] defines a Common Object Services (COS) [28] with IDL interfaces which provide the same functionality as a group communication toolkit. This functionality is provided to the clients as a CORBA Service and does not require any modifications to current CORBA standards. Not only does OGS provide fault-tolerance for CORBA but load balancing, object versioning and fault monitoring.

**Architecture of OGS**

OGS is made up of several components independently implemented interacting with each other through the ORB. An overview is shown in Figure 2-17. These components are:

1. A *Message Service* provides non-blocking reliable point-to-point and multicast communication.

2. A *Monitoring Service*: Monitors objects and detects failure.
3. A *Consensus Service* used to implement group multicast and membership protocols.
4. A *Group Service*: The *group service* is the core of the OGS environment. It manages the actual object groups. The *group service* implements the following two functionalities:
    a. *Group Membership*: It maintains the list of group members, allows for addition and removal of group members.
    b. *Group multicast*: allows for multicast and point-to-point communication with all members of a group.

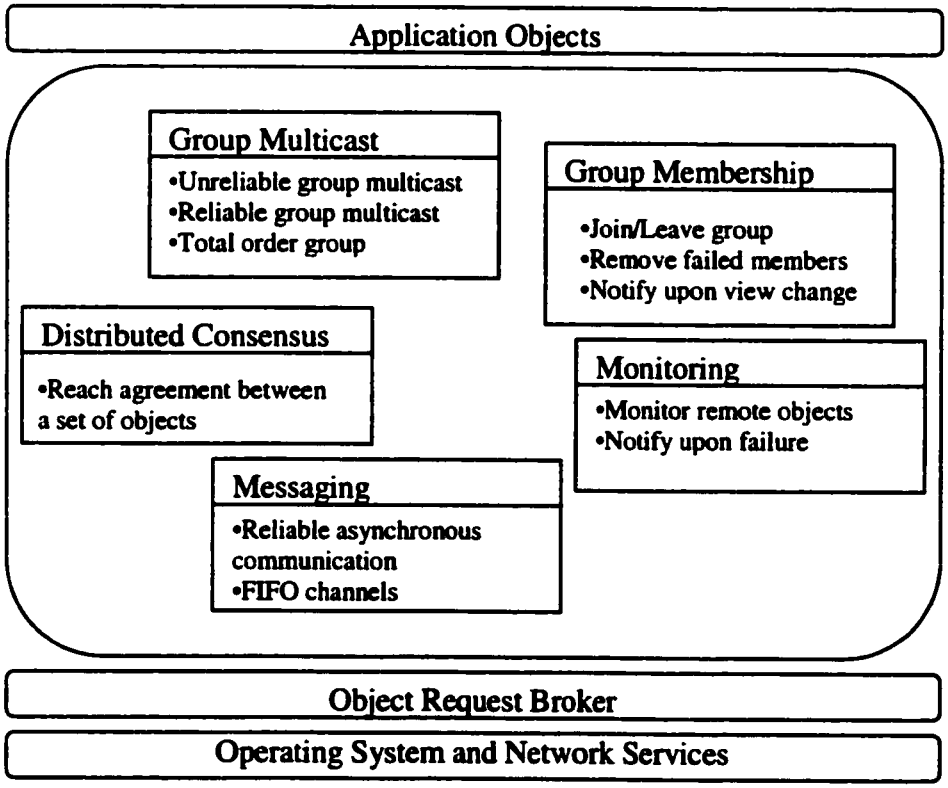


**Figure 2-16 Overview of the OGS Architecture [26]**

**Operational Overview**

OGS uses two advanced CORBA features to communicate with clients and servers: the *Dynamic Skeleton Interface* (DSI)[1] and the *Dynamic Invocation Interface* (DII)[1]. The client invokes the request on the OGS server instead of the server. The OGS server is able to receive the request even though the IDL interface of the OGS is different than the intended server using the DSI feature of CORBA. DSI allows for a server to receive requests that do not match its interface. Then the OGS invokes the request on the appropriate servers using DII. DII allows OGS to invoke a request on any server even though the IDL interface of the servers is not known at compile time.

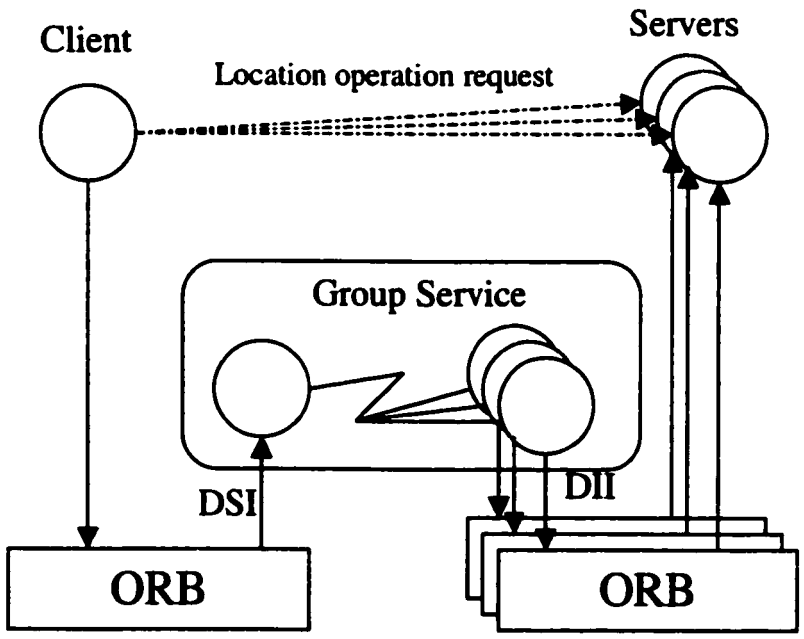


**Figure 2-17 Operational overview of OGS**

### 2.4.3.2 Interoperable Replication Logic

The Interoperable Replication Logic (IRL)[29][30] provides fault tolerance to CORBA as a service hidden from the user. Unlike OGS, the client does not directly interact with the service but indirectly through a Smart Proxy and is implemented as a single server instead of multiple servers. IRL does provide interfaces for managing server replication styles, i.e. active and passive replication. This approach is similar to the approach being considered by OMG, but does not directly modify the ORB. The architecture of IRL is shown in Figure 2-18.

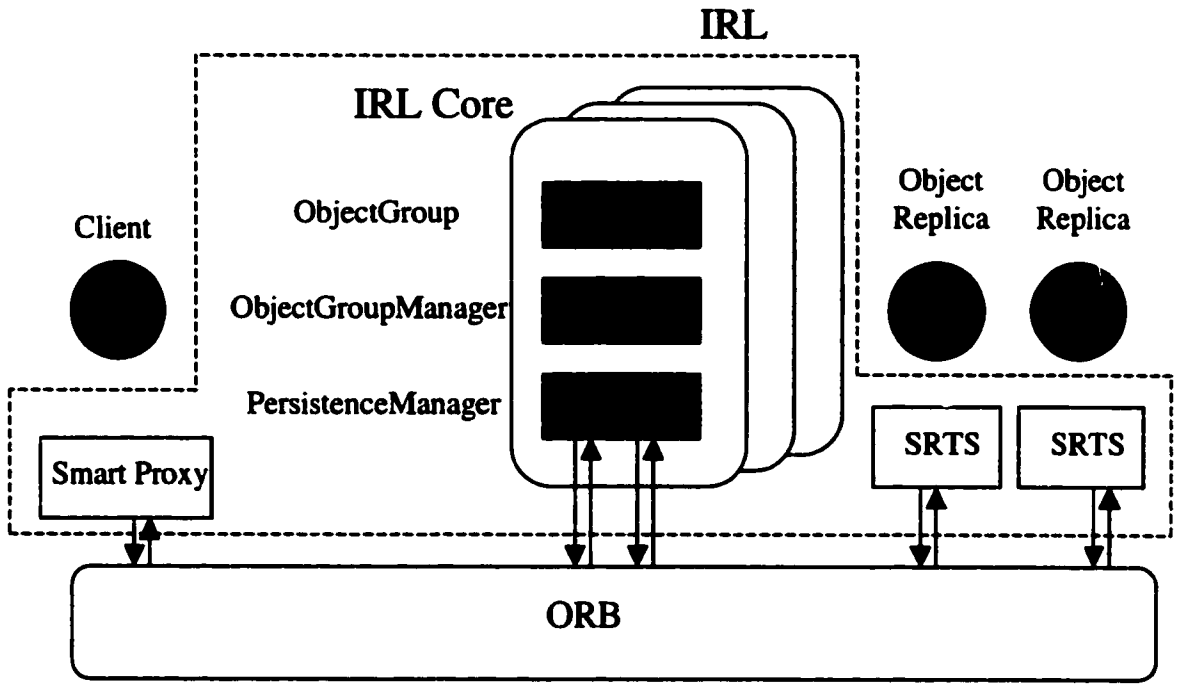


**Figure 2-18 IRL Architecture**

The IRL core includes three components running as a single CORBA process:

- ObjectGroup: An ObjectGroup object is logically associated to a group of replicated objects. It receives all requests directed at the group, then forwards

them to the appropriate servers (depending on whether active or passive replication is used). In the case of passive replication, the ObjectGroup object is responsible for maintaining the consistency of the states of all the replicated objects. A portion of the failure detection mechanism is done by the ObjectGroup who can detect failure of an object replica.

- ObjectGroupManager: The addition and removal of replicated objects to and from groups is done through an IRL management application, which interfaces with the ObjectGroupManager object.

- PresistenceManager: The three components of IRL core are implemented as a single process. To avoid a single point of failure, the IRL is replicated on different hosts. In the case of a fault in an IRL core, the SmartProxy of the client will automatically switch over to a backup IRL core. The PersistenceManager is in charge in maintaining the states of the backup IRL cores. The PersistenceManager will notify the backup IRL cores of any state change in the primary IRL core.

When a client invokes a request on a server, the request will be intercepted by the Smart Proxy on the client side and then forwarded to the appropriate ObjectGroup in the IRL core. The ObjectGroup object is then in charge of executing the request on the appropriate servers and returning the final result.

The disadvantages of the two service approaches detailed above, OGS and IRL are as follows:

- The clients using OGS or IRL must be aware of the address of the host where the two servers are located. This reference must be either hard coded or it can be

retrieved from the Naming Service. If it is hard coded, then the OGS or IRL cannot be moved at any time in the life of the client and server. If the reference is retrieved from the Naming Service, then the Naming Service becomes a point of failure. Research could be made to attempt duplicating the Naming Service in order to alleviate the single point of failure problem but the reference to the backup Naming Service must also be hard coded.

- IRL requires that existing clients be recompiled and redeployed because the new clients require the IRL smart proxies. The smart proxies are ORB dependant and IDL interface dependant.
- OGS requires that the entire client application be recoded because OGS is not transparent to the programmer. Clients are aware of the OGS server and make DII request directly to it.
- OGS and IRL do address load balancing and versioning in a CORBA system but cannot address the problem of quality of service and network security.

#### 2.4.3.3 Fault-Tolerance using the OMG Object Trading Service

Research done by the Trinity College in Ireland [44] attempts to use the OMG Object Trading Service [10] as the mechanism to advertise and manage the object references of fault-tolerant components.

The basic architecture of the fault tolerant system described in [44] consists of clients, servers and the OMG Object Trading Service. A more complete architecture with advanced features is also described and includes a *Service Manager* and a *Notification Service*. The fault-tolerant fail over is described as follows: Clients needing to access a

service will connect to the OMG Object Trading Service and retrieve the object references to the master and backup servers. The reference to the backup server is cached and invocations are sent to the master server. In the case of a failure at the master Server, the client will detect the failure and then switch over to the backup server. All invocations are now sent to the backup server. The client must be able to switch back to the Master Server, as soon as it becomes available again. In the basic architecture, the client would use pings to determine when the master server is running again. As soon as the master server becomes available again, the client resumes sending all invocations to it. In the complete architecture, the authors describe a *Service* Manager, a *Notification Service* and a *Trading Service Manager* as well. The Notification Service would be in charge of notifying all the clients registered on the state of the servers. So instead of the client continuously pinging the failed master server, the Notification Service would asynchronously inform the client in the event that the master server would become available again. The complete architecture improves the performance of the system by freeing the client from constant pinging as well as reducing network traffic. The authors also discuss the possibility of introducing the idea of load-balancing into the system. When returning object references to clients, the OMG Trading Service would return a different object reference over the previous one by cycling through the list of Masters and Backups.

There are several disadvantages of the just described fault-tolerant system with respect to the CORBA switch. The first disadvantage is the number of points of failure in the system. In the complete system, there are two main points of failure:

- The OMG Object Trading Service and the Trading Service Manager
- The Notification Service and the Service Manager

If any one of these components were to fail, the whole system would become crippled.

One of the goals of the just described system is to hide all aspects of fault-tolerance from the application. They attempt to do so by putting the fault-tolerant code in smart proxies in the client. The problem is that the user must implement this smart proxy of each IDL interface, and to make matters worse, the smart proxies are ORB dependent. This fact makes the system not completely hidden from the application. To convert an existing application, the programmer must code the smart proxies and then recompile the client code and redistribute the client. Any old client still running will not be fault-tolerant.

The authors of the system do not plan for the situation where the administrator would want to change the backup server. In this case, the administrator would launch a new backup server and rebind the reference in the OMG Object Trading Service. But already running clients will have the cached reference to the old backup server. The same problem occurs if the administrator wishes to dynamically change the master server. These features may be implemented using the *Notification Service* but it is clear that the authors had not envisioned these situations. This approach also creates disruptions in the clients who have to remain idle until server failure is detected. The fault-tolerant system described in [44] addresses the problems of fault-tolerance and load balancing to some extent but not the problems of versioning, quality of service and network security.

#### 2.4.3.4 Object Migration using the Extended Naming Service

Similar work to [44] has been done for France Telecom/CNET [46] for versioning where an Extended Naming Service is used instead of the OMG Trading Service. The research conducted deals with the problem of migrating of CORBA objects from one location to other. The approach taken is to bind the reference of the new object in the Extended Naming Service for all new clients to use. For clients already holding the object reference, an exception will be generated when the old server is shut down, causing the client to retrieve the new reference from the Extended Naming Service. To hide all code from the client, smart proxies are again used to do the failure detection and retrieval of the new object reference from the Extended Naming Service. The migration of the CORBA objects involves the following steps:

- Step 1: The object blocks all incoming client requests.
- Step 2: The object saves its current state.
- Step 3: The object is removed from memory and launched in a different location. The object retrieves the state saved by the previous object. The object then registers its new object reference with the Extended Naming Service.
- Step 4: All clients holding references to the old object will retrieve the new object reference from the Extended Naming Service after detecting the failure to communicate to the old object.

This approach, similar to [44] which was described earlier, has many of the same disadvantages. To convert an existing application, the programmer must code the smart proxies and then recompile the client code and redistribute the client. Any old client still running will not switch over to the new object. The administrator must make sure that

running clients contain the new smart proxies. The authors do describe some IDL post-processor tool to automatically generate the smart proxies, but nevertheless the clients need to be recompiled and redistributed. When the migration is to be made, the current object must be shut down before the new object is launched. In some situations, the administration may need to run both versions of the server at the same time. This approach does not allow for such a situation. The authors do not address the problems of load balancing, fault tolerance, quality of service nor security.

## 2.5 Other Works

### 2.5.1 Schema evolution in object-oriented databases

A lot of research has been done in object versioning for object-oriented databases (OODBs). OODBs allow for storing and sharing of objects. OODBs came into existence because conventional relational database could not easily store complex data structures. A schema describes the classes in the OODB as well as their relationships (inheritance, association ...). A schema is set at design time and therefore the structure of the classes is fixed for the duration of the life of the OODB. A lot of work has been done on allowing for the on-the-fly modification of the schema. This is termed "Schema Evolution".

When the schema of an OODBs is modified, the database needs to be changed in such a way that the schema and the OODB remain consistent with each other. There are several approaches used today that provide this feature.

One approach involves the definition of object versions to evolve the database from one version to another. In these systems, an object that conforms to the new class structure is created. When a value of the old version is required, a transformation program retrieves the value from the old version and assigns it to the new version. The objects that are versioned are therefore not transformed when they versioned but a new version is created and the old versions of the objects are kept. Examples of such systems include GemStone [59], CLOSQL [61], and Encore [62].

In another approach, objects are also not physically restructured but a new object that conforms to the new class structure is created. A transformation function reads the value from the old object and assigns it to the new object. All references to the old version are updated and then the old version is discarded. This approach differs from the versioning approach in the fact that older versions of the versioned objects are not kept. Systems that use this approach include Itasca[60] and Versant [63].

All the research done in schema evolution deals with the problems and complexity of object versioning in OODBs. None of the research has dealt with the problems of fault-tolerance, load balancing and Quality of Service.

These approaches do allow for object versioning but the object still reside in the same database process. The one disadvantage of this approach is that versioning cannot be used for moving the objects from one physical machine to another. For example, this feature is useful when the hardware on which the database is running needs upgrading.

Another limitation is that all new versions of the objects must be coded in the same programming language as the original version. This limitation is not present when using the CORBA switch.

#### 2.5.1.1 Dynamic Reconfiguration of CORBA-based Applications

Research conducted in [31][66] takes the approach of introducing the concept of a metamorphic object to COBRA. The metamorphic object is able to dynamically change its class definition, implementation and interface without destabilizing the system by invalidating object references. The mechanism used in [31] for implementing metamorphic object is forwarding. When an object is versioned, it simply forwards all incoming requests to its new version. Figure 2-19 illustrates the mechanism of Forwarding.

When object A_1 it versioned to object A_2, object A_1 becomes a forwarder. All incoming requests to object A_1 are then forwarded on to object A_2. The process repeats itself for every new version of object A.

The research done in [31] introduces a new mechanism for object versioning using a forwarding pattern. The research has not dealt with the problems of fault-tolerance, load balancing. Several points of failure exist in the system, one for each old version of a versioned object. For example, in Figure 2-19 when object A_1 is versioned to object A_2, object A_1 becomes a forwarder but also an extra point of failure. When object A_2 is versioned to object A_3, object A_1 and object A_2 become points of failure.

Extra delay is also added to the system every time an object is versioned. Since for every version the request must go through a forwarder, the more old versions exist the longer the delay.

Unlike schema evolution, the approach described in [31] allows for the versioning of an object to a new version that may be implemented using a different programming language. But like many of the works in this field, existing servers need to be recoded to introduce versioning. Also a software programmer would have to go through a learning curve to learn how to program using this new environment.

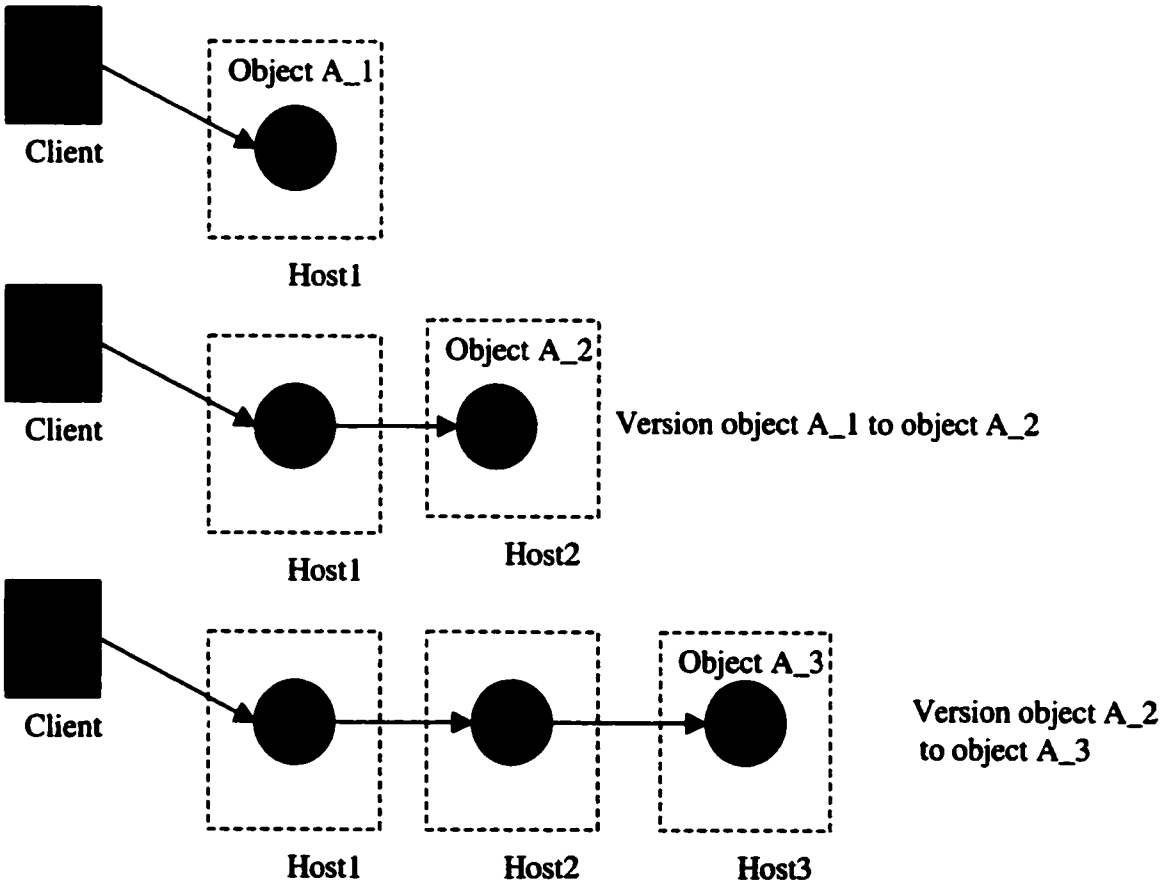


**Figure 2-19 Dynamic Reconfiguration of CORBA-based Applications**

# 3 CORBA Switch

The objective of CORBA switching is to increase the reliability and availability of an existing or future distributed CORBA application with the least amount of modification to the CORBA clients and CORBA servers. To accomplish this objective, a network approach is undertaken in this thesis. This requires that the intelligence needed to increase the reliability and availability of a distributed CORBA application is no longer placed at the end points of the communication process, i.e., the CORBA clients and CORBA servers, like in all the previous attempts (Integration Approach, Interception Approach and Service Approach). Instead the intelligence is placed in a network element (NE), therefore becoming invisible to the end points of the communication process, i.e., the CORBA clients and CORBA servers. An NE could be an Ethernet switch, IP router or any other type of switching product. Only one NE is required to have the added intelligence to treat CORBA packets while all the other components in the network need no modifications. This one network element with the added intelligence is termed the "CORBA switch". It is responsible for adding the different features to CORBA while remaining totally invisible to the CORBA clients and CORBA servers.

Introducing a CORBA switch into a network running a distributed CORBA application will convert the CORBA application from a non-reliable application into a high-reliability high-availability distributed application by introducing the following features to the CORBA application: fault tolerance, versioning, load balancing, quality of service and network security. The CORBA switch accomplishes this transformation without any

modifications to the client and server and with minimal down time. The CORBA switch increases the reliability and availability of the system by introducing the following concepts:

- Fault-tolerance: The CORBA switch will introduce fault tolerance to the CORBA application using the active replication approach. Fault tolerance will allow CORBA server failures to pass unperceived by the CORBA clients. These failures may be due to any number of factors (software bug, hardware bug, power loss). The fault tolerance feature is added to CORBA by replicating the client CORBA requests onto multiple CORBA servers. Any failure in any of the CORBA servers will not be sensed by any of the clients because the backup CORBA servers will take over the clients' requests.
- Versioning: Versioning allows for a CORBA server to be replaced with a newer version of the same CORBA server without interruption to the system. For example, the new version may be the same CORBA server but with some bug fixes, an implementation in a different language or coded in a different method. If the physical hardware on which the CORBA server is running needs upgrading, then the CORBA server is versioned to the same server but running on a different physical machine. This will allow the administrator of the system to upgrade the hardware without interruption of the service offered by the CORBA server. The CORBA switch will accomplish versioning by redirecting all requests from the old version to the new version. This is done seamlessly and without the knowledge of the CORBA clients.

- Load balancing: Load Balancing distributes execution load of the CORBA requests onto many different CORBA servers. By distributing the load of the execution of the CORBA requests onto many CORBA servers, this allows for the system to be able to handle a higher load of CORBA requests from the CORBA clients. The CORBA switch allows for load balancing by distributing the client requests onto many servers. This is transparent to the CORBA client who is not aware of which CORBA server is executing the request.
- Quality of service. The CORBA switch introduces quality of service into CORBA through flow control and queuing. All CORBA requests coming to the CORBA switch are queued to allow only a limited number of requests to be executed simultaneous by the CORBA servers. The CORBA switch therefore allows the administrator to control the amount of CORBA requests reaching the CORBA servers. If the queue is full, the CORBA switch selectively drops low priority requests, giving high priority requests a greater chance of been executed. Through this flow control and queuing, the administrator can assign different quality of service to different clients or groups of clients.
- Network security: The CORBA switch deals with only one aspect of network security. The CORBA switch only deals with Denial Of Service attacks. Denial Of Service attacks involve flooding the targets of the attack with dummy requests therefore overwhelming the servers and not allowing legitimate requests of being executed. The CORBA switch adds the capability of filtering out requests from abusive and unauthorized clients and therefore stopping the flooding from reaching the CORBA servers.

The CORBA switch is able to increase the reliability of the CORBA application to provide the following advantages:

- The CORBA switch does not require that any CORBA clients and any CORBA servers be recoded or recompiled. Existing CORBA clients and CORBA servers can be used in the new environment. This is a tremendous advantage over previous approaches in that no time is spent for the administrator and his or her team in recoding existing software. This reduces the cost of ownership of the CORBA switch and reduces the potential problems. Some problems may be caused by bugs introduced into the existing CORBA clients and CORBA servers while modifications are being made to accommodate the previously described approach. There must also be time and money spent on training the programmers if a previously described approach is used. When no modifications are necessary, like with the CORBA switch, the deployment of the system becomes very easy and cheap as well as low risk.
- Introducing only one CORBA switch into a system creates a single point of failure, the CORBA switch itself. As described earlier, only one network element is replaced by a CORBA switch, concentrating all the intelligence into one network element. This CORBA switch can therefore be physically replicated and when one fails, a backup one can immediately take over without interruption. The CORBA switch therefore, does not introduce a single point of failure.

- The CORBA switch is completely transparent to the CORBA clients and CORBA servers. At the end points of the system, no change is detected when the CORBA switch is introduced. This is one of the reasons why the CORBA clients and CORBA servers do not need to be recoded for use with the CORBA switch.

## 3.1 Design Assumptions

A current distributed CORBA application includes any number of clients invoking requests on any number of servers. The clients and servers are distributed over different hosts all connected together through some network. For the purpose of this thesis, an IP[7] network is considered to interconnect the different hosts. CORBA applications use the General Inter-Orb Protocol (GIOP) [6] for communication between clients and servers. The most used implementation of GIOP is the Internet Inter-ORB Protocol (IIOP) [6], which operates over TCP/IP. The fact that most networks today are IP-based explains the popularity of IIOP. The CORBA switch is designed to work with IIOP due to the popularity of IP-based networks and the low cost of IP-based equipment. The CORBA switch could potentially be modified to work with other connection-oriented protocols other than TCP.

All communication between a client and a server in the current CORBA specification is point-to-point. Figure 3-1 shows a typical distributed CORBA application. When a client first needs to invoke a request on the server, a TCP connection is opened. When the TCP connection is set, all CORBA messages, requests and replies, are passed between the client and the server using this opened TCP connection. TCP provides a reliable

connection-oriented means of communication for IIOP. Reception of the messages is guaranteed by TCP, which uses acknowledgments and retransmissions to guarantee safe delivery of the sent messages.

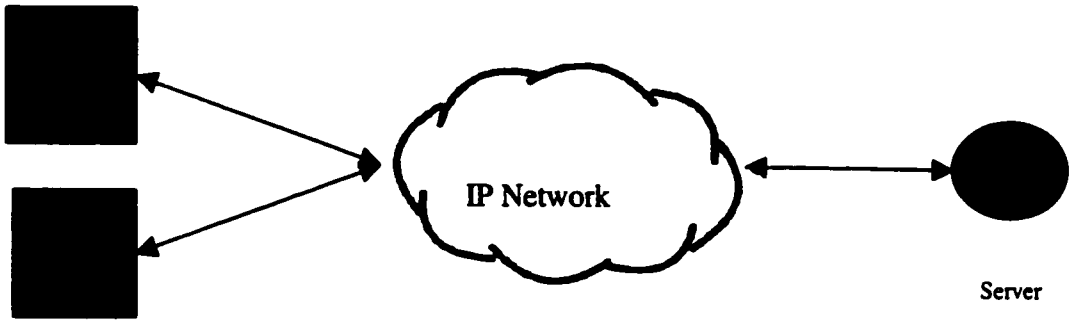


**Figure 3-1 Typical CORBA Application**

The CORBA switch introduces the network approach to improving the reliability of CORBA. The network approach involves introducing some intelligence into the network by delegating some of the work to a network element. In the CORBA switch approach, the intelligence must be implemented on a network element through which the traffic between the clients and the servers has to pass. That network element would then be called the CORBA switch. The network element must be carefully chosen so that all traffic between the clients and the servers must travel through it. The CORBA switch will therefore have all the CORBA requests and replies pass through it, allowing it to implement the features that will increase the reliability of CORBA. Figure 3-2 shows that all traffic between the clients and the servers is forced to pass through the CORBA switch. To ensure that the CORBA switch itself does not become a single point of failure, a backup CORBA switch can be introduced in parallel with the master CORBA switch. The master CORBA switch is responsible for transferring its state to the backup switch so

that in case the failure of the master, the backup switch would take over. The backup CORBA switch eliminates the problem of a single point of failure.

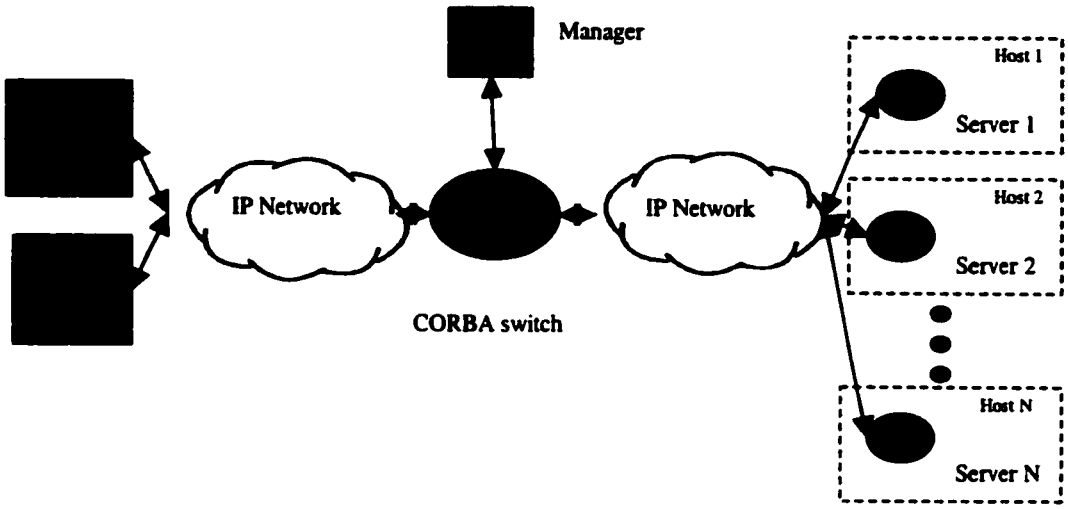


**Figure 3-2 Typical CORBA Application with CORBA switch introduced**

In typical network deployments, the servers are placed together into groups called server farms. These servers are usually placed physically together in large rooms or warehouses. These servers are connected together and to the outside world through a switch. This switch may be a layer 2 switch or a layer 3 switch (also know as a router) and is connected to the servers through the Ethernet protocol [52] over copper wiring. In this thesis, a layer 2 switch will refer to a switch that makes switching decisions based on an OSI layer 2 destination address. For example, certain layer 2 switches use the destination MAC address found in the Ethernet header to decide where to send the packets. The layer 2 switch looks in its address table to match the destination MAC address to an output port. A layer 3 switch is defined as a switch that uses an OSI layer 3 destination address.

For example, certain layer 3 switches use the destination IP address found in the IP header to decide where to send the packet. The layer 3 switch looks in its forwarding table to determine the output port for the packet.

The switch, being a layer 2 (L2) or layer 3 (L3) switch, is then connected to the clients through a network with IP running on top. This network may be a company's private network or the public Internet. This typical setup is shown in Fig 3-3. All traffic to and from the server farm must pass through the switch.

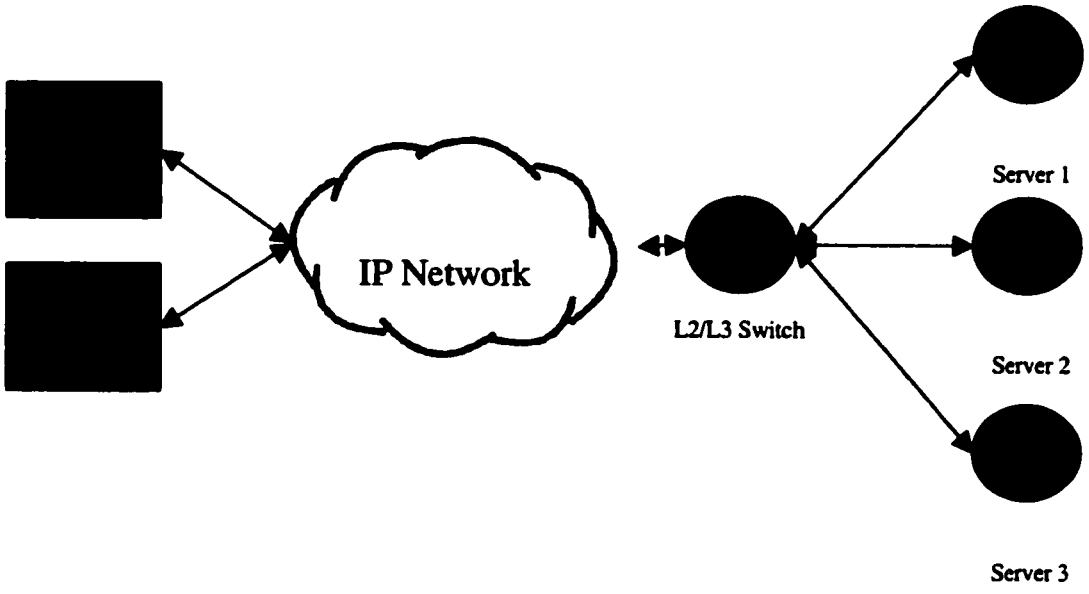


**Figure 3-3 Typical server farm setup**

Replacing the switch connecting the server farm to the world with a CORBA switch will introduce more reliability into the CORBA system. The CORBA switch is implemented using hardware and software components. The CORBA switch contains a component that implements a Layer 3 or Layer 2 switch and components responsible for the CORBA switching. If the CORBA features of the CORBA switch are not used the CORBA switch

acts like a Layer 3 or Layer 2 switch. The CORBA switch also requires an administrator application that controls the CORBA switch. This application can be implemented on the CORBA switch and accessed by the client remotely from any machine. The CORBA switch, being placed between the clients and the server, forces all messages to pass through it on their way to and from the servers.

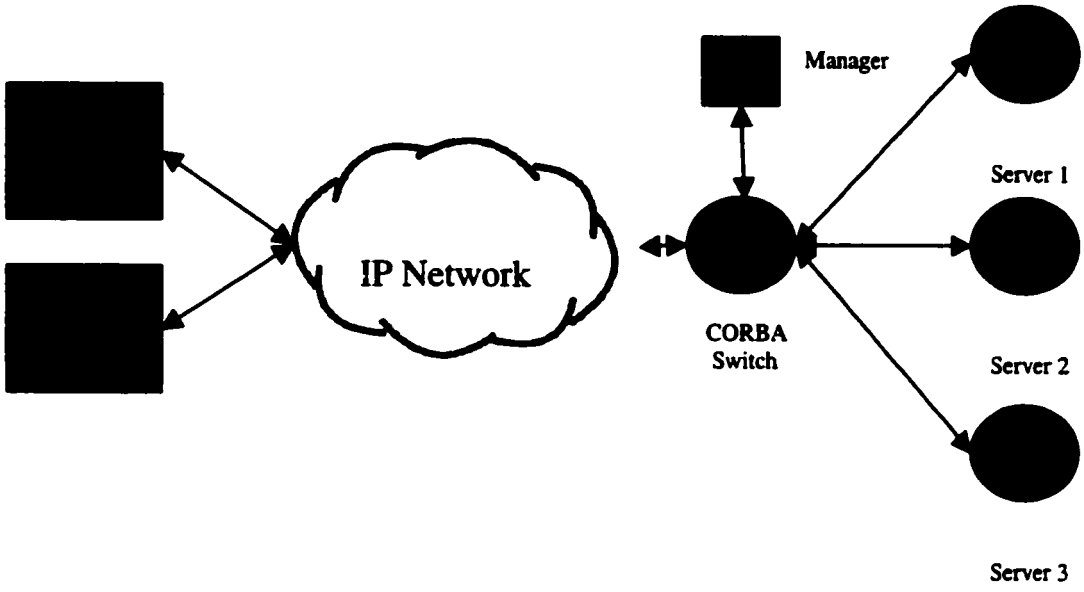


**Figure 3-4 Typical server farm setup with CORBA Switch introduced**

### 3.1.1 Functional Specification of the CORBA Switch

#### 3.1.1.1 Traditional Client-Server Operation

When a client has a CORBA invocation to be made, the following events occur:

1. The client checks the IOR object reference of the target CORBA object to obtain the IP address and the TCP port of the CORBA server on which the target CORBA object resides.

2. The client then initiates a TCP connection with the CORBA server using the IP address and TCP port obtained from the IOR object reference of the target CORBA object.
3. After the TCP three-way handshaking is accomplished, a TCP connection is open.
4. Over this open TCP connection, the client and server can exchange CORBA messages. TCP will guarantee transmission of the messages through retransmission in case of failure.

#### 3.1.1.2 CORBA Switch operation

The CORBA switch represents replicated servers as a server group. When the CORBA switch is first introduced into the system the following steps are taking by the administrator through the administrator application:

1. Create a server group with a certain policy. The policy may be fault-tolerance, versioning, load balancing or smart load balancing.
2. The administrator then enters the replicated servers, one at a time, by inputting the IP address and TCP port number of each server.
3. The CORBA switch then establishes a TCP connection with each server.
4. The TCP connection is kept alive by periodic transmission of TCP messages.

When a client makes a CORBA call on a server, the following events occurs:

1. The client checks the IOR object reference of the target CORBA object to obtain the IP address and the TCP port of the CORBA server on which the target CORBA object resides.

2. The client then initiates a TCP connection with the CORBA server using the IP address TCP port just obtained from the IOR object reference of the target CORBA object.
3. The CORBA switch intercepts the TCP handshaking messages and accepts the TCP connection from the client on behalf of the CORBA server.
4. The client then transmits the CORBA requests to the CORBA switch.
5. The CORBA switch in turn forwards the CORBA request to the appropriate server depending on the policy chosen by the administrator.

## 3.2 Architecture of the CORBA switch

The CORBA switch consists of three main components. The three components are shown in Figure 3-5:

- **L2/L3 switching component**: This component acts like an Ethernet switch or IP switch in every way. Any incoming packet with destination one of the server will be switched to the output physical port to which the server is connected and the same applies to outgoing packets. Messages such as icmp pings [50] or DNS requests [49] passing through are handled by this component. This component has the added feature of a filter that is controlled by the CORBA switching component. When an incoming packet matches the filter set by the CORBA switching component, the L2/L3 switching component redirects the packet to the CORBA switch component instead of the destination server. When the CORBA switching component has completed all work on the packet, it may choose to ask

the L2/L3 switching component to continue transmission of the packet, which may or may not have been modified by the CORBA switching component.

- **CORBA switching component**: This component is involved in the actual CORBA switching. This component resides on the same host as the L2/L3 switching component for the purposes of reducing delays caused by the CORBA switch. After receiving a packet from the L2/L3 switching component, the CORBA switching component will decide on the course of action to take according to the policy entered by the administrator using the Administrator Application.
- **Administrator Application**: This application will allow the administrator of the system to setup the different policies (fault-tolerance, load-balancing, versioning), setup the different servers, monitor the system and receive notification of server failure.

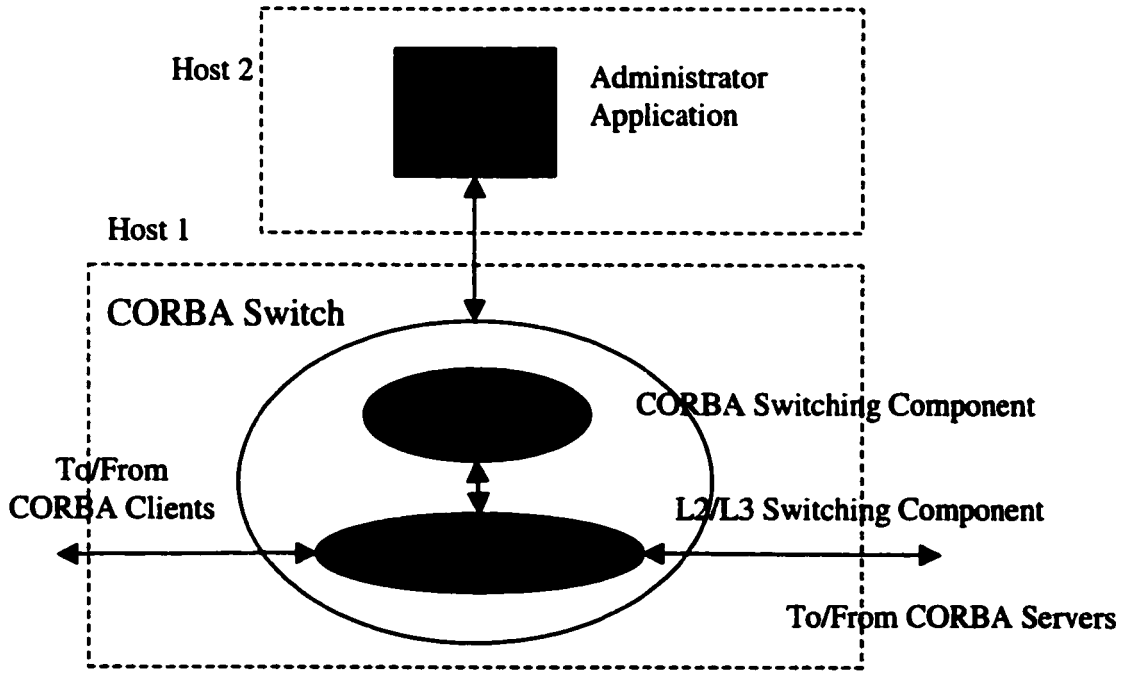


**Figure 3-5 Architecture of The CORBA Switch**

### 3.2.1 L2/L3 Switching component

The L2/L3 switching component is implemented to act in the same manner as a conventional Layer 2 switch [55] or Layer 3 switch [34]. The L2/L3 switching component consists of several different components: physical uplink port, uplink filter, physical download ports, switching fabric, Central Processing Unit (CPU) and an Interface with the CORBA switching component. Through the physical uplink port, the CORBA switch is connected to a private backbone network or to the public Internet. The CORBA switch is connected to the CORBA servers through the physical download port. Typically downlink physical ports use the Ethernet Protocol to communicate with the servers over Copper or Fiber. The uplink physical port may be implemented using different technologies including Ethernet. When a Layer 2 switch is used, the packets are switched according to the MAC address of the destination address. If an L3 switch is used, the packets are switched according the IP address of the destination. Either type of switching can be used with the CORBA switch the only difference is the setup used by the administrator in deploying his or her network.

The architecture of an L2 switching component is shown in Figure 3-6. The architecture includes the following components:

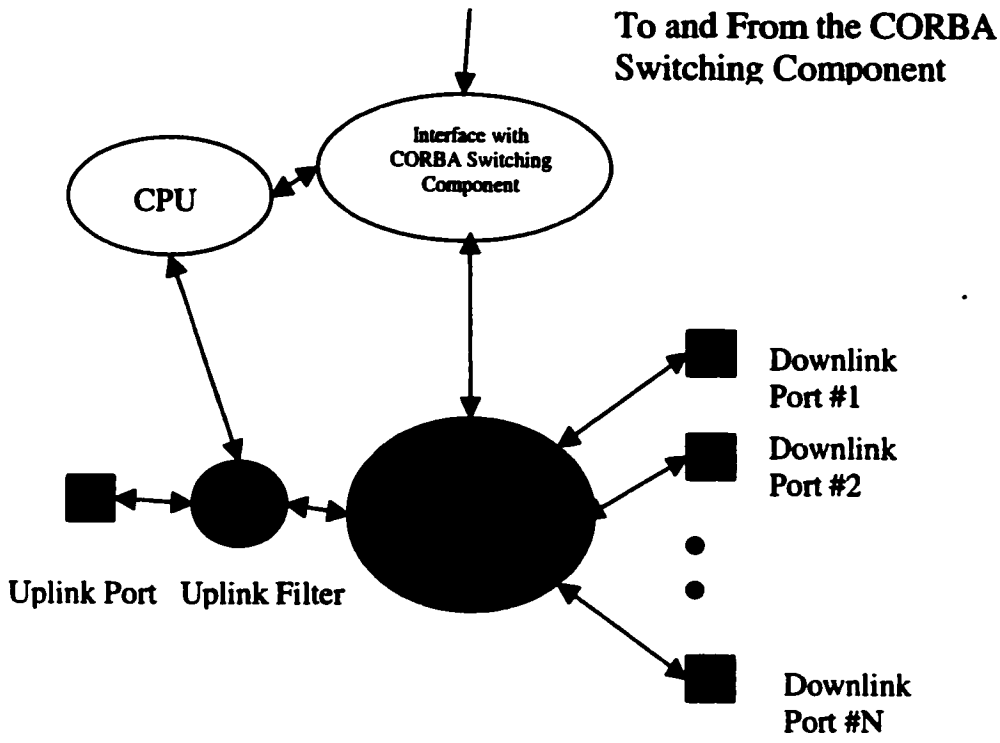


**Figure 3-6 Architecture of L2 switching component**

#### 3.2.1.1 Physical UpLink and Downlink Port

Through the physical uplink port, the CORBA switch is connected to the rest of world through an IP-based backbone network. This backbone network will, in most typical situations, be a company's private intranet. This port is an interface, which is responsible for the lowest layer header and with the actual physical transmission of the packets on the physical wire. For example, in the case on a Layer 3 switch, this port is responsible for the all protocol headers below the IP header.

#### 3.2.1.2 UpLink Filter

Incoming packets from clients will enter the CORBA switch through the Physical UpLink Port. All packets from this port, CORBA or not, will be passed through a filter. This filter compares the packets headers to certain criteria. If a match is found, then the

packets are flagged and sent to the switching fabric. The switching fabric would in turn forward the packet to the interface with the CORBA switching component. The CORBA switching component sets the filter criterion. The criterion consists of an <IP address> and a <TCP port>. One such criterion is set for each server in the server farm. An <IP address>,<TCP> pair uniquely identifies a CORBA server. Any packet, which is destined to one of the CORBA servers, will be redirected to the CORBA switching component. Any other packets will be switched normally. For example, an icmp ping [49] would not match any of the criteria and will be switched to the appropriate Physical Downlink Port to which the target server is connected.

There is no need to intercept packets from the downlink ports since the destination of the packets is the CORBA switch. The packets coming through the uplink port have the CORBA servers as destination and therefore must be intercepted by the uplink filter.

#### 3.2.1.3 Switching Fabric

The switching fabric provides a communication bus between all the components in the system. The switching fabric is of the same kind as the commercially available routing and switching products. In this Thesis, a switching fabric was emulated in software on a Linux machine. The switching fabric takes packets from the Physical Uplink Port and switches it to the appropriate downlink port. The switching fabric also takes packets from any of the Physical Downlink Port and switches it to the Physical Uplink Port or even one of the other Physical Downlink Ports. If a packet is marked by the uplink filter, then the switching fabric switches the packet to the Interface with the CORBA switching component. When the CORBA Switching component wants to transmit a packet, the

packet is passed to the switching fabric, which in turn switches the packet to the appropriate port.

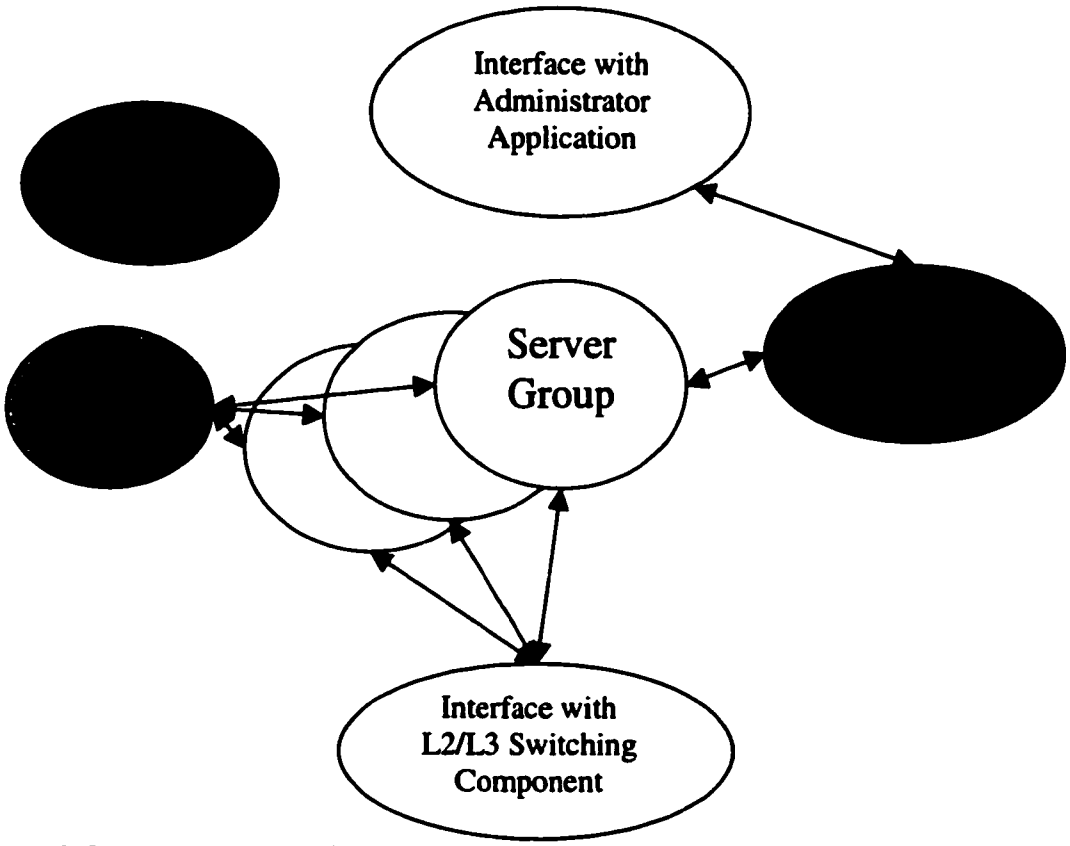


**Figure 3-7 Architecture of CORBA switching component**

### 3.2.2 The CORBA switching component

The CORBA switching component is responsible for the switching of CORBA messages. CORBA messages are intercepted by the L2 switching component and then passed to the CORBA Switching component. The CORBA messages are then forwarded to the proper destination, according to one of the different policies. These policies include fault-

tolerance, versioning and load balancing. The CORBA switching component consists of the following components:

#### 3.2.2.1 Interface with L2/L3 switching component

Through this interface the CORBA switching component is able to configure the filter of the L2/L3 switching component as well as exchanging packet to and from the switching fabric. When a packet is intercepted by the filter in the L2/L3 switching component, it sent to the CORBA switching component through this interface. When the CORBA switching component needs to transmit a packet, it is sent to the switching fabric through this interface for transmission.

#### 3.2.2.2 Replication Manager

If only one CORBA switch is used in a system, then the CORBA switch becomes a single point of failure. One of the requirements of the CORBA switch is to not allow for a single point of failure. To accomplish this, the CORBA switch is replicated. The Replication Manager is responsible for monitoring the state of the CORBA switch and transmitting any changes to the backup CORBA switches. Any time a state change occurs in the active CORBA switch, it is transmitted to the backup CORBA switches. The backup CORBA switches remain inactive while the main CORBA switch is running. In the event of a failure in the CORBA switch, the backup CORBA switch can immediately take over without interruption. The detection of failure is accomplished by continuous transmission messages of a *is_alive* message between the main and backup CORBA

switch. When the messages are no longer received from the main CORBA switch, the backup assumes a failure has occurred and takes over the load from the main CORBA switch.

#### 3.2.2.3 Mini Interface Repository (miniIFR)

This repository holds some information about the IDL interfaces of the servers that are needed in some cases when object references are being passed as function arguments. If object references are passed as function arguments, they must be checked to make sure they do not reference an invalid object. The knowledge of the IDL interfaces of the servers allows the CORBA Switch to decode and modify the arguments of all the function calls. A more detailed explanation is given in 3.4.1. IDL interface definitions can be loaded into the IFR through the Administrator Application.

#### 3.2.2.4 Server group

CORBA servers are grouped into server groups. For example, in a fault-tolerant system, all copies of the same CORBA server are grouped in a server group. When using the versioning policy, all versions of a server are grouped in one server group. A server group is represented by an instance of a server group object in the CORBA Switch. The server group object is responsible for the actual switching of the CORBA messages depending on the policy of that particular group. The policy may be fault tolerance, versioning, client/request load balancing and smart load balancing.

To reduce bottlenecks, it is recommended that only one server group is used per CORBA switch and that for multiple types of servers, different CORBA switches be used. But for low traffic applications, the use of multiple server groups per CORBA switches might be the most efficient approach.

## 3.3 CORBA Switch Policies

A server group is a group of replicated servers with an assigned policy. The policy is assigned by the administrator at the creation of the group through the Administrator Application. The different policies available for the user are:

- Fault-tolerance
- Load balancing
- Smart load balancing
- Versioning

### 3.3.1 Fault Tolerance

The fault-tolerance policy is used to guard against faults in one or more of the CORBA servers in the group. Any crash of the one of the CORBA servers would be undetectable by any of the clients. Two techniques are implemented in the CORBA switch for fault-tolerance for CORBA servers.

#### 3.3.1.1 Active replication

In this technique of replication, all CORBA requests are sent to all the servers in the group. To guarantee that all the servers in the group maintain the same state, the server group object must guarantee that all CORBA requests are received in the same order by all the servers in the group. If the order is not guaranteed then the state of the servers in a fault tolerant server group may become inconsistent. The server group object will receive N CORBA replies, with N representing the number of CORBA servers. There are two options for dealing for the CORBA replies at the server group Object:

1. First Come First Serve: The first CORBA reply received the server group Object is sent back to the user while the order CORBA replies are dropped. This is the faster fashion of dealing with the duplicate CORBA replies in terms of delay at the CORBA switch.
2. Voting: The CORBA replies received are collected and then compared amongst each other by the server group object. The CORBA replies that appear the greatest number of times amongst the CORBA replies is considered as correct and sent back to the client. This approach introduces a greater amount of delay into the system than the First Come First Serve approach but returns a more reliable reply to the client. The server group object must wait for all CORBA replies to return from the CORBA servers but must be careful not to exceed any timeouts at the client.

### 3.3.1.2 Passive replication

Passive replication, also known as the primary backup approach, is used with stateless servers. When a CORBA server crashes, the server group object will forward the CORBA request to an already running backup server. This server must have the same state as the primary server. One of the requirements of the CORBA switch is to increase the reliability of CORBA without ANY modification to existing servers and clients. Because of this fact, already implemented and running CORBA servers have no facilities for transfer of states. This is the reason why the passive replication technique can only be used with stateless servers. If some kind of mechanism of state transfer is introduced to new CORBA servers, the passive replication technique can then also be used with these servers with state changes.

The Sequence diagram of the interactions of all the components in the system is shown in Figure 3-8 in the case of fault-tolerance.

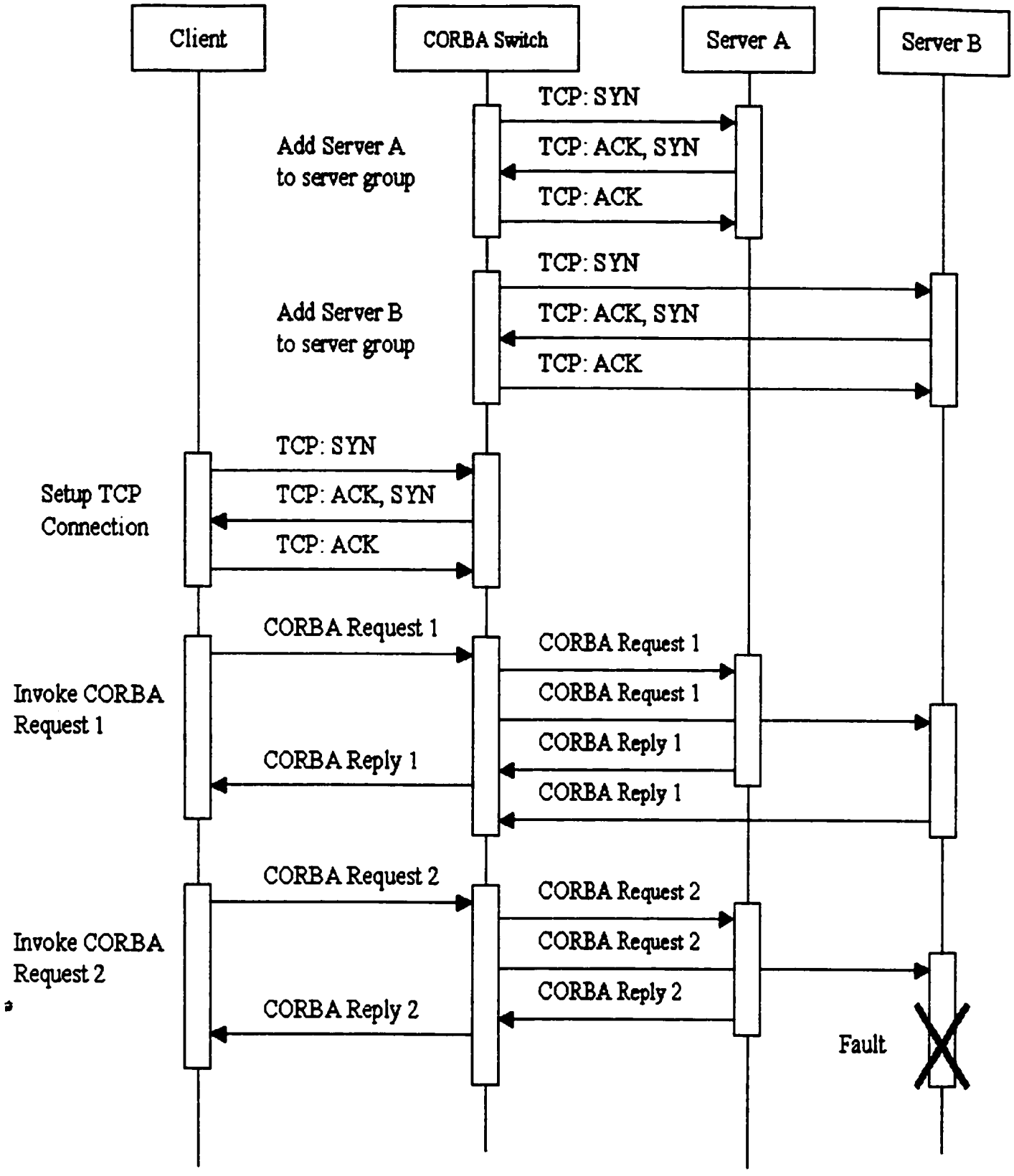


**Figure 3-8 Sequence Diagram for Fault-Tolerance policy**

### 3.3.2 Load Balancing

Load Balancing attempts to distribute the CORBA requests onto many CORBA servers so as not to overload any of the CORBA servers. Load Balancing only applies to stateless servers. The Sequence Diagram for load balancing is similar to the Sequence Diagram shown in Figure 3-8, excepting the facts that Request 1 goes to server A and Request 2 goes to server B. There are two types of load balancing used by a server group object:

#### 3.3.2.1 Request Load Balancing

When a CORBA request is received by the server group object, a server is picked from the list of replicated CORBA server and the CORBA request is sent to it. The process of choosing servers from the list depends on the load percentages indicated by the administrator through the Administrator Application. If the administrator chooses equal weighting for each server, then the server group object will distribute the CORBA requests in an equal fashion amongst all the CORBA servers. If the administrator assigns different weightings, then the CORBA server will distribute the CORBA requests accordingly.

#### 3.3.2.2 Client Load Balancing:

Every client is assigned a different server chosen from the list of replicated servers by the server group object according to their weight. All CORBA requests from one client are sent to the same CORBA server unlike to request load balancing approach.

### 3.3.3 Smart Load Balancing

Smart load balancing applies to applications with state where the majority of requests do not modify the state of the servers but simply retrieve information. For example, the CORBA Naming service is mostly used for retrieving object references using the function call *resolve*. The state of the CORBA Naming service can be modifies by adding or removing entries through the function calls *bind* and *unbind*. The state of the CORBA Naming Service is modified on occasion while most of the calls are of type *resolve*. The server group object will forward all the state-modifying requests (i.e. *unbind* and *bind*) to all the servers in the group to maintain the state but will load balance all the non-state modifying requests (i.e. *resolve*). This technique allows for the introduction of the load-balancing concept to servers with state.

The administrator using the administrator application must therefore enter a list of all state modifying function calls. The list of state modifying function calls is unique for each server group and is kept in the miniIFR.

### 3.3.4 Versioning

The CORBA switch deals with server versioning and not object versioning. The CORBA switch is not able to deal with object versioning because of the problem of inter-object communication of objects residing on the same server. For example two objects A1 and B1 reside on server1. First object A1 is versioned to object A2 on server2 and object B1 is versioned to object B3. If A2 attempts to communicate with object B, it is likely that it

holds a reference to version B1. The request made by object A2 does not pass through the CORBA switch because all the servers reside behind the CORBA switch. The CORBA request can therefore not be redirected to B3 and an error will occur. The CORBA switch therefore only considers server versioning where the entire CORBA server, including all objects running in the server, is versioned.

The server group object receives the packet from the switching fabric matching a server that was versioned. The server group object will then look up the new address of the server and send the CORBA request to the CORBA server through the already open TCP connection. A server can be versioned at any time but already received CORBA request will still be executed by the old version of the server.

Using these versioning capabilities of the CORBA switch, an administrator may version a server A to a newer version B but may also very quickly revert to the older version A. The only requirement on the CORBA server is that they are stateless. One of the requirements of the CORBA switch is to increase the reliability of CORBA without any modification to existing servers and clients. Because of this fact, already implemented and running CORBA servers have no facilities for transfer of states. This is the reason why versioning can only be use with stateless servers.

## 3.4 Other Components

### 3.4.1 Mini-Interface Repository (miniIFR)

The miniIFR is a repository that holds the format of the CORBA requests. For each function call, it holds the types of arguments that are passed as well as the return type. There are two situations that may arise where the miniIFR is needed.

The first situation is when smart load balancing is used and a list of state modifying requests is required.

The second situation is where an Object Reference is passed as a parameter to the server from the client. If an object reference to an object, residing on the server that was versioned, is passed to another object on a different version of the same server, then that reference must be modified by the CORBA switch to match the second server. The following example illustrates the situation in greater detail. This situation applies to three of the four policies: fault-tolerance, load balancing and versioning. The situation is illustrated in the case of versioning.

The setup includes one client and two servers, version A and version B, and is shown in Fig 3-13. Each server has two objects running: an object Bank and an object Account. Server A is the current version and Server B is the newer version to be versioned to at a later time.

The Bank object allows for two CORBA calls to be made:

- One called *getAccount()* which returns a reference an Account object,
- And one called *balanceFromAccount(Account_ptr acc)* which retrieves the balance from the account object pointed to by the reference *acc*.

The following sequence of events will cause a problem that can only be resolved by the miniIFR.

1. The Client invokes *getAccount()* on object Bank A. The client now holds a reference to Account A.
2. The administrator versions Server A to Server B. Now all requests will be redirected to Server B.
3. The Client now invokes *balanceFromAccount()* on object Bank A, passing the reference to Account A.
4. The CORBA switch redirects the request to the object Bank B.
5. Bank B tries to access the object Account A. Server A was versioned and the object Account A is no longer valid. Bank B should access object Account B instead. This is the reason why the miniIFR is needed.

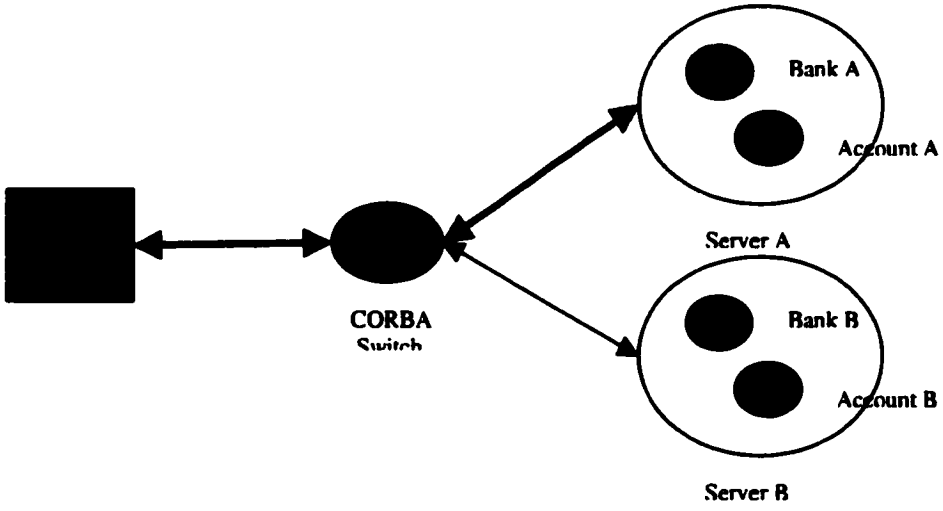


**Figure 3-9 Example of miniIFR use**

The CORBA switch, or the server group object more specifically, must decode the CORBA request and extract all object references. If the object references are for an object from a server from the server group or that was previously in the group, the

CORBA switch must then modify the object references to reference to same objects on the current version of the server.

CORBA function parameters are coded into the CORBA request using a common syntax. This syntax must be standard as to ensure vendor interoperability. To this end, OMG defined the CDR transfer syntax specification [6]. One of the properties of the CDR is that to decode the coded data, the data types must be known before hand. Therefore to decode the object reference from the CORBA request, the CORBA switch must be aware of the data types of the parameters in the CORBA request. The only way to be aware of the data types is for the administrator to provide the IDL definitions of the interfaces of the servers in the server group. This can be easily done using the administrator application. The entire IDL definition is not needed for all functions, only the functions that have object references as a parameter. That is why a miniIFR is needed instead of a complete IFR. Only that little bit of information is needed. In most situations, object references are not passed as parameters and the miniIFR will not be needed. But for the specific situations where object references are passed as parameters, the CORBA switch can retrieve any relevant information about a CORBA function from the miniIFR.

### 3.4.2 Administrator Application

Through the Administrator Application, the CORBA switch can be configured and monitored from a remote location. Communication between the CORBA switch and the Administrator Application is done through CORBA, allowing the Administrator Application to run virtually anywhere. Using the Administrator Application, an administrator may:

- Create/Delete server groups

  Server groups can be created at any time. In the case where several types of servers reside down stream from the CORBA switch, many server groups will be needed.

- Add/Remove servers from a server group

  The adding and removing of servers from a server group is done dynamically without any interruptions in the system. An administrator may add a new server to distribute the load (using load balancing), or a new version of server (using versioning).

- Upload the IDL interface definitions of the CORBA servers in the server group to the miniIFR.

- Version a server in a server group with Versioning policy.

- Be notified in the event of a CORBA server failure.

  Using the CORBA Event Service [1], the CORBA switch can send an event to the Administrator Application indicating a CORBA server failure. The CORBA Event Service allows for asynchronous communication between CORBA clients and servers. The Administrator Application may then signal the administrator in different ways (Beep, Pop Window...)

# 4 Extensions to the CORBA Switch

## 4.1 Quality of Service

### 4.1.1 Overview

Quality of service or flow control capabilities can be added to the CORBA switch to enhance the reliability of the CORBA system. The CORBA Switch deals with only one aspect of Quality of Service. The CORBA Switch can provide different levels of service to different CORBA clients by controlling the flow of CORBA requests to the servers. By using queues and selective dropping the CORBA Switch can provide Quality of Service. Quality of service or flow control comes into play when the amount of CORBA requests overwhelms the CORBA servers. Even if the CORBA requests are load balanced amongst many different servers, there is always the possibility of a surge in demand that would overwhelm the CORBA servers and therefore reduce response time to all of the clients. To deal with such a situation, some of the CORBA requests must be dropped, reducing the demand on the CORBA servers. This can be accomplished by placing a buffer of CORBA requests in the CORBA switch itself. The CORBA switch would then only allow N requests to be simultaneously executed at any time by the CORBA servers. The limit N is to be determined by the administrator depending on the type of servers and the resources available in machine they reside on. The Administrator Application will allow that value to be set by the administrator. Any excess CORBA

requests will be placed in the buffer. When the buffer is full, CORBA requests will have to be dropped. There are two different approaches that can be used for choosing which packets are to be dropped:

**Last come first dropped**

When the buffer of CORBA requests is full, any incoming CORBA requests will be dropped. This approach is non-discriminative and will affect incoming CORBA Requests in a random fashion.

**Selective Dropping**

Selective Dropping allows the CORBA switch to discriminate against low priority CORBA requests. Low priority request would be dropped first, allowing the higher priority requests to go through. The administrator may give higher priority to paying customers or to more important clients. CORBA requests could be identified by the source IP address of the client. The administrator may set the CORBA switch to give higher priority to all requests from a list of IP addresses or from a list of subnets. Other types of filters may be envisioned such as giving certain CORBA requests higher priority, notwithstanding the source of the request, but depending on the effect it would have on the CORBA servers. For example, let us consider the CORBA Naming Service. An administrator could give higher priority to state modifying requests (*bind* and *unbind* operations) and give lower priority to none-state modifying requests (*resolve* operation).

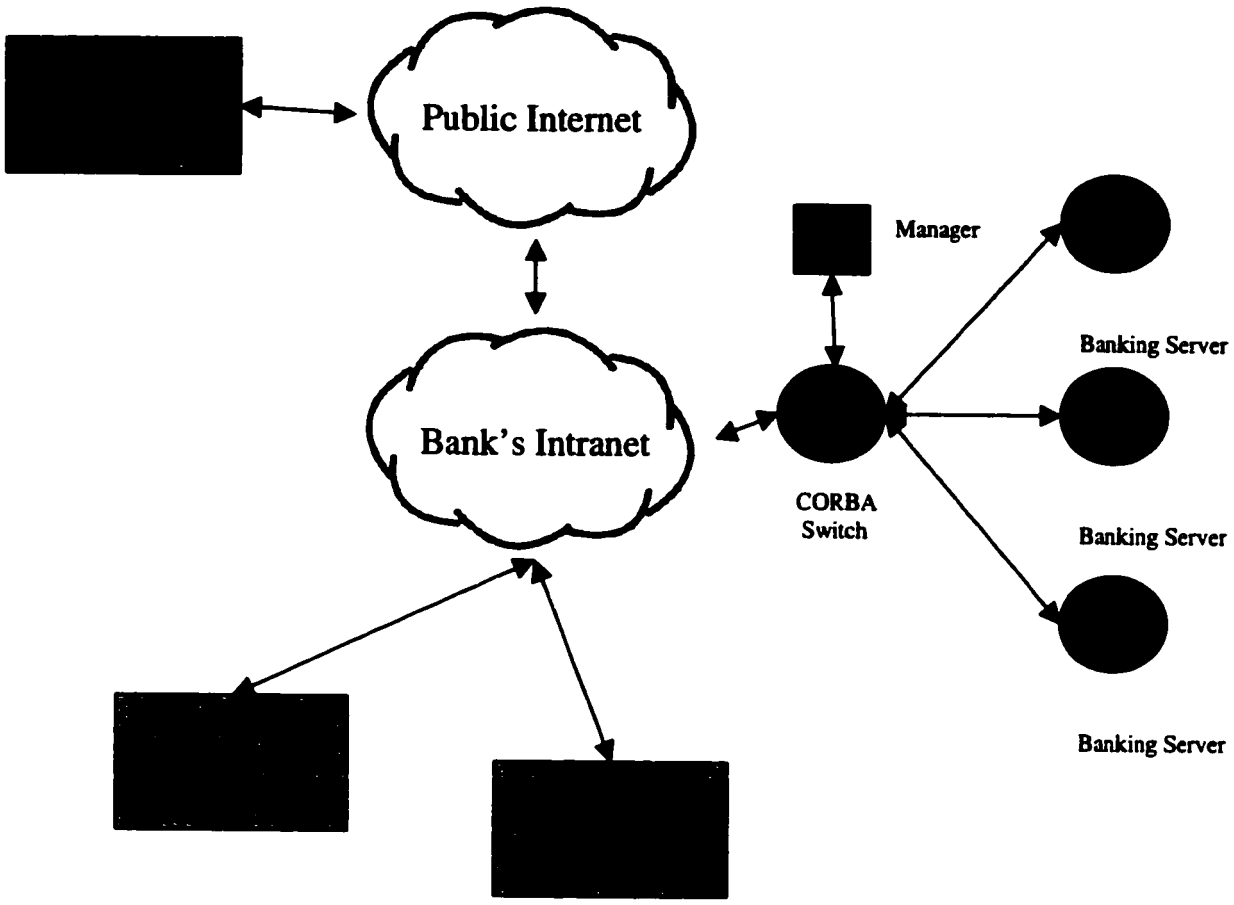


**Figure 4-1 Example setup**

The following example illustrates where the Quality of service or flow control feature of the CORBA switch may be used:

Let us consider the banking system shown in Fig 4-1. There are two sets of clients:

- The bank tellers that are located in the different branches of the bank around the country,
- And the Banking Online customers that are banking from home.

During peak hours, the banking servers might become overloaded due to a surge of demand. The bank would want to give higher priority to bank tellers and lower priority to

Banking Online customers. The bank tellers must have their requests handled first because the bank customers are waiting in line at the branch. The requests from the other customers can be dropped because the other customers are sitting at home or at work and can afford to wait. This is a situation where the quality of service or flow control feature of the CORBA switch would be needed.

### 4.1.2 New CORBA Switch Architecture

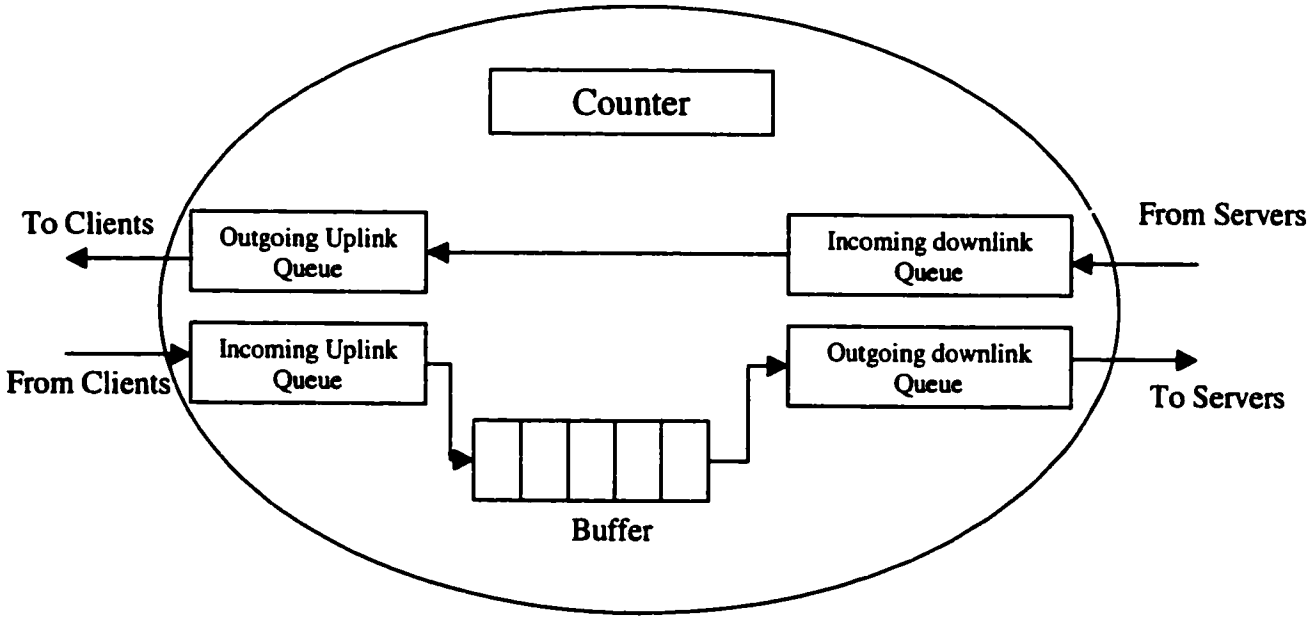


**Figure 4-2 New Architecture for the Server Group component of CORBA switching component**

The architecture of the server group component of the CORBA switching component must be modified to include a buffer. First the administrator sets the maximum number of CORBA requests that can simultaneously be executed by the CORBA servers. Using the

counter in the server group, the current number of CORBA requests being executed can be stored. When a CORBA request is sent to the servers, the counter is incremented. When a CORBA reply is received from the servers, the counter is decremented. When the counter reaches the maximum allowable value, the CORBA requests are no longer sent to the servers but are instead placed in a buffer. As soon as a CORBA reply is received, the counter is decremented and another CORBA request is sent from the requests waiting in the buffer. If the buffer starts to overflow, CORBA requests will have to be dropped according to one of the previously described techniques. This architecture will guarantee that none of the CORBA servers will be overwhelmed with a very large number of CORBA requests. Quality of service at the Layer 7 level is therefore introduced into the CORBA system through this new architecture of the CORBA switch.

When a CORBA request is dropped, two things may happen at the client, depending on the vendor's implementation of the ORB:

1. Retransmission of the CORBA request.
2. An exception is generated in the client.

## 4.2 Network Security

Network Security is very important in a more connected world. Any hacker attack may potentially crash a company's servers, and leave a portion or all of its customers without any service. Any loss of service is unacceptable to most companies. Hacker attacks on large companies have cost them millions of dollars in lost revenues. Companies that have fallen victim to hacker attacks include Microsoft, Amazon.com and Ebay. The most

common forms of attacks are ones where the hackers flood the target servers with a very large number of requests, overwhelming regular traffic. These types of attacks are called denial-of-service attacks. The CORBA Switch only deals with such attacks. Two types of denial-of-service attacks are addressed in this Thesis:

-Flooding by non-CORBA Traffic. This situation occurs when the hacker attempts to overwhelm the operating system of the target server by invoking a very large number of requests. The most common request used is an *icmp* ping.

-Flooding using CORBA requests. This situation occurs when a hacker attempts to overwhelm the CORBA server itself by flooding it with a very large number of CORBA requests.

### 4.2.1 Current Approaches

Current approaches involve using a firewall proxy. CORBA clients send all their CORBA requests to a proxy. The proxy then compares the CORBA request with a filter set by the administrator of the proxy. According to the policy chosen by the administrator, the CORBA request will in turn be forwarded to the target CORBA server. This approach is taken by Orbix's Wonderwall [58] from Iona Technologies, which is commercially available firewall proxy. Wonderwall monitors the IIOP requests and applies access control rules to determine whether to permit or block the request.

### 4.2.2 New CORBA Switch Architecture

To deal with non-CORBA traffic flooding, the CORBA switch is programmed to filter out all non CORBA Traffic. The uplink Filter component that is part of the L2/L3 switching component can be used to filter out all non-CORBA requests. CORBA requests can be identified by the *protocol* field in the TCP header. The non-CORBA requests are simply discarded by the L2/L3 component preventing any flooding to reach the CORBA servers.

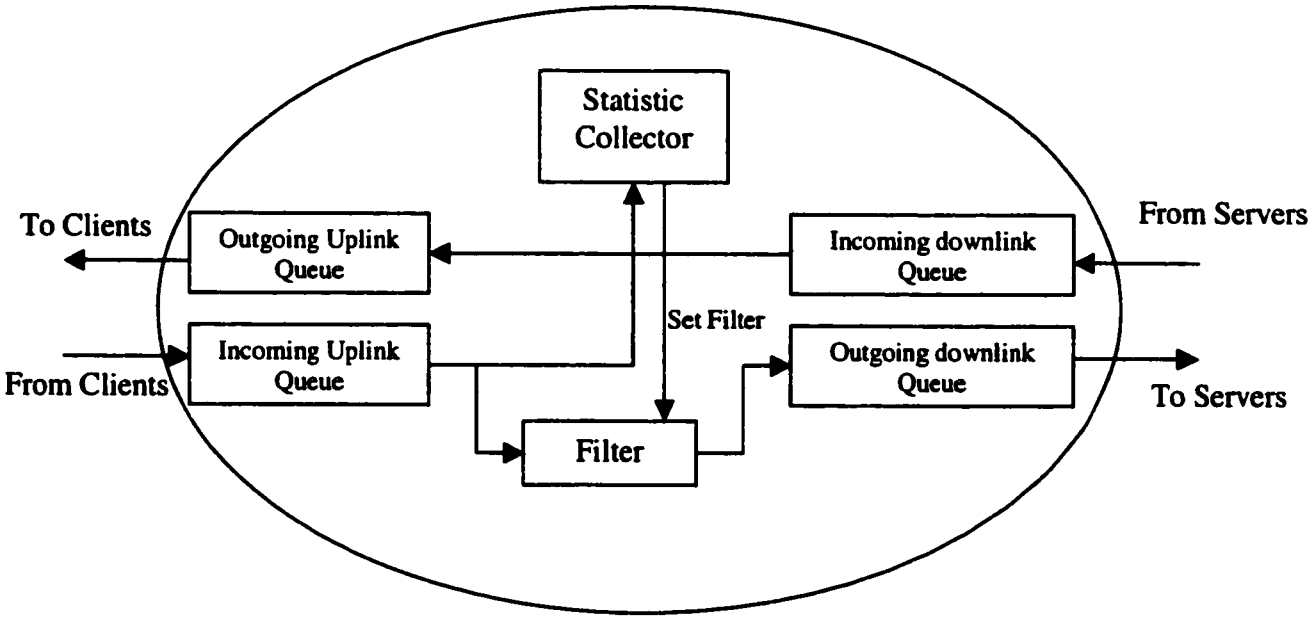


**Figure 4-3 New Architecture for the server group component of CORBA switching component**

There are two approaches that can be taken to prevent flooding of CORBA requests. The first approach is used if the IP address of all the CORBA clients is known beforehand. The CORBA switch can be programmed to only forward CORBA requests that match a list of IP addresses. Any CORBA request not matching the filter will be discarded. The

situation where the IP address of all the clients is known beforehand is rare. The second approach is then used. In the second approach, the CORBA switch keeps track of the average number of requests per client. If a client is seen as being abusive, the IP address of the client is then added to the filter, which will in turn discard all CORBA requests from that client. A client is marked as abusive if the average number of CORBA requests per second sent by that client exceeds a predefined value set by the administrator.

The new architecture for the server group component of the CORBA switching component is shown in Figure 4-3. Two new components are introduced. The first component added is a filter, which discards any CORBA request matching the criteria set by the administrator or the Statistic collector. The second component is the Statistic Collector, which measures the incoming CORBA request rate for each client and determines if a client is abusive. If a client is found to be abusive, the IP address of the client is added to the set of criteria used by the filter. The Statistic Collector can also be used to store to disk the statistics about the traffic through the CORBA switch. This would allow an administrator to better understand the demand profile and adjust the amount and capacity of his CORBA servers.

# 5 An implementation of the CORBA Switch

A prototype implementation of the CORBA switch as described in Chapter 3 was done for the purpose of this thesis. The implementation consists of five parts: A simple CORBA client application, a simple CORBA server application, an Administration Application, the L2/L3 switching component of the CORBA switch, and the CORBA switching component. The implementation of the prototype was designed to accomplish three objectives:

1. Prove the concept of the CORBA switch
2. Determine what amount of delay that will be introduced into the system by the CORBA switch.
3. Determine how the CORBA switch acts under heavy load.

This chapter describes the design and implementation of the prototype of the CORBA switch as well as the tests conducted and their results.

## 5.1 Implementation of the L2/L3 Switching Component

### 5.1.1 Scope of the prototype of the L2/L3 Switching Component

As described in Chapter 3, the L2/L3 switching component behaves in the same manner as commercially available Layer 2 or Layer 3 switches. L2/L3 switches sold today are

mainly implemented in hardware, with some software on an embedded system with a real-time operating system. These systems are closed to the users and cannot be modified. For the CORBA switch, an L2/L3 switch would have to be implemented from scratch as a software component. The L2/L3 switching component is coded as a Layer 2 switch in the prototype of the CORBA switch. The reason of the choice is that the implementation of a Layer 2 switch is straightforward and not as complex as the implementation of a Layer 3 switch or router. The Layer 2 switching component implemented for the CORBA switch prototype consists of two physical ports (uplink and downlink port). It switches packets according to their Ethernet header. The switching is done according the following rules:

- When a packet arrives on the uplink port with the destination Ethernet address of one of the downlink machines, then switch the packet to the downlink port.
- When a broadcast Ethernet packet arrives on the uplink port, the switch the packet to the downlink port.
- When a packet arrives on the downlink port with the destination Ethernet address not being one of the downlink machines, then switch the packet to the uplink port.
- When a broadcast packet arrives on the downlink port where the source Ethernet address is of the downlink machines, then switch the packet to the uplink port.

Components traditional found in an Ethernet switch and that are not implemented in the Layer 2 switching component include:

- Auto-discovery component. An Ethernet switch is able to discover the Ethernet address of all the downlink machines. In the Layer 2 switching component implemented in the prototype, the list of downlink machines is hard coded. Since during all the performance tests to be conducted the Ethernet address of the machines is known ahead of time, this component is not necessary.

- Multiple Downlink ports. To increase the switching capacity, Ethernet switches are designed with multiple downlink ports (8, 16, 24 ports). For this thesis only two downlink machines are used for the different tests not requiring more that one downlink port.

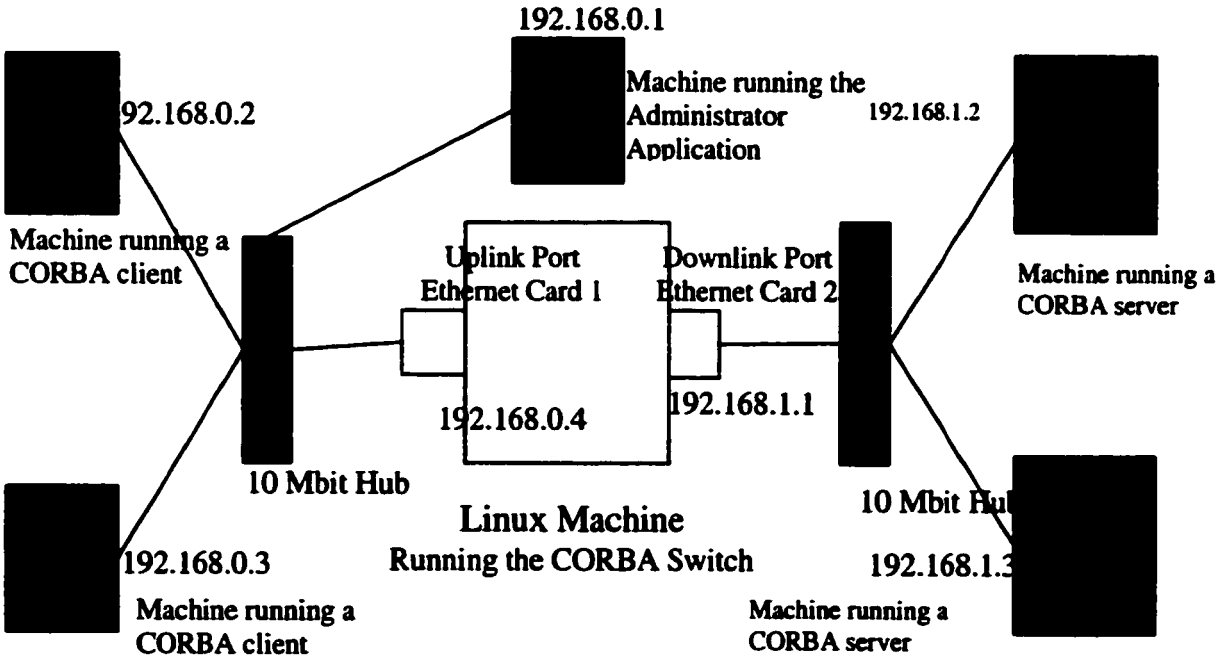


**Figure 5-1 Test setup**

### 5.1.2 Setup of the test environment

Six machines where employed in the test setup. A Pentium-based Linux machine with two Ethernet Cards running the CORBA switch software and five Windows-based machines running the CORBA servers, the CORBA clients and the Administrator Application. The test setup is shown in Figure 5-1. The IP addresses of all the machines are also shown. The system used as the CORBA switch was a Pentium-based system with the Linux operating system (Red Hat 6.1) with two 10 Mbit Ethernet Cards (from Linksys), one representing the uplink port and the other the downlink port.

### 5.1.3 Operational Overview

The L2 switching component contains a table with its Ethernet Addresses as well as all the Ethernet Addresses of the machines connected to it on the downlink port, in our case that would be the machines on which the CORBA servers reside.

The L2 switching component is made up of seven components:

- Four threads [48]
- Two Queues[40][41]
- One filter

The different components are shown in Figure 5-2.

**Thread recPacketUpLink**

The recPacketUpLink thread listens to all packets on the uplink port (Ethernet Card 1). For each packet, the destination address in the Ethernet Header is compared to the table of Ethernet Addresses of the downlink machines. The packet is passed to the downlink queue if and only if:

- The destination address matches the Ethernet Address of one of the downlink machines
- The destination address is the Ethernet Broadcast Address.

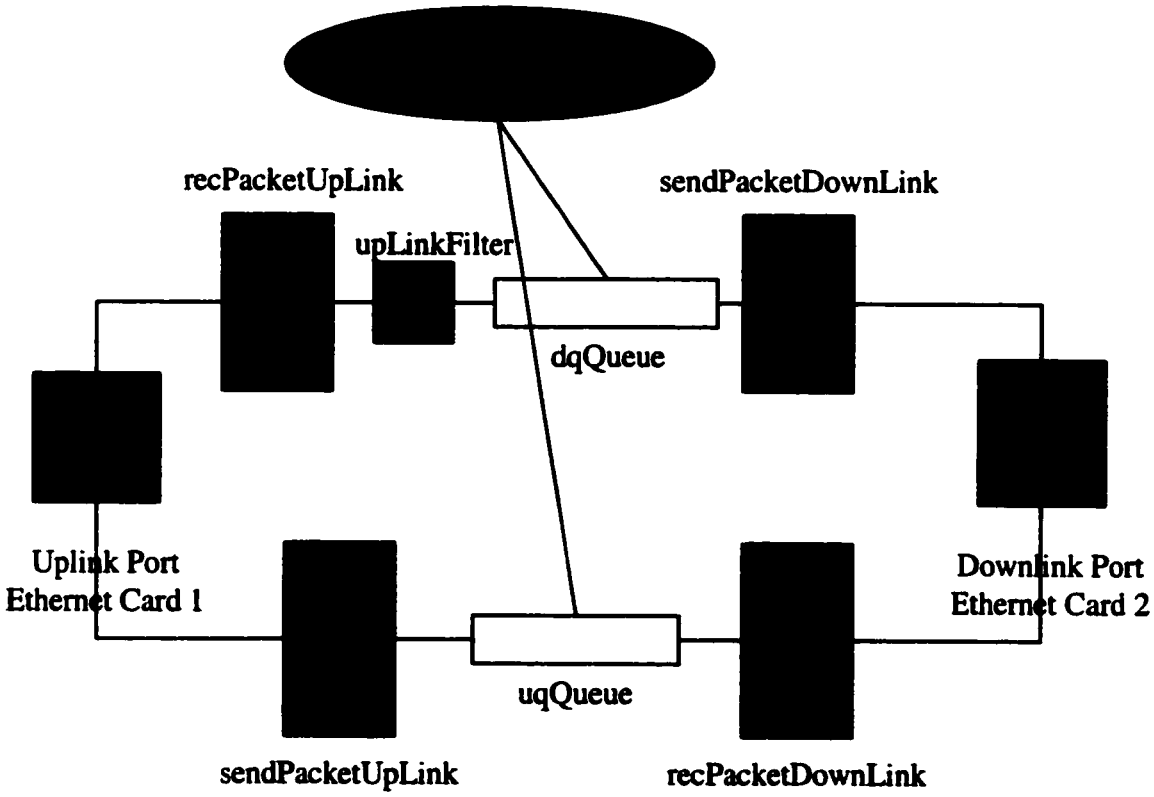


**Figure 5-2 L2 switching component Prototype**

**Thread recPacketDownLink**

The recPacketDownLink thread listens to all packets on the downlink port (Ethernet Card 2). For each packet, the destination and source addresses in the Ethernet Header are compared to the table of Ethernet Address of the downlink machines. The packet is passed to the uplink queue if and only if:

- The destination address does not match the Ethernet Address of one of the downlink machines
- The destination address is the Ethernet Broadcast and the source address matches the Ethernet Addresses of one of the downlink machines.

**Filter upLinkFilter**

The L2 switching component has a table with the IP addresses and TCP ports of all the CORBA servers on the downlink machines. The Administrator Application is used to populate this table. When a packet is received from the uplink port and it matches the filter, it is no longer passed to the downlink queue but instead passed to the CORBA switching component.

**Thread sendPacketUpLink**

The sendPacketUpLink thread is inactive until a packet is placed into the uplink queue by the recPacketDownLink Thread. When a new packet enters the queue, the

sendPacketUpLink thread awakens and transmits the packets on the uplink port (Ethernet Card 1). The origin of the packet may be the recPacketDownLink thread or the CORBA switching component.

**Thread sendPacketDownLink**

The sendPacketDownLink thread is inactive until a packet is placed into the downlink queue by the recPacketUpLink Thread. When a new packet enters the queue, the sendPacketDownLink thread awakens and transmits the packets on the downlink port (Ethernet Card 2). The origin of the packet may be the recPacketUpLink thread or the CORBA switching component.

**Queue dqQueue**

Packets destined to be transmitted on the downlink port (Ethernet Card 2) are placed in the dqQueue. The dqQueue queue is based upon the queue implemented in the ANSI standard CGI Standard Template Library (STL) [40][41]. In the interest of speed, a reference to the memory location of the bytes in the packet is placed in the queue and not the entire packets.

**Queue uqQueue**

Packets destined to be transmitted on the uplink port (Ethernet Card 1) are placed in the uqQueue. The uqQueue queue is based upon the queue implemented in the ANSI standard CGI Standard Template Library (STL) [40][41]. In the interest of speed, a reference to the memory location of the bytes in the packet is placed in the queue and not the entire packets.

### 5.1.4 Implementation of the L2 switching component

The implementation of the L2 switching component was done in C and C++[36] using Redhat Linux 6.2 as the operating system. The compiler used was a the GNU g++ compiler release 1.1.2 . The header file is shown in Figure 5.3. The header file includes the definition of four C functions, which will be called in the four threads, and the LSwichingcomponent Class. The LSwichingComponent holds references to the two queues as well as the table that holds the Ethernet addresses of all the downlink servers. The LSwichingComponent is also responsible for starting and shutting down all threads.

```
void *sendPacketUpLink (void *args);
void *sendPacketDownLink (void *args);

void *recvPacketUpLink (void *args);
void *recvPacketDownLink (void *args);

int filterUpLinkPacket (u_char * buffer, int intLength);


class LSwitchingComponent
{
public:
 /* Socket IDs */
 int m_intSockInDownLink;
 int m_intSockInUpLink;
 int m_intSockOutDownLink;
 int m_intSockOutUpLink;
 char *m_strDownLinkDevice;
 char *m_strUpLinkDevice;
 pthread_t m_thrRecvUpLink;
 pthread_t m_thrRecvDownLink;
 pthread_t m_thrSendUpLink;
 pthread_t m_thrSendDownLink;

 STDQUEUE *m_dqDownLinkOutputQueue;
 STDQUEUE *m_dqUpLinkProcessQueue;
 MAPSTRINGTOMAC *m_mapMacAddressesS;
 MAPSTRINGTOMAC *m_mapMacAddressesD;

 LSwitchingComponent ();
 virtual ~ LSwitchingComponent ();
 virtual void OpenForRecvDownLink (char *strDevice);
 virtual void OpenForRecvUpLink (char *strDevice);
 virtual void OpenForSendDownLink (char *strDevice);
 virtual void OpenForSendUpLink (char *strDevice);

 virtual void Run ();
 virtual void ShutDown ();
};
```

**Figure 5-3 L2 switching component header file**

## 5.2 Implementation of the CORBA Switching Component

### 5.2.1 Challenges for the implementation of the prototype of the CORBA Switching Component

The implementation of the prototype of the CORBA switching component poses one major challenge. The challenge deals with the fact that the CORBA switch needs to accept TCP connections, from the CORBA clients, where the destination of the TCP connection is not the CORBA switch but one of the CORBA servers. One of the requirements of the CORBA switch is that it accepts and sets up this connection "on behalf" of the CORBA servers. The CORBA client must not be aware that the connection setup is not with the CORBA server but with the CORBA switch. The challenge is that the standard TCP stack cannot accept TCP connections not destined to the local host. To accomplish this requirement and overcome the obstacle, the TCP stack embedded in Linux must therefore be replaced with a new TCP stack. This new TCP stack will be able to accept TCP connections where the local host is not the destination of the TCP connection. This new TCP stack would also identify each connection not by the TCP port but by the TCP port and the destination IP address. In a standard TCP stack, the destination IP address is always the local host and therefore the TCP port is sufficient to identify each TCP connection. While in the CORBA switch, the destination IP address is the IP of one of the CORBA servers and therefore the TCP stack must therefore keep track of it. The implementation of this new TCP stack is too labor intensive and not necessary for this thesis. Using the following approach, the implementation of a new TCP

stack is not necessary, and the concept of the CORBA switch can therefore be quickly verified:

The approach involves modifying the IP address of the CORBA server in the CORBA server IOR address published to the CORBA clients. In a typical CORBA distributed system, there are two ways a CORBA client may obtain a reference to a CORBA server:

1. From the CORBA Naming or Trading Service;
2. From a file.

In the test environment for the CORBA switch, the reference to the CORBA server is obtained by the CORBA clients from a file. When the CORBA server is launched, it saves its CORBA reference in a file. The file is then copied onto the hosts where the CORBA clients reside. When the CORBA client is launched, it opens the file and reads the CORBA reference. To overcome the need for a new TCP stack, the object reference saved in the file was manually modified. The modification involves changing the IP address in the reference to indicate that the IP address of the CORBA server is the same as the CORBA switch. In this situation, the CORBA client is fooled to believe that the CORBA switch resides on the same host as the CORBA switch. In this manner, when CORBA client setups its TCP connection, it will attempt to set it up with the CORBA switch. The IP destination address in the TCP control messages will be the CORBA switch, allowing the reuse of the standard TCP stack by the CORBA switch.

### 5.2.2 Scope of the prototype of the CORBA switching component

For the purpose of this thesis, a partial implementation of the CORBA switching component was done. The components of the CORBA switching component implemented for the prototype include:

- Interface with Administrator Application
- Interface with L2 switching component
- Server groups, including:
    - Fault-tolerance
    - Request load balancing
    - Client load balancing
    - Versioning
- Manager

Components not implemented for the prototype include:

- MiniIFR
- Replication Manager
- Server group:
    - Smart load balancing

### 5.2.3 Implementation of the prototype CORBA switching component

```
module CORBASwitch
{

        typedef sequence<string> strings;


        interface csManager
        {
                strings listServerGroups();
                short addNewServerGroup (in string strServerGroupName, in short
intPolicyType);
                short removeServerGroup (in short intServerGroupId);
                short addServer(in short intServerGroupId , in string strServerName, in
string strIpAddress , in short intPortNumber);
                strings listServers(in short intServerGroupId);
                short getServerGroupPolicyType(in short intServerGroupId);
                short changeVersion(in short intServerGroupId , in short intServerId);
                short getCurrentVersion(in short intServerGroupId);
                short runServerGroup (in short intServerGroupId);
                short getServerGroupState(in short intServerGroupId);

        };
};
```

**Figure 5-4 IDL interface definition of the CORBA Switch**

#### 5.2.3.1 Manager Component and the Interface with the Administrator Application

The communication between the CORBA switch and the Administrator Application is implemented using CORBA. By implementing the Manager component as a CORBA object, the Interface with the Administrator Application becomes the IDL interface of the Manager Object. The functions defined in the IDL interface of the Manager Object will be accessible by the Administrator Application. Figure 5-3 shows the IDL of the Manager Object implemented in the prototype. The CORBA ORB used for the CORBA switching

component is Mico [25]. Mico is a freely available and fully compliant implementation of the CORBA standard. The version of Mico used for the CORBA switch is version 2.3.3, which fully implements CORBA 2.3. Mico is available for several different operating systems including Linux and Microsoft Windows The IDL interface was written as an Interface called *csManager* defined in a module called *CORBASwitch*. The following functions are defined:

**strings listServerGroups();**

This function will return a list of all the server groups running in the CORBA switch.

**short addNewServerGroup (in string strServerGroupName, in short intPolicyType);**

This function creates a new server group. A name for the server group is passed as parameter as well as the policy type. The policy type may be one of the following:

- Fault-tolerance
- Request load balancing
- Client load balancing
- Versioning

An Id is returned uniquely identifying the server group.

**SHORT REMOVESERVERGROUP (IN SHORT INTSERVERGROUPID);**

This function removes the server group identified by the parameter *intServerGroupID* from the CORBA switch

**short addServer(in short intServerGroupId , in string strServerName, in string strIpAddress , in short intPortNumber);**

This function adds a CORBA server to a server group. The server group is identified by the parameter *intServerGroupID*. The CORBA server to be added is given a name using the parameter *strServerName*. The IP address and the TCP port number of the CORBA server are passed to the CORBA switch through the parameters *strIPAddress* and *intPortNumber*.

**STRINGS LISTSERVERS(IN SHORT INTSERVERGROUPID);**

This function returns the list of CORBA servers in the server group identified by the Id *intServerGroupId*.

**short getServerGroupPolicyType(in short intServerGroupId);**

This function returns the Policy Type of the server group identified by the Id *intServerGroupId*.

**SHORT CHANGEVERSION(IN SHORT INTSERVERGROUPID , IN SHORT INTSERVERID);**

This function is used with a server group of policy type versioning. Using the function the Administrator may change the CORBA server currently designated as current version to a new CORBA server.

**SHORT GETCURRENTVERSION(IN SHORT INTSERVERGROUPID);**

This function is used with a server group of policy type versioning. This function returns the Id identifying the CORBA server which is the currently designated as current version.

**SHORT RUNSERVERGROUP (IN SHORT INTSERVERGROUPID);**

Before a server group becomes active, the Administrator must first activate it by calling this function and indicating the server group using the parameter *intServerGroupId.*

**SHORT GETSERVERGROUPSTATE(IN SHORT INTSERVERGROUPID);**

This function returns whether or not a server group is currently running. The server group is identified using the parameter *intServerGroupId.*

#### 5.2.3.2 Implementation of the server groups

There is one base class called *CcsServerGroup* from which the different server group classes inherit. This class contains properties in common to all of the server groups. Having a base class and having the server group class inherit from it allow the CORBA switch to easily be upgraded with new policies. For the prototype there are five classes that inherit from *CcsServerGroup*, one for each of the different policy types. The UML [43] diagram is shown in Fig 5-6

```
class CcsServerGroup
{
public:
        char* m_strServerGroupName;
        int m_intInputSocket;
        int m_intPolicyType;
        int m_intState; /*ON or OFF*/

        CcsServerGroup (char* strServerName, int intPolicyType);
        virtual ~ CcsServerGroup ();

        char* getServerGroupName();
        int getInputSocket();
        short setInputSocket(int intInputSocket);
        int getPolicyType();
        virtual int shutDown();
        virtual short run();
        virtual CORBASwitch::strings* listServers();
        int getState();
        int setState(int intState);

};
```

**Figure 5-5 Class definition of the CcsServerGroup Class**

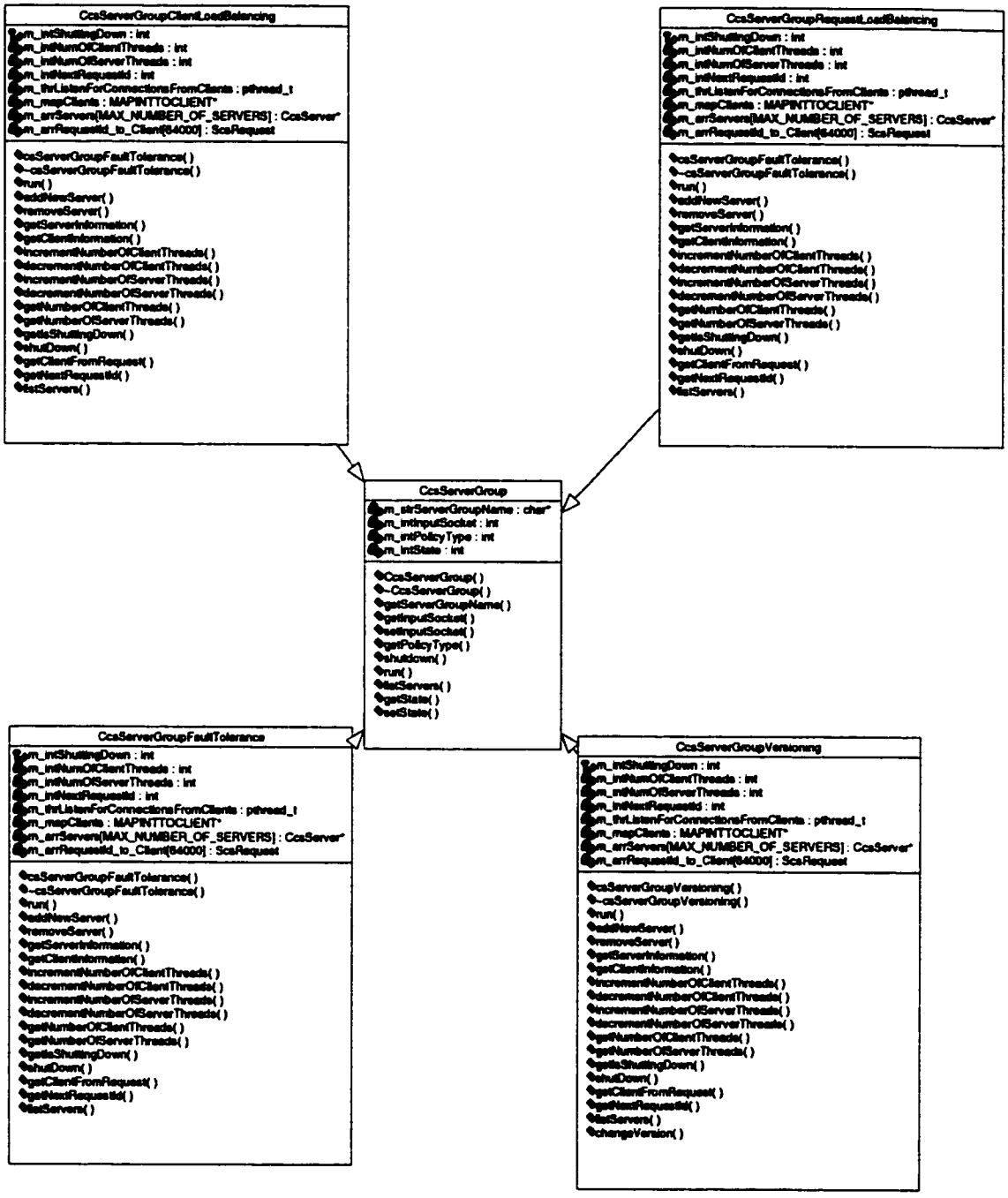


**Figure 5-6 UML Diagram of the CORBA Switch**

## 5.3 Implementation of the Administrator Application

Using the Administrator Application, the Administrator of the distributed CORBA application can configure the CORBA switches in his system as well as receiving notification of failures in his system. Ideally the Administrator Application would be implemented in Java [46] and placed on the CORBA switch itself. The Administrator Application would use CORBA to communicate with the CORBA switch. A web server [47] would also be running on the CORBA switch. The Administrator would only need to set the http address of his browser to the IP address of the CORBA switch and the Administration Application would be automatically loaded into the web browser. This would allow the Administrator Application to run on any machine the Administrator desires.

For the purpose of thesis the Administrator Application was implemented in Java 1.2 [53] using the Borland Jbuilder 3 Development Suite of Applications [54]. The CORBA ORB used was JacORB. A java implementation of the Mico ORB is not available. JacORB is a CORBA ORB written in Java and is compliant with version 2.3 of CORBA standard. JacORB is freely available over the Internet. The version used in this thesis is version 1.2.3.

The Administrator Application implemented for this thesis allows an administrator to do the following:

- Create a server group with one of the following policies:
    - Fault-tolerance

- Request load balancing
- Client load balancing
- Versioning

- Add a CORBA server to a server group,
- Activate a server group,
- Change CORBA server version for the versioning policy.

The features of the Administrator Application not implemented are:

- Create a server group of type smart load balancing,
- Receive asynchronous notification of a CORBA server failure
- Upload the IDL interface definition of the CORBA server to the miniIFR of the CORBA switch

A few screen captures of the implementation of the Administrator Application are shown in Figures 5-6, 5-7, 5-8, 5-9 and 5-10

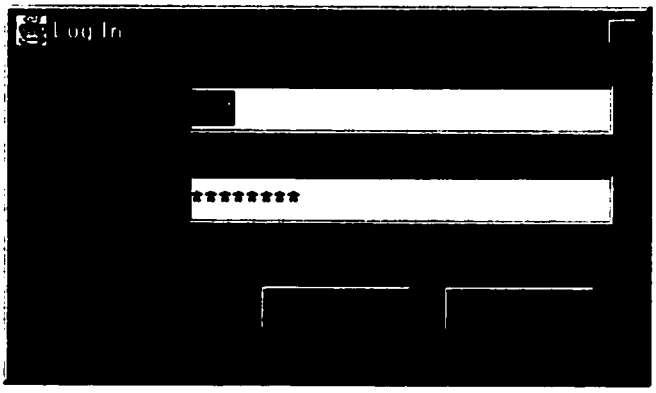


**Figure 5-7 Log in window**

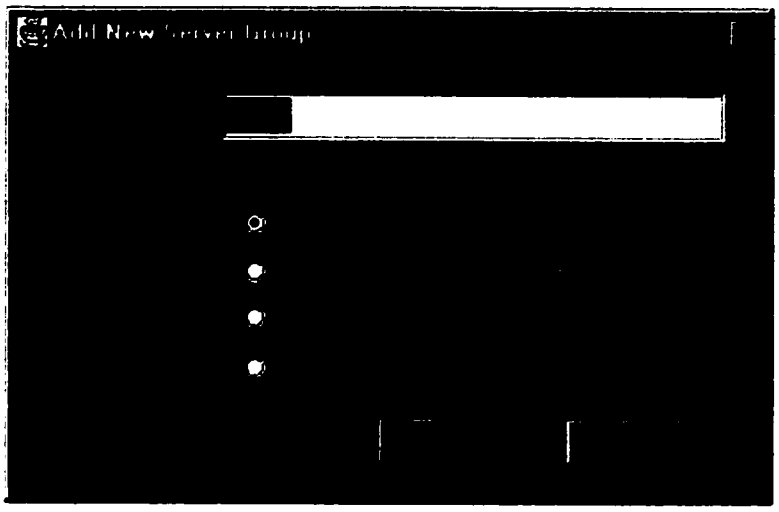

**Figure 5-8 Add New Server Group Window**

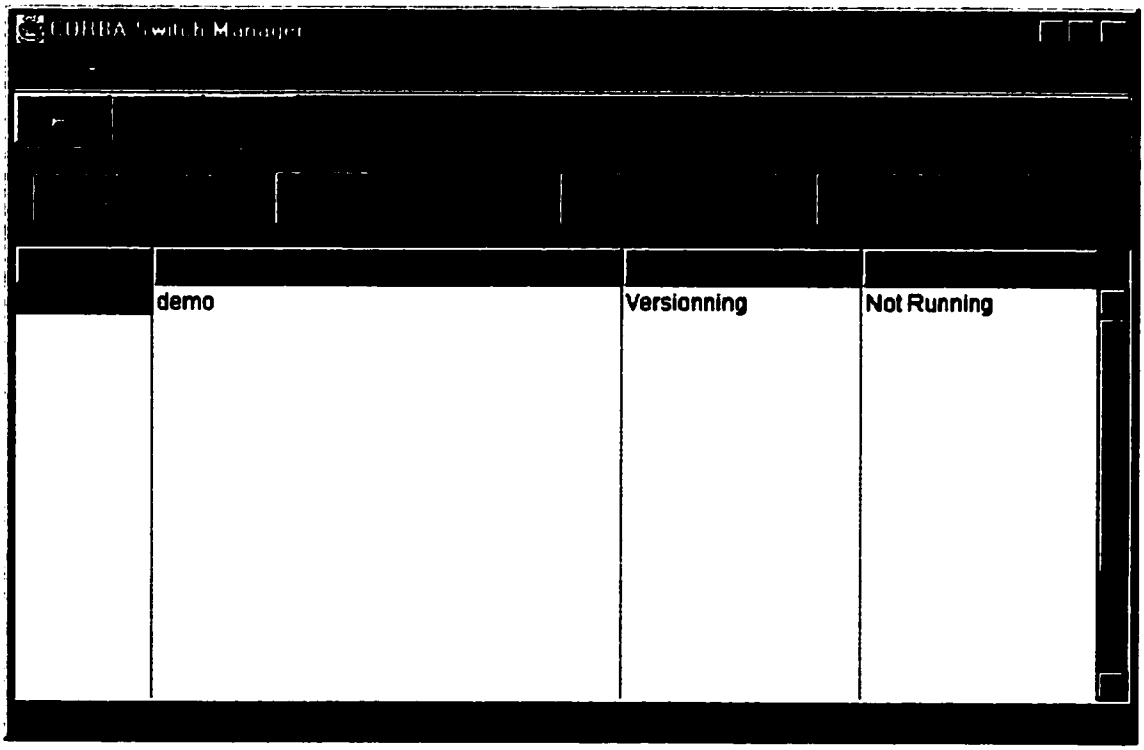


**Figure 5-9 Main window**

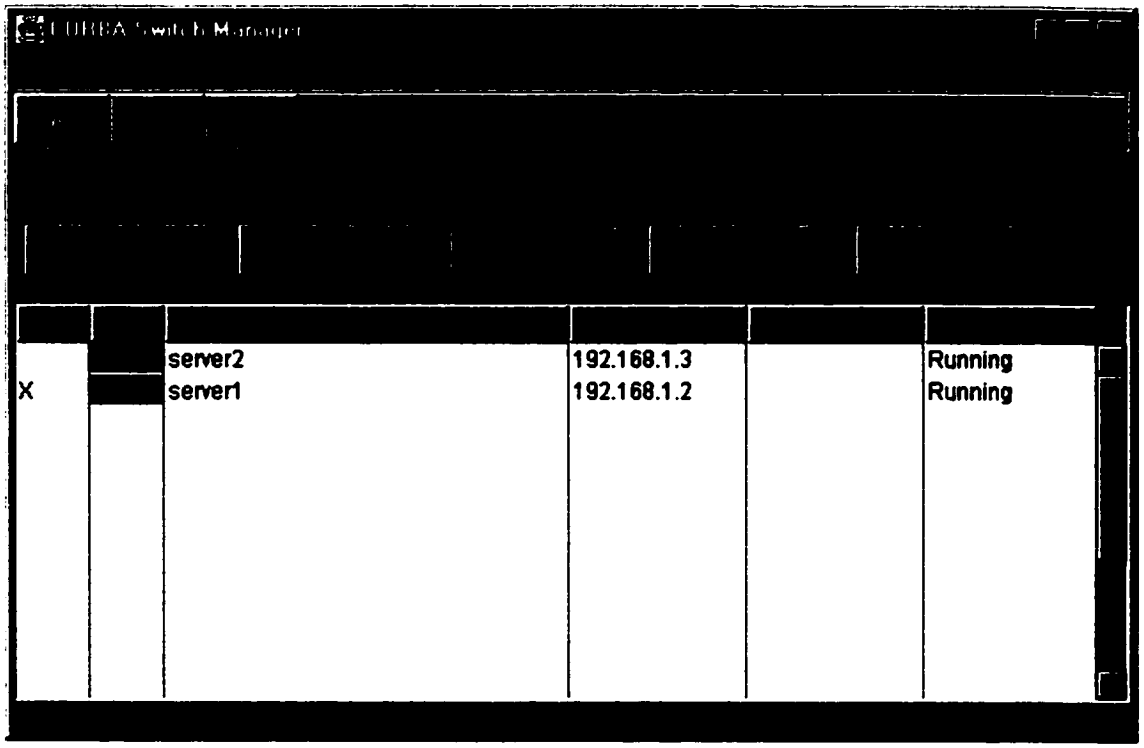


**Figure 5-10 Versioning Policy Window**

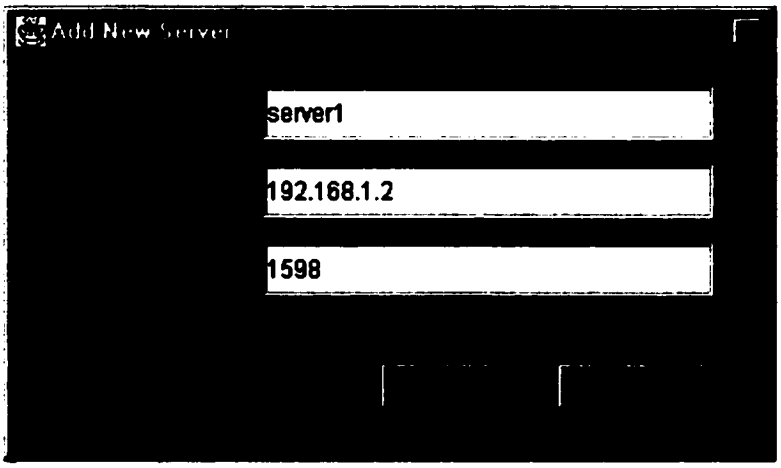


**Figure 5-11 Add New Server Window**

The Administrator Application communicates with the CORBA switch using CORBA. All the CORBA functionality is placed inside one java class called *csInterface*. Through this class, the Administrator Application communicates with the CORBA switch. The definition of the class is shown in Figure 5-12.

csManager

Class csInterface
java.lang.Object
|
+--**csManager.csInterface**

---

public class **csInterface**
extends java.lang.Object

---

| | |
|---|---|
| (package private) CORBASwitch.csManager | **m_Manager** |
| (package private) org.omg.CORBA.ORB | **orb** |

| |
|---|
| **csInterface**() |

| | |
|---|---|
| boolean | **addNewServer**(int intServiceId, java.lang.String strServerName, java.lang.String strIPAddress, int intPortNumber) |
| int | **addNewServerGroup**(java.lang.String strServiceName, int intServiceType) |
| short | **changeVersion**(int intServiceId, int intServerId) |
| boolean | **connect**(java.lang.String strUserName, java.lang.String strPassword) |
| short | **getCurrentVersion**(int intServiceId) |

| | |
|---|---|
| short | getServerGroupState(int intServiceId) |
| short | getServiceType(int intServiceId) |
| java.lang.String[][] | listServerGroups() |
| java.lang.String[][] | listServers(int intServiceId) |
| boolean | removeServerGroup(int intServiceId) |
| short | runServerGroup(int intServiceId) |

| Methods inherited from class java.lang.Object |
|---|
| , clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait |

**Figure 5-12 CsInterface Class Definition**

# 6 Implementation of the test CORBA clients and servers

The CORBA clients and servers implemented for the purpose of this thesis were intentionally designed to be as simple as possible. The reasoning behind the simplicity of the clients and servers is that during performance analysis tests of the CORBA switch, the delay to be measured is the one caused by the CORBA switch. Reducing the delay introduced by the CORBA clients and CORBA servers would help measure the delay introduced by the CORBA switch. The CORBA clients and CORBA servers are implemented in C++ and using the Mico ORB.

The CORBA server only exports one function called *doNothing* which simply displays a message on the screen which includes the number *intNumber* passed as parameter. The CORBA server IDL interface is shown in Figure 5-11.

```
interface Bank {
        void doNothing(in short intNumber);
};
```

**Figure 6-1 IDL interface of the test CORBA Server used**

There are two types of CORBA clients.

- One simply calls the function *doNothing* of the CORBA server as many times as possible. These CORBA clients will be used to generate traffic through the CORBA switch.

- The other CORBA client measures the time taken for each CORBA requests and saves it into a file.

## 6.1 Test results for the CORBA Switch

### 6.1.1 Proof of Concept Tests

The first series of test runs were done to prove that the CORBA switch works. There were four tests run, one for each type of policy.

#### 6.1.1.1 Versioning

For this test, two CORBA servers were employed, serverA and serverB as well as one client. Using the Administrator Application, the CORBA switch was setup by creating a server group of policy versioning and by adding the two servers to the server group. The CORBA server serverA was set as "current version" using the Administrator Application. When the client was run, the output messages indicating that the CORBA requests were executed appeared on the machine running serverA. Using the Administrator Application, the "current version" was set to serverB while the CORBA client was still running. Almost immediately after this occurred, the output message indicating that the CORBA requests were executed started appearing on the machine running serverB. ServerA was no longer sending out the messages. This was proof that the CORBA switch had versioned serverA to serverB without knowledge or interruption of the CORBA client.

#### 6.1.1.2 Request Load Balancing

For this test, two CORBA servers were employed, serverA and serverB as well as one client. Using the Administrator Application the CORBA switch was setup by creating a server group of policy request load balancing and by adding the two servers to the server group. The CORBA client was coded to call the function *doNothing* of the CORBA server and to pass a continuously incremented integer to the server. When the client was run, the output messages indicating that the CORBA requests were executed appeared on the machine running serverA and serverB. On serverA, the output messages included only odd numbers while output message on serverB showed only even numbers. Request 1 was sent to serverA by the CORBA switch, then request 2 was sent to server B, request 3 to server A and so on. This was proof that the CORBA switch was load balancing the requests onto serverA and serverB without knowledge or interruption of the CORBA client.

#### 6.1.1.3 Client Load Balancing

For this test, two CORBA servers were employed, serverA and serverB as well as two clients. Using the Administrator Application the CORBA switch was setup by creating a server group of policy client load balancing and by adding the two servers to the server group. When the first client was run, the output messages indicating that the CORBA requests were executed appeared on the machine running serverA. Then the second client was started and then the output messages started to appear on the machine running

serverB. This was proof that the CORBA switch was load balancing the clients onto serverA and serverB without knowledge of the CORBA clients.

#### 6.1.1.4 Fault Tolerance

For this test, two CORBA servers were employed, serverA and serverB as well as one client. Using the Administrator Application the CORBA switch was setup by creating a server group of policy fault tolerance and by adding the two servers to the server group. The CORBA client was coded to call the function *doNothing* of the CORBA server and to pass a continuously incremented integer to the server. When the client was run, the output messages indicating that the CORBA requests were executed appeared on the machine running serverA and serverB. The number shown was the same on both serverA and serverB. This indicated that both servers executed the same requests. Server B was then killed, leaving serverA running. The requests were now only executed by serverA. The client was unaware of the failure of serverB. When serverA was then killed, the client then reported an exception. The test was redone but failing serverA first, then serverB, and the same results were obtained. This was proof that the CORBA switch was executing the request on serverA and serverB without knowledge of the CORBA clients allowing the system to continue operating even faced with the failure of one of the CORBA servers.

### 6.1.2 Delay Introduced by the CORBA Switch

The second series of tests were run to determine the amount of delay introduced by the CORBA switch. The following five tests were run:

- CORBA switch absent,
- CORBA switch with fault tolerance policy,
- CORBA switch with versioning policy,
- CORBA switch with request load balancing policy,
- CORBA switch with client load balancing policy.

#### 6.1.2.1 CORBA Switch absent

The first test was run with the one CORBA client and one CORBA server running without a CORBA switch. One hundred requests were sent, and the delay was measured and is displayed in Figure 6-1. The average time taken to transmit, execute and receive a CORBA request is 4.68 ms. The standard deviation of the average time taken to transmit, execute and receive a CORBA request is 1.5 ms

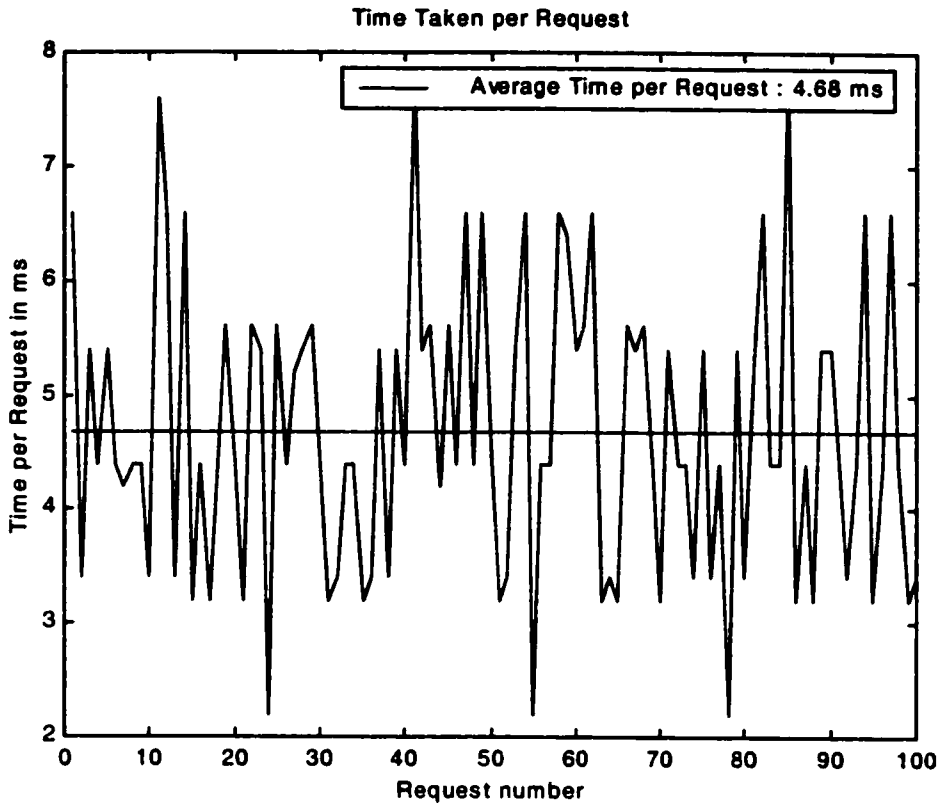


**Figure 6-2 Time taken per request with no CORBA Switch**

### 6.1.2.2 CORBA Switch with Fault Tolerance policy

The second test was run with the one CORBA client and two servers running with a CORBA switch using the fault tolerance policy. One hundred requests were sent, and the delay was measured and is displayed in Figure 6-2. The average time taken to transmit, execute and receive a CORBA request is 7.068 ms. The standard deviation of the average time taken to transmit, execute and receive a CORBA request is 1.7 ms. The average delay introduced is therefore 7.068 – 4.68 = 2.388 ms. From the Figure 6-2, it is clear that a failure of the server did not cause any additional delay to the CORBA requests sent when the failure occurred.

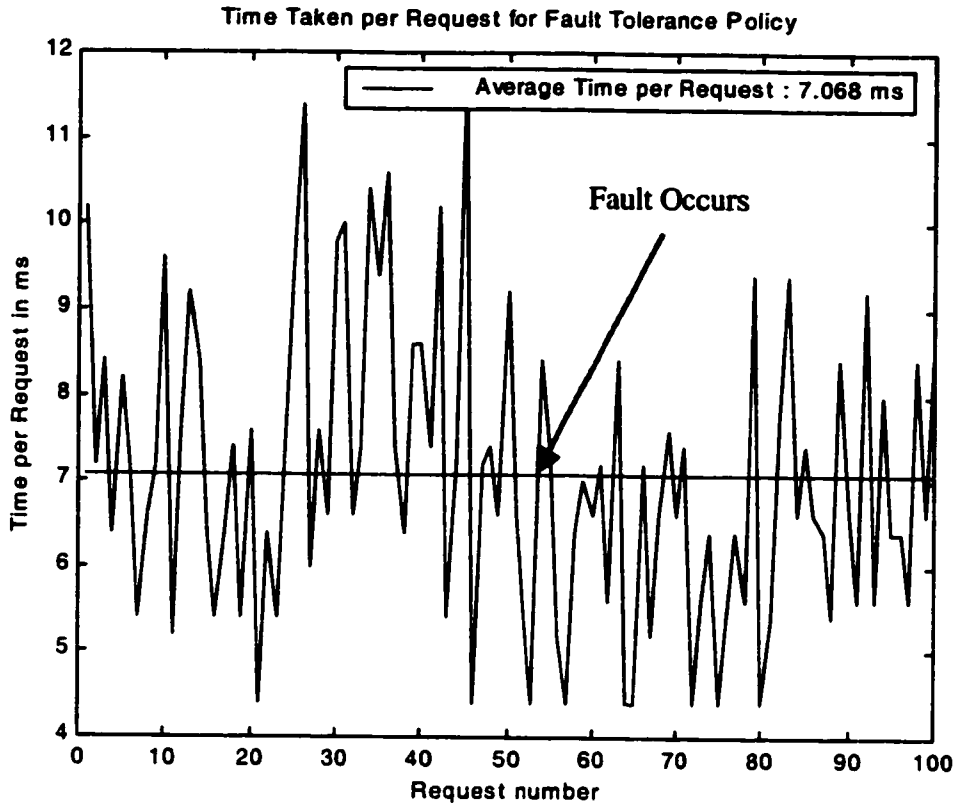


**Figure 6-3 Time Taken per request for Fault Tolerance Policy**

### 6.1.2.3 CORBA Switch with Versioning policy

The third test was run with the one CORBA client and two servers running with a CORBA switch using the versioning policy. One hundred requests were sent, and the delay was measured and is displayed in Figure 6-3. The average time taken to transmit, execute and receive a CORBA request is 6.414 ms. The standard deviation of the average time taken to transmit, execute and receive a CORBA request is 1.6 ms. The average delay introduced is therefore 6.414 – 4.68 = 1.734 ms. From the Figure 6-3, it is clear that the versioning of the server did not cause any additional delay to the CORBA requests sent when the versioning occurred.

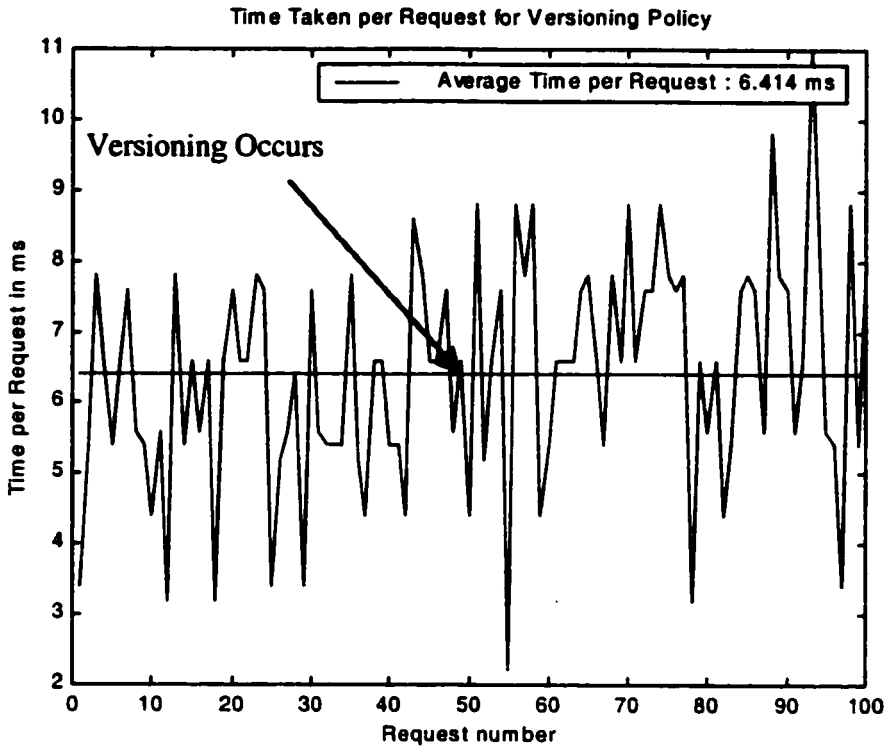


**Figure 6-4 Time taken per request for Versioning policy**

#### 6.1.2.4 CORBA Switch with Request Load Balancing policy

The fourth test was run with the two CORBA clients and two servers running with a CORBA switch using the request load balancing policy. One hundred requests were sent, and the delay was measured and is displayed in Figure 6-4. The average time taken to transmit, execute and receive a CORBA request is 6.64 ms. The standard deviation of the average time taken to transmit, execute and receive a CORBA request is 1.5 ms. The average delay introduced is therefore 6.64 – 4.68 = 1.96 ms

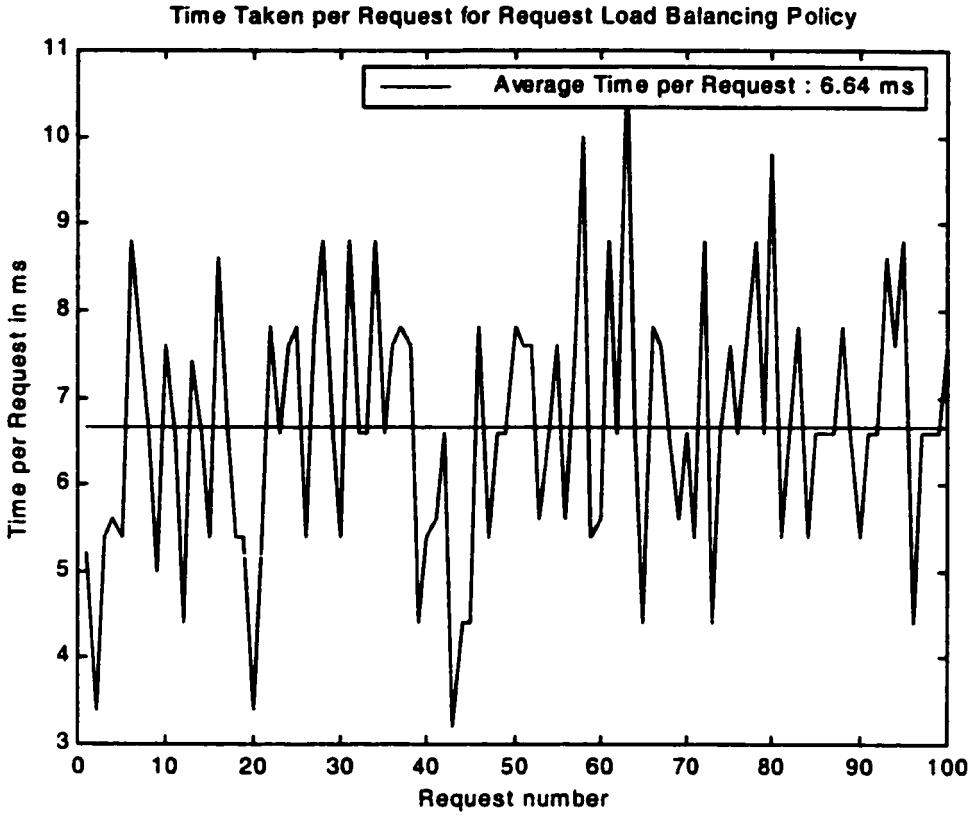


**Figure 6-5 Time taken per request for Request Load Balancing Policy**

### 6.1.2.5 CORBA Switch with Client Load Balancing policy

The fifth test was run with the two CORBA clients and two servers running with a CORBA switch using the client load balancing policy. One hundred requests were sent, and the delay was measured and is displayed in Figure 6-5. The average time taken to transmit, execute and receive a CORBA request is 6.318 ms. The standard deviation of the average time taken to transmit, execute and receive a CORBA request is 1.7 ms The delay introduced is therefore 6.318 – 4.68 = 1.638 ms.

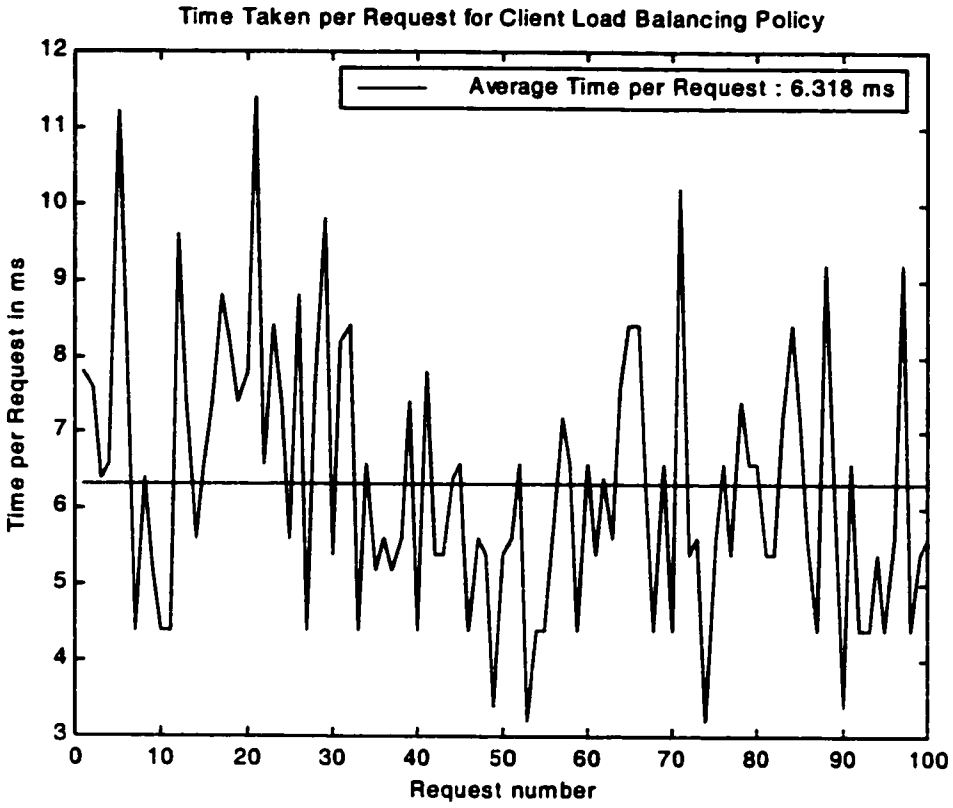


**Figure 6-6 Time taken per request for Client Load Balancing policy**

#### 6.1.2.6 Proof of Concept and Delay test Conclusions

The results observed during this series of tests prove that the CORBA switch works for the four policies implemented. The fifth policy, smart load balancing, is a combination of load balancing and fault tolerance policy depending on the CORBA function called and therefore would work because its sub-components are proven. The delay added is negligible in absolute terms. 2 ms is negligible because in some instances the delay introduced by the network would be in the range for hundreds of milliseconds to several seconds. The delay added in percentage terms is high because the network delay in our

test environment is negligible because there are no switches or routers along the way (except for the CORBA switch) and there is no other traffic on the Ethernet links. The execution time of the CORBA server is minimal by design because the function called does nothing but display a message on screen.

Another point to be emphasized is that the CORBA switch was implemented totally in software on a non real-time operating system. It is conceivable that an implementation done in software and hardware on a real time operating system would yield even lower delays.

| **Policy Type** | **Average Time per Request (ms)** | **Average added time per Request (ms)** | **Average added Time per Request in %** |
|---|---|---|---|
| Fault tolerance | 7.068 | 2.388 | 51 |
| Versioning | 6.414 | 1.734 | 37 |
| Request load balancing | 6.64 | 1.96 | 41 |
| Client load balancing | 6.318 | 1.638 | 35 |

**Table 1 Average Times obtained during experiments**

### 6.1.3 Stress test

This final series of tests is intended to study the delay introduced by the CORBA switch under a heavy load. In this test setup, three CORBA clients running on three different machines were used as well as two CORBA servers. Two of the clients were used to generate as many CORBA requests as possible and are termed traffic-generating clients. The third client was used to measure the time taken to transmit, execute and receive each CORBA request and is termed the delay-measuring client. Under a heavy load, the execution time of the CORBA servers may increase and affect the result. To prevent this from happening, the policy used will be client load balancing. First a traffic-generating client was launched, and the CORBA switch then sent all requests from the client to serverA. Then the delay-measuring client was launched and its requests were sent to serverB. Finally the other traffic-generating client was launched, and its requests were sent to serverA. ServerA was flooded with requests, while serverB operated normally and received requests only from the delay measuring.

The results are shown in Fig 6-6. The two traffic-generating clients generated a total of 258 CORBA requests per second. The average delay obtained was 12.49 ms. The delay added from the CORBA switch is therefore 12.49 – 4.68 = 7.81 ms. In absolute terms the delay is still negligible compared to other delays found in a typical system such as propagation delay. A better implementation of the CORBA switch would improve this already negligible delay.

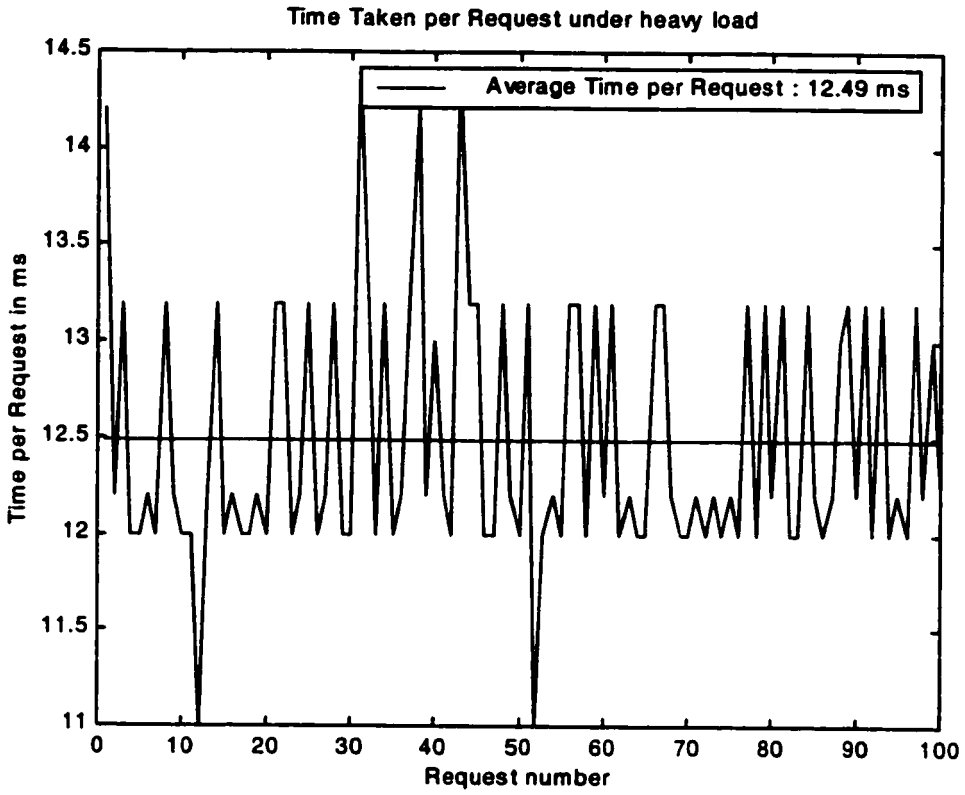


**Figure 6-7 Time taken per request under heavy load**

# 7 Conclusions and Future Research

The CORBA standard was designed to help deal with the problem of software development for heterogeneous networks. But the CORBA standard did not deal with the problem of reliability. Using the innovative concept of CORBA switching presented in this thesis, the current CORBA standard can be enhanced to become more reliable without any modifications to the standard itself, or existing CORBA clients and servers. CORBA switching could help accelerate the adoption of CORBA by businesses around the world.

## 7.1 Concepts Addressed in this Thesis

CORBA switching increases the reliability of a distributed CORBA application by implementing the following concepts: fault-tolerance, versioning, load balancing, quality of service and network security. Some of these concepts are currently available for CORBA but fall short when compared to CORBA switching. The introduction of a CORBA switch into a system running a distributed CORBA application would result in a more reliable and robust system without almost any interruption of service. The system enhanced by the CORBA switch would be able to handle server failures without causing any loss of service. The servers may now undergo a software or hardware upgrade without causing an interruption of service. And finally the system would be better able to

handle larger amounts of requests from the clients. CORBA switching is not suited for every situation. The limitations are as follows:

- CORBA switching can only version complete servers and not individual objects,
- CORBA switching does not allow for inter-server communication,
- Ownership of the network is required so that the physical introduction of the CORBA switch can be made.

This thesis presents a prototype of a CORBA switching, which implements the concepts of fault-tolerance, request and client load-balancing and versioning. Test CORBA clients and servers were developed and used in a series of tests using the CORBA switch. Through this series of tests, the concept of CORBA switching was proven. The series of tests demonstrated that the delay introduced by the CORBA switch into the system is minimal. The CORBA switch was shown to be able to handle large amounts of traffic and still not introduce large delays.

## 7.2 Contributions of this Thesis

Overall, this thesis examines the problem of high reliability CORBA in a distributed environment, resulting in the new concept of CORBA switching. A design of a CORBA switch is proposed. A prototype was implemented and the results of the effect of the prototype on a CORBA distributed system are presented.

This thesis made several significant research contributions, which can be summarized as follows:

1. An innovative concept of CORBA switching is presented for the first time. A new methodology for CORBA fault tolerance, versioning and load balancing is introduced. A new approach to CORBA quality of service and network security is described.
2. A fully functional prototype of a CORBA switch is constructed. A test application is produced and tested. The impact of CORBA switching on the performance of a CORBA distributed application is explored.

## 7.3 Future Research

The following work should be conducted to continue the development of the concept of CORBA switching:

- Development of a fully functional implementation of the CORBA switch in a more efficient fashion. Tests should then be conducted on real CORBA applications to study the real life effect of the CORBA switch
- Investigate the introduction of the concepts of quality of service and network security to the CORBA switch.
- Investigate the application of CORBA switching onto protocols other than IIOP, such as HTTP.

# 8 Bibliography

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specifications Revision 2.2*, OMG, Framingham, MA, 1999.

[2] ORB Portability Joint Submission to Object Management Group (OMG) , May 20 1997

[3] Orbix Programmer's Guide (Version 2.3), IONA Technologies PLC, Dublin, Ireland, 1997.

[4] Orbix Programmer's Reference (Version 2.3), IONA Technologies PLC, Dublin, Ireland, 1997.

[5] S. Vinoski and M. Henning, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1998.

[6] W. Ruh, T. Herron, P. Klinker, IIOP Complete, Understanding CORBA and Middleware Interoperability, Addison Wesley, 1998

[7] Defense Advanced Research Projects Agency, RFC 791 *Internet Protocol*, September 1981

[8] Defense Advanced Research Projects Agency, RFC 793 *Transmission Control Protocol*, September 1981

[9] Object Management Group, *Interoperable Naming Service Specification*, 1999

[10] Object Management Group, *Trading Object Service Specification*, 2000

[11] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The Isis System Manual*. Dept of Computer Science, Cornell University, September 1990.

[12] C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 1996.

[13] R. Van Renesse, K. Birman, and R. Cooper. *The HORUS system*. Technical report, University of Cornell (NY), 1993.

[14] M.G. Hayden, *The Ensemble System*, Ph.D thesis, Cornell University, Ithaca, NY, 1998.

[15] A. Vaysburd, K. Birman, "Building Reliable Adaptive Distributed Objects with the Maestro Tools, in Proceeding of *Workshop on Dependable Distributed Object Systems*.

[16] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311-342, November 1995.

[17] Y. Amir, D. Dolev, S. Kramer, and D. Malki. *Transis: a communication sub-system for high availability*. In Proceedings of the 22nd International Symposium on Fault Tolerant Computing Systems, 1992.

[18] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, University of Zurich (Switzerland), February 1995.

[19] S. Landis, S. Maffeism "Building Reliable Distributed Systems with CORBA", *Theory and Practice of Object Systems*, vol 3, no 1, 1997.

[20] Object Management Group, *Fault Tolerant CORBA*, OMG, Framingham, MA, December 1999

[21] Network Working Group, *Domain Name System Structure and Delegation*, March 1994

[22] Gerald Brose, *Reflection in CORBA, Java and JacORB*, Procs. JIT 1998, Frankfurt, Germany, November 1998, Springer Verlag.

[23] Gerald Brose, JacORB: *Implementation and Design of a Java ORB*, Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems, September 30 - October 2, Cottbus, Germany, Chapman & Hall 1997.

[24] Gerald Brose, *JacORB - a Java Object Request Broker*, Technical Report B-97-02, Freie Universität Berlin, April 1997

[25] Kay Römer, Arno Puder, Frank Pilhofer, *Mico is CORBA, an Open Source CORBA 2.3 implementation*,http://www.mico.org, 1999

[26] Pascal FELBER, *The CORBA Object Group Service, A Service Approach to Object Groups in CORBA*, THESE #1867, Ecole Polytechnique Federal de Lausanne, 1998

[27] P. Felber, B. Garbinato, and R. Guerraoui. *The design of a CORBA group communication service.* In Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems, pages 150-159, October 1996.

[28] Object Management Group, *CORBAService: Common Object Services Specification*, OMG, Framingham, MA, 1997

[29] Carlo Marchetti, Massimo Mecella, Robertoa Baldoni, *Architectural Issues on Fault Tolerance in CORBA*, Proceedings of the SSGRR 2000 Computer & Business Conference, L'Aquila, Italy, 2000

[30] Carlo Marchetti, Massimo Mecella, A. Virgillito, Robertoa Baldoni, *An Interoperable Replication Logic for CORBA Systems*, Department of Computer Science, University of Rome, 2000

[31] Derek Shawn Elsaesser, *Metamorphic Objects For Dynamic Reconfiguration of CORBA-based Applications*, School of Graduate Studies and Research, Ottawa-Carleton Institute for Electrical Engineering, 2000

[32] M. Cukier, J. Ren, C. Sabmis et al., AquA: An Adaptice Archictecture that Provides Dependable Distrobuted Objects, in Proceeding of the IEEE 17th Symposium on Reliable Distributed System, West Lafayette, IN, 1998.

[33] A. Vaysburd, K. P. Birman, *Building Reliable Adaptive Distributed Object with the Maestro Tools*, Proc. Of Workshop on Dependable Distributed Object Systems, OOPSLA 1997, Atlanta, Georgia.

[34] Uyless Black, *Advanced Internet Technologies*, Prentice Hall Series, 1998

[35] Stephen Satchell, H.B.J. Clifford, Linux IP Stacks commentary, Coriolis, 2000

[36] Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, 1995

[37] Network Working Group, *RFC 1945 Hypertext Transfer Protocol -- HTTP/1.0*, May 1996

[38] Network Working Group, RFC 1866 Hypertext Markup Language - 2.0, November 1995.

[39] Anil Arora, Guillermo Leon, Scott Wallace, *The CORBA Replication Service*, University of Michigan, 1998

[40] Cameron Hughes, Tracey Hughes, *Mastering the Standard C++ Classes*, Wiley,1999

[41] Robert Robson, *Using the STL: The C++ Standard Template Library*, Second Edition, Springer, 1999

[42] V. Cimpu, D. Ionescu, M. Cimpu, "Dynamic Managed Objects for Network Management", Machine Intelligence Laboratory, University of Ottawa, Canada, 1998.

[43] T. Quatrani, *Visual Modeling With Rational Rose And UML*, Addison-Wesley Longman, 1998.

[44] Rene Meier, Paddy Nixon, *Managing Fault Tolerance Transparently using CORBA Services*, Trinity College, Dublin, Ireland, 1999

[45] K. Arnold, J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996

[46] Pellegrini, N-C, *Dynamic reconfiguration of CORBA-based applications*, Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 IEEE Computer Society, 1999, pp.329-40. Los Alamitos, CA, USA.

[47] Apache HTTP Server Documentation Project, Apache HTTP Server Version 2.0, http://httpd.apache.org/docs-2.0/, 2000

[48] A. D. Marshall, Programming in C UNIX System Calls and Subroutines using C, 1994-9

[49] Comer, and Stevens, *Internetworking with TCP/IP, Volume I: Principles, Protocols and Architecture*, Englewood Cliffs, NJ: Prentice Hall, 1994

[50] Comer and Stevens, *Internetworking with TCP/IP, Volume II: Design Implementation, and Intervals*, Englewood Cliffs, NJ: Prentice Hall, 1994

[51] Comer and Stevens, *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications*, Englewood Cliffs, NJ, Prentice Hall, 1994

[52] Spurgeon, *Quick Reference Guide To Ethernet*, Austin TX: Harris Park Press, 1995

[53] Sun Microsystems Corporation, *JavaTM 2 SDK, Standard Edition Documentation*, http://java.sun.com/j2se/1.3/docs/index.html, 1999

[54] Borland Corporation, *JBuilder 3 Documentation*, 1999

[55] William Stallings, *Data and Computer Communications*, fifth edition, Prentice Hall, 1996

[56] Pellegrini, N-C, "Dynamic reconfiguration of Corba-based applications", Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 IEEE Computer Society, 1999, pp.329-40. Los Alamitos, CA, USA.

[57] Active Network Working Group, *Architectural Framework for Active Networks*, Version 0.9, August 1998.

[58] Orbix Wonderwall Administrator's Guide, IONA Technologies PLC, Dublin, Ireland, 1997.

[59] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. *The GemStone Data Management System.* In W. Kim and F. H. Lockovsky, editors, Object-Oriented Concepts, Databases and Applications, chapter 12. ACM Press, 1989.

[60] Itasca Systems, Inc. *Itasca Systems Technical Report Number TM-92-001.* OODBMS Feature Checklist. Rev 1.1, December 1993.

[61] S. Monk and I. Sommerville. *A Model for Versioning Classes in Object-Oriented Databases*. In Proc. of the 10th British National Conf. on Databases, Aberdeen, Scotland, July 1992.

[62]A. H. Skarra and S. B. Zdonik. *Type Evolution in an Object-Oriented Database*. In Shriver and Wegner, editors, Research Directions in Object-Oriented Programming, pages 393-416. MIT Press, Cambridge, MA, 1987.

[63] *Versant Object Technology*, 4500 Bohannon Drive Menlo Park, CA 94025. Versant User Manual, 1992.

[64] R. Zicari. *A Framework for Schema Updates in an Object-Oriented Database System*. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, Building an Object Oriented Database System - The Story of . Morgan Kaufmann, San Mateo, CA, 1992.

[65] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, *Schema and Database Evolution in the Object Database System*

[66] Derek Elsaesser and Dan Ionescu: Dynamic Versioning of CORBA Applications, 5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers (CSCC 2001)