Mobile Proxies

Benjamin Aziz, B.Sc.

A dissertation submitted to the University of Dublin, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

1999

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Benjamin Aziz

17 September 1999

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: _____

Benjamin Aziz

17 September 1999

Acknowledgements

I would like to thank my supervisor, Mr. Christian Jensen, for his support and advice without which this work would not have been possible. I would also like to thank the teaching staff for all the knowledge they imparted to me throughout year.

Finally, I would like to thank my loving dad and mom for their support and encouragement.

Summary

One of the important issues that are quite often a challenge in modern distributed systems is the issue of adaptability.

The need for adaptability rises from the fact that these systems are very much open, heterogeneous environments encompassing a wide range of hardware/software products from an ever-increasing number of vendors. Therefore, mechanisms are needed that allow these products to live and interact dynamically in a way that satisfies the different performance and functionality requirements of their heterogeneous environments.

The Mobile Proxies (MP) project provides one such mechanism for client/server systems that are integrated using the Common Object Request Broker Architecture (CORBA) technology. The system achieves adaptability by exposing the low-level data traffic, exchanged between clients and servers, to different functionality-manipulating programs, each capable of modifying the data traffic in such a way that adaptability is achieved between client and server environments.

The system design relies on two major principles: The first is the well-known proxy principle, defined by Shapiro in 1986, and implemented in a number of middleware technologies including CORBA. The other is the concept of interceptors defined by the Object Management Group (OMG) as part of the CORBA v2.3 specification. These interceptors allow the above-mentioned exposure of the low-level data.

The system combines these two principles such that a client is allowed to specialize the proxy it has by downloading from the server, another proxy that performs some extra functionality. The MP system then informs the server of the downloaded proxy, allowing it in turn to specialize itself with the same type of functionality.

Two aspects of adaptability were taken into consideration when developing the system and the functionality extensions it offers. The first is performance, which aims at minimizing the communication overhead using performance-enhancement techniques like compression. The second is security, where the issues of confidentiality, message integrity, and trust of downloaded code were all taken into consideration using digital signatures and encryption (both symmetric and asymmetric).

Table of Contents

List of Figures

Chapter One
Introduction

The rapid advances in modern computer science have facilitated the development of a wide range of distributed technologies ranging from office-wide local area networks (LANs) to the worldwide Internet. These technologies form a sophisticated environment that encompasses every type of products (hardware and software) from an ever-increasing number of vendors.

Being such an open environment means that changes take place so rapidly. New products being added, old ones removed or upgraded, new standards, protocols and data formats emerging all the time. It is in fact a very dynamic and heterogeneous environment, and unless some mechanisms are devised to respond to these changes, the integrity of the whole system is in question.

One of the technical answers to this problem is adaptability.

The Mobile Proxies (MP) project is one of many attempts that aimed at achieving adaptability in distributed systems.

In this chapter, a general introduction will be made to the problem of adaptability, explaining it with a simple example, and then clarifying the goal that is sought from this project.

1.1 The Adaptability Problem

1.1.1 What is adaptability?

At the most basic level, adaptability may be defined as being the ability to alter a course of action when new information becomes available [www1]. It might also be defined as the ability of some entity to adjust (or adapt) itself to the requirements and capabilities of other entities.

In computer terms, a software entity, such as a client, a server, or a peer may have to change its state and/or behavior whenever it interacts with another new entity. This change may be necessary to make the interaction possible on one hand, and then carried out in the best possible way (or at least to some acceptable level of satisfaction) on the other hand.

The issue of adaptability rises from the fact that modern computer systems are heterogeneous environments comprising a wide range of hardware/software products from different vendors. This variety of products is driven by the openness of these systems and it eventually raises the question of how adaptable entities living in such a changing environment are, and at which level(s) of detail. Such a question can only be answered once it is determined how satisfying a certain entity is performing in its environment.

1.1.2 How Important is Adaptability?

To answer this question, it is necessary to realize what is really required of our entity. This can be seen as either a matter of survival, where unless the entity adapts itself to the rapidly changing environment, it won't be able to continue in service and will have to be changed by another more "adapting" entity. On the other hand, adaptability could only be a matter of reaching some level of satisfaction in its performance, or in other words a Quality of Service (QoS) level. Or even going one step further and trying to achieve a best effort level of service. In either case, adaptability is desirable since it lengthens the service lifetime, saves effort, time, and money.

Another important result of adaptability is to offer more flexibility and more freedom, and hence facilitating mobility. It allows mobile systems to configure themselves dynamically depending on the environment they are living in at the time and react quickly to any changes in that environment. [Katz 1994] looks at the issue of adaptation and mobility in wireless information systems where the awareness of the location and situation are taken as forms of adaptability.

1.1.3 An Example

Let's take one example of how adaptability can be used to optimize the interaction between two entities. In our case, assume a client-server application where the server is running on a host on a fixed network, and its client is running on a slow mobile network. With the server being aware of the two environments, it might for example use some performance enhancement techniques, like compression, to minimize the amount of data being exchanged with the client and hence optimizing the use of the slow mobile network the client is running on.

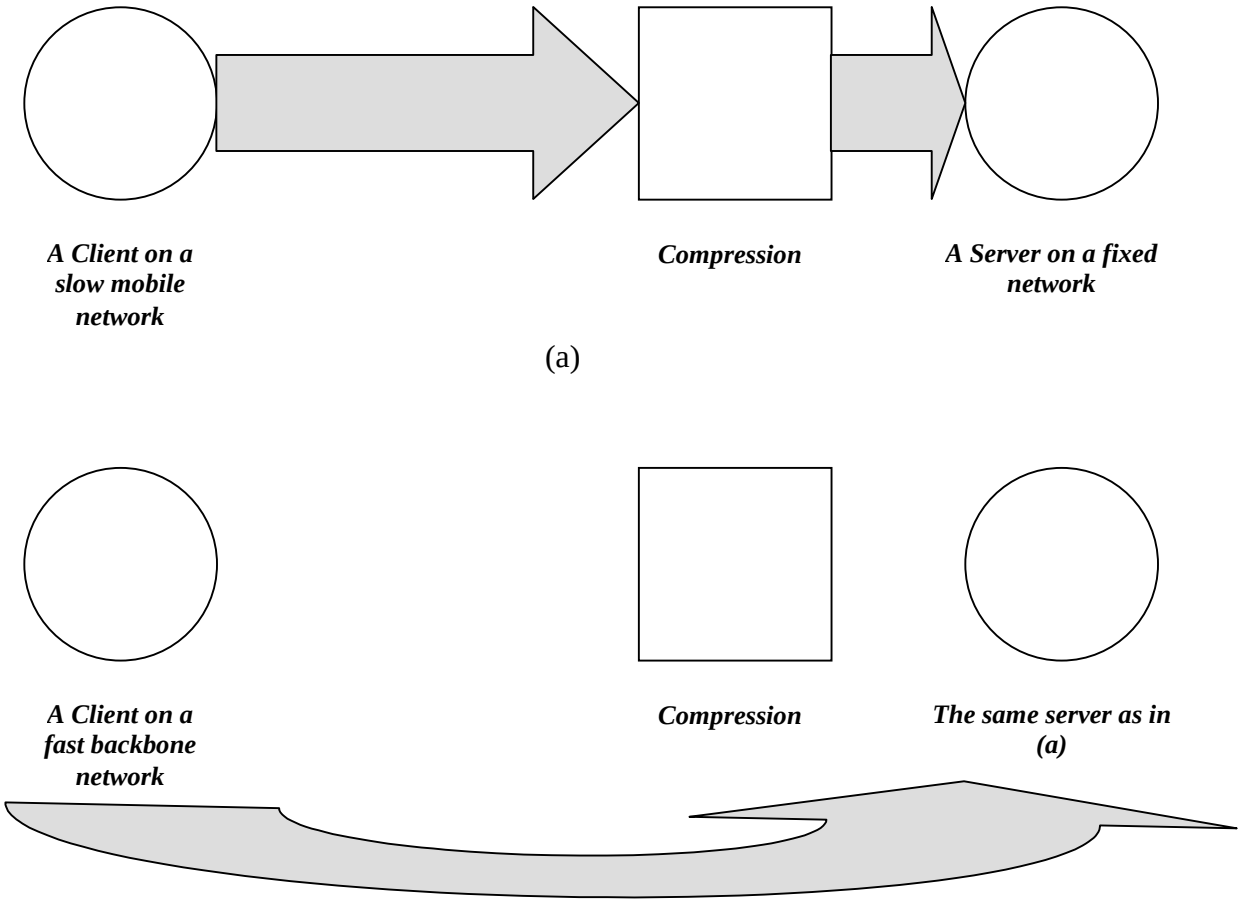

Figure 1.1: An example of adaptability.

Now another situation can occur with another client trying to interact with the same server. This client is running on a fast backbone network for which compression would be an overhead rather than performance raising. The server in this scenario, equipped with knowledge of the client's environment would have to adjust itself automatically to the new situation and switch, for example, to another mode of operation. In this new mode no compression is employed and data is exchanged directly knowing that this will yield a better performance than if that data were compressed.

With this ability of the server to switch its modes according to the different clients it's interacting with means that the server is an adaptable one, and it's trying to achieve the optimal interaction with its clients. Figure 1.1 illustrates this example, where the server in (a) is using compression in its communication with the client. In (b), the same server has "adapted" to the new client and its new environment.

1.2 The Aim of the MP Project

Many of the middleware distributed paradigms today that are used to integrate applications across a wide range of languages, operating systems, and network architectures offer a good level of transparency that hides the underlying details and helps ease the programmers task.

An application, for example, would not normally be aware of the data traffic that is going in and out of the application, and therefore cannot control it. However, it is this cross-level accessibility that would help some sort of intelligent adaptability that is at a high level (i.e.; the applications level) and yet it controls the low level details (i.e.; the byte level).

OMG's Common Object Request Broker Architecture (CORBA), which will be overviewed quickly in next chapter, defines the principle of *interceptors*, which expose to the high level of applications, the low level bytes that are moving around between the client and the server. These bytes once controlled and adjusted can enhance the performance of the application or they may be securely shielded against external attacks

if the application is running on two networks separated widely by untrusted foreign networks.

Another feature that is used by CORBA and is quite popular among other technologies as well, like Remote Procedure Calls (RPC) and Java Remote Method Invocation (RMI) is the proxy feature introduced by [Shapiro 1986] and also overviewed in next chapter.
A proxy represents the server locally at the client. It offers taking care of the low-level details of requests and replies including the marshalling and unmarshalling of the data sent. What it doesn't offer is the access to those details, something that is possible with interceptors.
The idea of this project is to implement a mechanism that combines the proxy principle with interceptors and so it can be used to achieve adaptability between CORBA client and server environments.
Any number of interceptors can be built each encapsulating a certain performance- or security-enhancement functionality, like compression or encryption, that will be used in modifying the traffic going out of and coming in to the client or server processes. A number of functionality extensions can even be combined into one interceptor.
Once these interceptors have been built, they can be used to specialize any other piece of code, including proxies. Therefore a number of proxies will come up to represent some service and each will be rightly equipped to adapt a different client environment.
As we mentioned earlier, the project aims at building a system that offers a mechanism by which a CORBA proxy, written in Java, will be specialized dynamically at runtime by downloading different interceptors from the server depending on a previous decision. This decision could have been reached through some interaction between the client and either the server or a third entity that acts for example, as a repository containing knowledge about the server's environment. This entity should be able to reach a decision, after examining the client's environment, as to what functionality best suits the client. Once the client is aware of this decision it can use the MP system to adapt to the server's environment.

Two aspects of adaptability will be tackled. The first is performance, where the delay incurred in sending and receiving data is cut by employing compression. The other aspect is security, where a client may be running on a foreign network and hence all the data sent and received would be susceptible to attacks from outside. Therefore, encryption and digital signatures would be necessary to ensure the integrity and authenticity of the all the messages exchanged with the server.

Iona's OrbixWeb was chosen as the framework for this project because of the many features it offers including its implementation of the principle of interceptors in the form of filters and transformers.
On the other hand, Java 1.1 was used as then programming environment, because of the many features it offers including code mobility and security.

Chapter Two
Survey

In this chapter, a number of concepts and technologies will be overviewed that have a direct relationship to the project.

The first is the proxy principle introduced by Shapiro in a paper in 1986 [Shapiro 1986] that appeared earlier in Nelson's Remote Procedure Calls (RPC) [Nelson 1981]. The paper defines the meaning of a proxy and its functionality as well as its properties in a distributed system.

The proxy principle reappears in a number of modern middleware technologies of which CORBA (Common Object Request Broker Architecture) is one of them and that provides the general framework under which this project is built. CORBA will be overviewed with a particular emphasis on the concept of interceptors, which, along with the concept of proxies, constitute the two most important tools used in building the MP system and reaching the goals behind it.

Also a number of other technologies are included to help make the overall picture clearer and complete as to where the project started from, and what were the options available before any real implementation took place. The chapter also shows where the system stands in terms of what it tries to achieve and how those goals were tackled by other technologies.

The first of these is the Jini™ technology put forward by Sun as a new approach to distributed computing that, as we feel, has a number of common features with our system, including mobility although, it is used in a different context.

Then there is the RMX (Reliable Multicast proXy) architecture, which aims at achieving adaptability among client/server environments but at the level of IP multicast.

Two operating systems will also be overviewed, the first of these is Spin operating system implemented by Computer Science Department, University of Washington

[www3]. The system can be dynamically specialized to offer a different interface to meet the requirements and performance of different applications and hence can be considered an adaptable system.

The final system that is included in this chapter is Sun's Spring system, which uses the concept of proxies in network communication between a client and a server. These proxies use different communication protocols and a host can have any number of them, using the one that suits the other host with which communication session is established. This could be envisioned as a form of adaptability, although in a completely different context than the one with which the MP system is concerned.

2.1 The Proxy Principle

2.1.1 Overview

The first piece of work that directly relates to this project is the proxy principle as introduced by Shapiro. The principle is widely used in modern distributed systems since it reconciles flexibility with the capability of encapsulating a structured object behind a black-box boundary.

The principle states that:*" in order for a client to avail of some service, it must first acquire a proxy for that service; the proxy being the only visible interface to the service."*

Which means that the principle is built on an object oriented model of computation, which well adapts to distributed systems, since it allows to abstract processes, processors, services, resources, and virtually everything into one concept, namely that of the object.

When an object is represented by a proxy, it's called the proxy's *principal*, and together, they form a single distributed object called a *group*, as shown in figure 2.1.

Several properties for the proxy were mentioned in the paper, including the *locality* property, which states that the proxy is always local to its client, and so the network becomes transparent and all accesses are local from the client view. Also the proxy *encapsulates* the service and hence becomes the only way to avail of that service, and it also deals with all the marshalling of the data into a network-compatible format that is ready for transmission, making it a *stub*.

Other properties were mentioned like the *access protocol* property, where a proxy enforces a certain order on the client calls, the *capability* property, where it tests the validity of calls before being dispatched to the server. Also the *trusted communication property*, since both the proxy and its server had originated from the same source, and finally the *protocol encapsulation* property, where the protocol between the server and the proxy is hidden from the client.

An important consequence of the principle is whether it allows for the implementation of open distributed systems, which are characterized by their ability for interprocess communication, which facilitates resource sharing and makes them extensible and independent from specific vendors. The encapsulation property touches on this matter although it does not state explicitly its appropriateness for the heterogeneous environments, which are a characteristic of open, distributed systems.

Concluding, the main idea behind the proxy principle is for the client to delegate the responsibility of accessing a service (delivering the request and obtaining the result) to a local proxy. This frees the client from many worries that rise from remote service invocations, and it allows the implementation of the service to be changed without informing its clients.

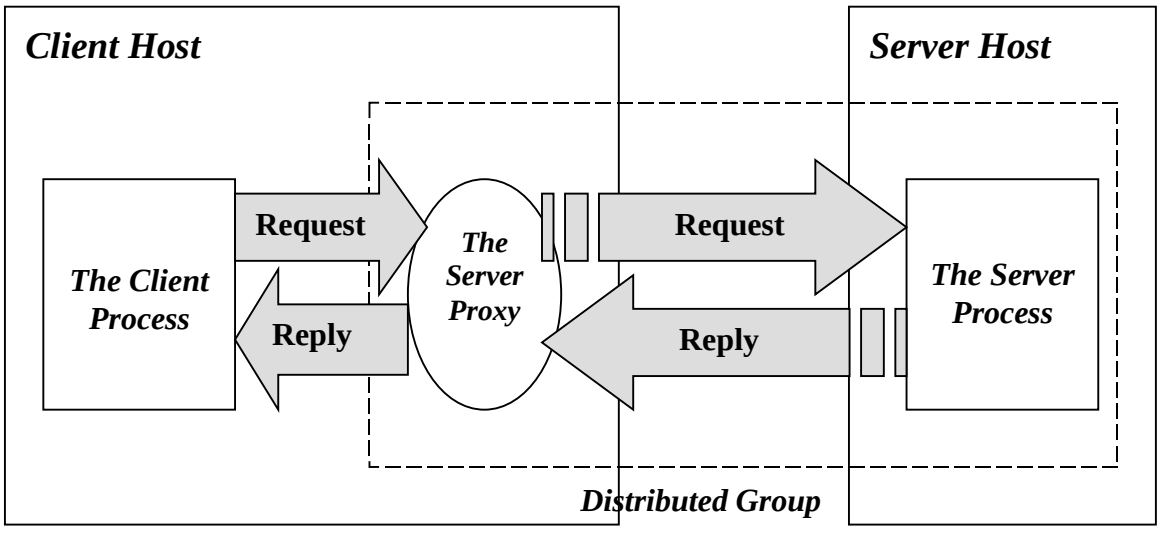

Figure 2.1: The Proxy in a Distributed System.

2.2 The Common Object Request Broker Architecture (CORBA)

2.2.1 Overview

CORBA is a standard devised by the Object Management Group (OMG) [www4] representing an influential implementation of the Object Request Broker (ORB) middleware. It is part of a wider architecture the OMG has dubbed, which is the Object Management Architecture (OMA). The two architectures are shown in figures 2.2 and 2.3.

The OMA provides a reference model within which standards are developed. It uses OMG's object model, which provides the object semantics for specifying the visible characteristics of objects in a standard and implementation independent way.

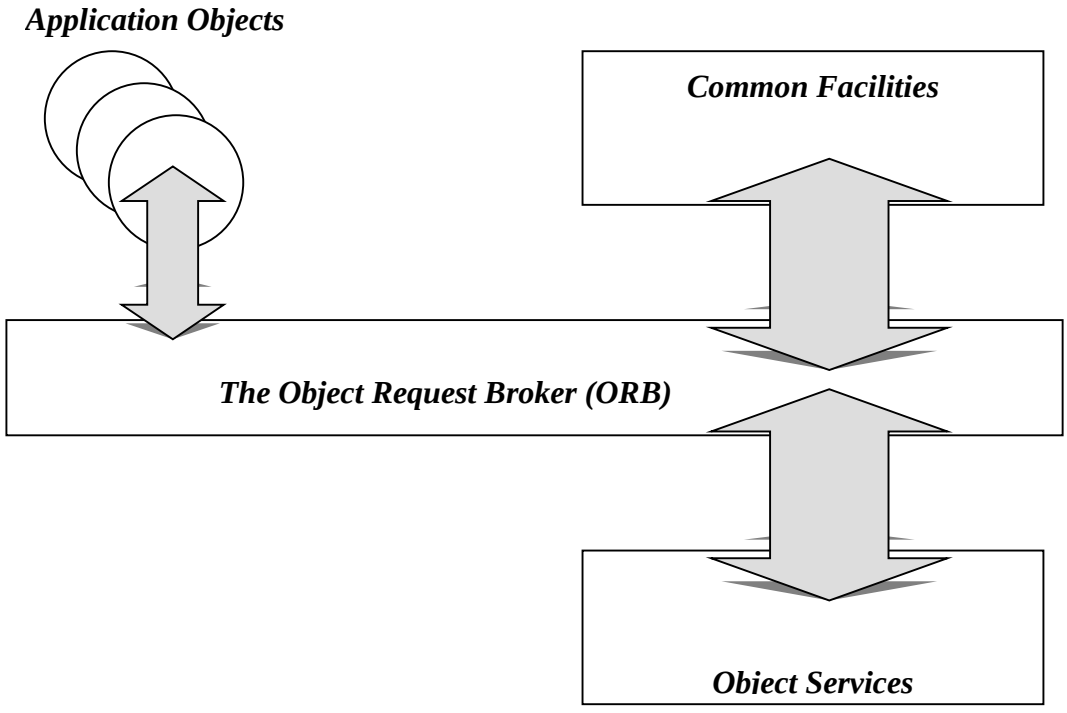

Figure 2.2: The Object Management Group Object Management Architecture

The ORB as shown from figure 2.2 lies at the heart of the OMA: It is the communication mechanism enabling objects to send and receive messages in a distributed, heterogeneous environment. Around the ORB are various services, all provided by classes and objects, which are invoked via well-defined interfaces. The language used to write these interfaces is another standard OMG has devised, called the *Interface Definition Language*

(IDL). This language achieves both location transparency (representing objects by object references) and programming language transparency, hence allowing the implementation behind the interface to be in one of several programming languages.

CORBA on the other hand portrays how the OMA can be implemented, and especially the ORB part of it. It gives an abstract definition of the ORB's functionality as a bus that conveys requests from CORBA clients to CORBA object implementations (which is the term used to describe an object that implements some service) leaving the actual implementation of the ORB to different vendors.

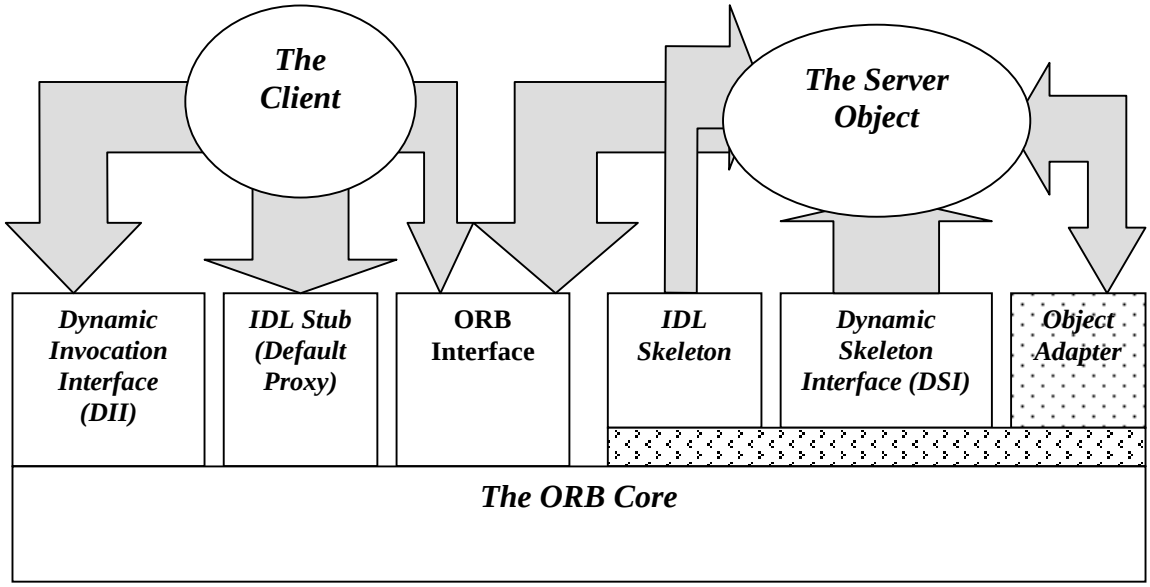

Figure 2.3: The CORBA model

The ORB is seen as composed of a number of components:

- The *IDL Stub*, which is the code generated for a specific IDL interface to allow static invocations to operations in that interface. It basically acts as a proxy to the implementation object.
- The *Dynamic Invocation Interface* (DII) which is a way for making dynamic invocations at runtime without any prior knowledge of the IDL interface. The *Interface Repository* (IR) may then be queried to obtain information about the operations and their parameters.

- The *ORB Interface* which offers an interface to miscellaneous services from the to clients and servers.
- The *IDL skeleton*, which is the code, generated for a specific IDL interface that invokes object implementations of that type.
- The *Dynamic Skeleton Interface* (DSI) which is a generic interface allowing interpretation of incoming requests to a server for IDL types that were unknown at compile time.
- And finally, the *Object Adapter*, which defines standard interfaces to servers because the ORB Core is free to be implemented in a variety of ways (depending on the vendor). Recently the *Portable Object Adapter* (POA) replaced the old standard known as the *Basic Object Adapter* (BOA).

In addition to the above components, which are basically part of the ORB Core, there exists another major part known as the ORB Services. These services are built on top of the core part and they provide additional level of transparency and functionality to the application making the ORB look as if it's a higher level entity.

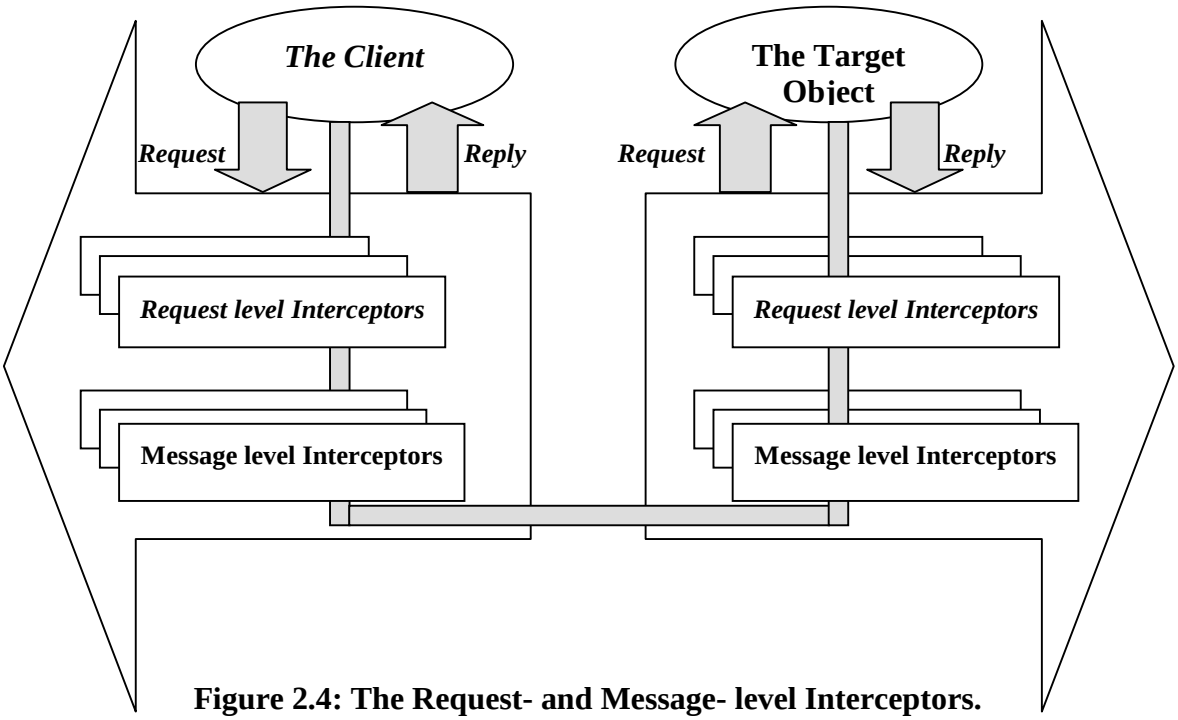

Figure 2.4: The Request- and Message- level Interceptors.

One of these services is the Security Service, which defines, under the Replaceable Security option, the concept of *Interceptors*. The specification of interceptors as well as their uses is described in [CORBA/IIOP v2.3 Spec. 1999] and figure 2.4 shows the two types of interceptors: The Request level, and the Message level interceptors. The Request level interceptor acts at a higher level (on the Request object) than the Message level interceptor does. However, the latter is the one of direct interest to the MP project since it allows access to the specific details of a message and hence permits certain performance and/or security measures to act upon it just before it is sent over the network.

2.3 The Jini™ Technology

2.3.1 Overview

Jini™ technology brought with it a new approach to distributed computing which, as claimed, will change the traditional view of what computers are and how software should be written for them.

The new technology is based on an extremely simple-to-use computing power distributed across a diverse collection of devices all connected by a network and each able to share with others' resources. This view replaces the old notion of peripherals and applications with that of network-available services and clients that use those services, all forming a flexible distributed system that can change easily over time.

Many factors played a major role in developing this view. The fact that networks are an expensive resource that live long and that are the central connecting tissues of all the modern computing systems, had to be taken into consideration by the developers of Jini™ Technology. Components, both hardware and software, are constantly added and removed making it increasingly difficult to update these networks as a single entity. Therefore, support for changing and updating network components, and the way these components interact, without having to shut down the whole network each time (which is an expensive and a difficult process) had to be provided.

A direct result of the above situation, and the fact that the new technology builds around the network, is that the data and code running on any particular device in the network

cannot be assumed by the users or the developers to have been designed especially for that device. In fact, that code and data are often constructed and gathered well before the device is designed or built.

Jini™ arose from a Sun research and development project that started in 1994 and that was committed to dramatically simplifying the interaction among networked components. The project was the inspiration of Sun cofounder and vice president Bill Joy who, with a small team of engineers, worked for four years to develop the new paradigm for distributed computing. And the result was small, very efficient, well-designed, and powerful Java-based software that does not exceed 48K of Java software binaries.

2.3.2 Jini™ Goals

On top of the list of goals that Joy's team maintained was to make adding an electronic device to a network as easy as plugging in the base unit of a new cordless phone. Jim Waldo, the chief architect, puts it in his own word [Byous 1999]: *"One of the keys behind the Jini system is that we have tried to erase the distinction between hardware and software."* This implies that basically anything with a processor, some memory, and a network connection, is allowed to offer services to other entities in the network or to use the services that are so offered. This class of devices includes not only computers, but also most of the things people usually think of as peripherals, like printers, storage devices, and specialized hardware. The potential is also there to include other classes of devices, such as cell phones, personal digital assistants, and microprocessor-controlled devices, like televisions, stereo systems, and even, modern thermostats.

The other goal that was as well important was to facilitate distributed computing by creating *federations* or *communities* of shared data, storage, and computing power, hence allowing users to easily access the needed power and features of any device on the network no matter what their individual capabilities are. Mike Clary, director of the Jini™ project describes it: *"With Jini technology, computing power could become a service where computers on a network rent their 'brains' or processors out to other computers."* This concept promotes the idea of spontaneous networking, where any number of entities can combine spontaneously together for a period of time to avail of some service or to offer one.

The size of today's networks and their rapid growth impose a final goal, or consequence of the above goals. If every device (including embedded ones) is given the chance to be part of the Jini structure, this will introduce the problem of scalability, probably, to levels previously never thought of. In other words, a Jini network has to be able to scale and adapt the increasing number of components that might be connected to it.

In short words, the Jini system aims at enabling users to share services and resources over a network, regardless of the location, the nature (being hardware or software), or the capabilities of those services and resources. Therefore simplifying the task of building, maintaining, and altering a network of devices, software, and users.

2.3.3 Why Java?

To know why Java was chosen as the language of the system comes as no surprise. Java offers a number of attractive features that help in achieving what Jini is aiming at. These include for example, code mobility where a program can be executed anywhere after being compiled into byte codes. This has promoted Java's slogan: *"write-once-run-everywhere"*. This allows code to be moved from somewhere else and loaded dynamically into a running program, hence allowing a new state and behavior to be added. Nonetheless, it also introduces serious security risk regarding the trust of the mobile code, but this issue has been solved to a certain degree by Java's security model and its extensions.

The other feature of Java is the homogeneous environment it provides to its applications by turning an otherwise heterogeneous network of computing entities into a homogeneous collection of Java Virtual Machines (JVM). This will ensure a consistent environment in which the Jini system can exist, allowing services to be able to run in their clients' environments.

As indicated above, the security model of Java would be another attractive features that enables the Jini technology maintain safety through referential integrity, array-bounds checking, and type safety. In addition to allowing fine-grain control of the operations that can be performed by any program.

In general terms, the combination of code mobility, homogeneity, and security, allows clients and services to join and leave a network federation at any time, hence representing

a spontaneous form of networking and allowing for the introduction of new services that will extend the functionality of the federation.

2.3.4 Jini™ System Structure

The Jini system can be seen as composed of three high level components: The infrastructure, the programming model, and the services. Each of these components is regarded as a logical extension of the Java language to a fully distributed case.

The infrastructure defines the minimal core technology required to build a Jini federation. It consists of three parts: A distributed security system, which builds on the Java Remote Method Invocation (RMI) and extends the basic Java Virtual Machine security model. Discovery and join protocols that allow different entities to discover and join a Jini federation. Finally, a lookup service, which acts as a repository that contains information about other services that are part of the federation, and allows clients to avail of those services.

The programming model is a set of interfaces that enable the construction of reliable services including those that are part of the infrastructure and those that join it later. It is implemented as three sets of interfaces that are meant to extend Java in a way that will permit connecting objects to the model in a robust and flexible way. The first of these sets defines a distributed event model that is an extension of the standard Java event model in JavaBeans™ and that enables event-based communication between Jini services. The second set is a two-phase-commit protocol, which is a simplified distributed version of the Java transaction service that allows Jini applications to coordinate state changes. Finally, the last set of interfaces defines the notion of leasing, which was developed especially for problems in resource allocation and reclamation in distributed systems that might arise from entities leaving the federation. The leasing model is a duration-based, renewable model.

The final component that defines a Jini system is the services component which is enabled by the first two components. A service appears as an object with a well defined set of operations that constitute its interface and that is determined by the type of the service itself. A service may be used by other services or by clients interacting with the Jini federation. Examples of services would be a printing service, a communication

service (like the JavaSpaces™), and a transaction manager, which enables entities to participate in a Jini transaction protocol.

Sun's Jini™ Architecture Specification [Jini™ Architecture Spec. 1999] provides a good description of the Jini structure including its relationship with the Java language.

2.3.5 Jini and Mobile Proxies

The notion of proxies is a central one to the Jini technology enabling the idea of spontaneous formation of entities into a federation and for other entities to join that federation and use the services advertised in it.

An entity that wants to join a Jini federation sends out a discovery request as a multicast packet to some network asking for any lookup services in it. Upon receipt of that request, a lookup service responds by sending back a proxy that represents that lookup service to the requesting entity. This happens by downloading the proxy dynamically from the lookup service to the requester with enough information that allows the latter to proceed. This process is clarified in figure 2.5. The entity then proceeds to the next stage in which it either decides to join the federation as a service, whereby it'll be required to place a proxy object representing it in the lookup service, or avails itself of some service(s) in the federation, in which case it behaves as a mere client.

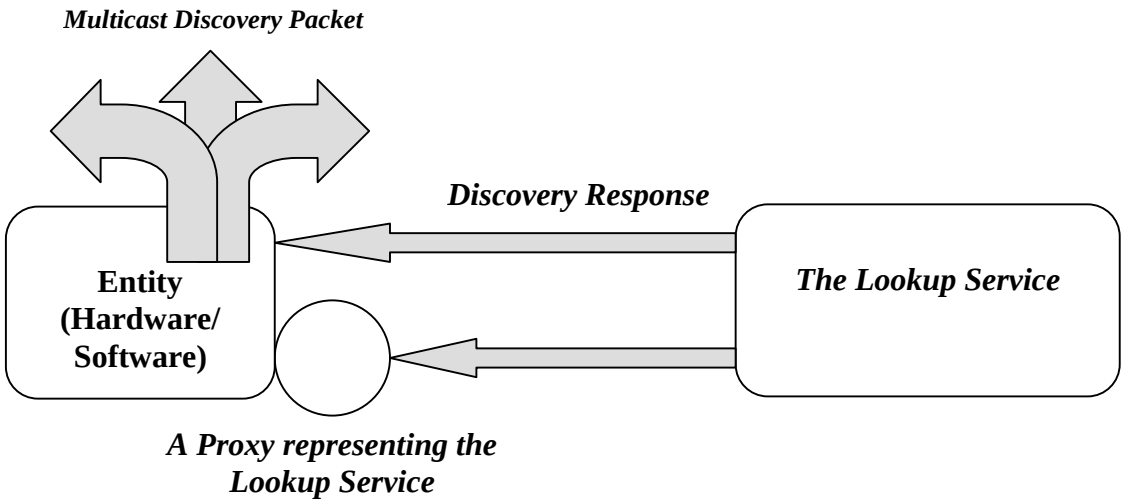

Figure 2.5: An entity discovering a Jini federation

This ability to download code at runtime is what distinguishes Jini proxies from proxies used in other distributed technologies such as CORBA. A CORBA client for example, has to know beforehand what the interface of the service it wants to use looks like. In other words, it is linked statically to the proxy of that service, and the communication knowledge between the proxy and the service is built into the client, despite the fact that this knowledge is hidden from the client (the protocol encapsulation property, section 2.1). This implies that any changes made to this communication protocol have to be coordinated in both the client and the service.
The mobility aspect of Jini's proxies allows such changes to be private to the service and the code sent to the client by that service. Effectively, these changes are only propagated as needed without the client being aware of them.

Unlike Jini, which is a complete standalone distributed paradigm; the MP system is only part of the CORBA middleware. However, the one important common feature between the two is the mobility feature that allows information to be sent to the client depending on some previous requirement. This requirement in the case of Jini is merely the need to use some service. In the case of the MP system, it is the need to adapt the client and service environments. Therefore, the nature of the downloaded code is different in the two cases. In the MP system, unlike Jini, the client has to have beforehand a proxy to the service it wants to use (called the default proxy). The downloaded proxies then are only used to specialize the original, default proxy, in a way that will meet certain environmental needs regarding performance or security, between the client and the service. Jini proxies on the other hand are not used in the context of adaptability, although the potential is very much there, and the technology as well.
Jini's lookup service could also play a role in the MP system, although in a different way, acting this time as an environmental repository that holds information about the clients and services environments. Then it would respond to queries by sending a "specialization" proxy to adapt the requester's environment.

2.4 Reliable Multicast proXy (RMX)

2.4.1 Overview

RMX is a generic model, presented by a number of researchers at Computer Science department, UC Berkley [Chawathe et al 1998] as a solution to the adaptability problem in heterogeneous environments during reliable multicast sessions.

The solution tackles the heterogeneity problem with a hybrid model that relies both on end-to-end loss recovery mechanisms and an intelligent and application-aware adaptation carried out within the network.

In multicast communication, the data sent by some source will be delivered to a number of interested receivers according to a multicast routing tree. And although multicasting offers enormous savings in bandwidth (especially for large-scale communication), it is challenged by the fact that the environments for those receivers as well as the connecting network are very much disparate. Consequently, a communication source is potentially confronted with a wide range of path characteristics to each receiver, for example, different delays, link rates, and packet losses. In addition to that, the source cannot simply modify its stream transmission to satisfy the conflicting requirements of different paths and the heterogeneous end-users.

So, the proxy model is adopted to actively transform the transmission to each receiver in a way that will bridge the gap between the end environments and any route requirements. Moreover, this will even allow communication, in the first place, among otherwise incompatible receivers.

The model suggests a twofold solution. First, it relaxes the semantics of reliability, lifting the constraint that all receivers advance uniformly with the sender's data stream, and allowing each receiver to define its own level of reliability. The second is decoupling the members of the reliable multicast session through a proxy-based communication model.

The end receivers will interact with the proxy to customize their transport decisions and reliability semantics in a fine-grained, application-specific fashion, using proxy-embedded computational and protocol bridging elements.

2.4.2 RMX Architecture

A number of features characterize the RMX design. These include the exposure of application-specific information to optimize the receiver/network adaptation process and to tune the transport protocol, otherwise known as the cross-level optimization or information accessibility (as mentioned in 1.2) and the leveraging of the data semantics when creating data adaptation algorithms

The model basically splits the communication session into two sub-sessions: the RM session (or the main multicast session) and the proxied session. The RM agent interfaces with the RM session whereas the Protocol agent interfaces with the proxied session. The model also has a Protocol Adapter that uses transformation engines in transforming the data store between the formats of the two sessions. This architecture is shown in figure 2.6.

The RM agent participates in the RM session on behalf of the RMX client (the receiver) handling all the details of the communication protocol. At the same time, it builds a data store of all the data that is part of the reliable session. Both the main and the proxied sessions update the data store, with the RM agent adding the main session data and propagating the proxied session data. The RM agent uses loss recovery mechanisms in the event that the store is lost due to a system crash, recovering the data from other agents that have participated in the reliable multicast session.

The Protocol Adapter and the Protocol Agent provide the interface to the proxied session. The Protocol agent implementing the actual communication protocol used with the clients. It could be a totally different protocol than the one used in the RM session. The Protocol Agent is designed based on Application level Framing (ALF) principles and the characteristics of the proxied clients and the network separating them. On the other hand, the RM Agent may only apply simple congestion control techniques to limit the transmission rate.

The Protocol Adapter is the most sophisticated element in the RMX model. It not only provides the required functionality for heterogeneous environments, but also relies heavily on ALF information to achieve reasonable performance.

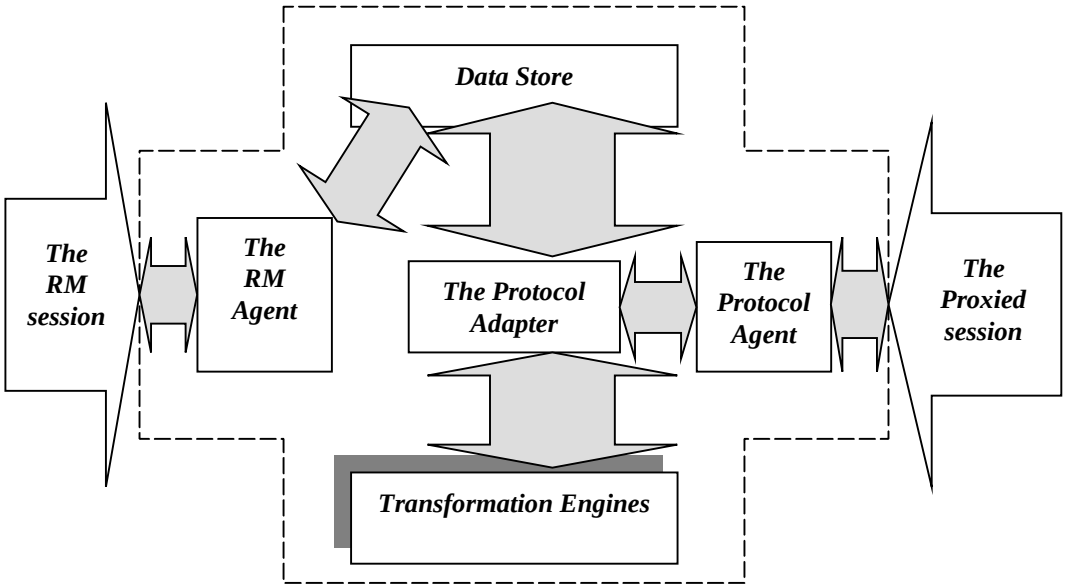

Figure 2.6: The RMX model

The Protocol Adapter also maintains a connection object for every client connected to the proxy, which encapsulates the state of that client at the proxy. This up-to-date information is used to assist the Protocol Adapter in the adaptation process. This process mainly appears in three forms of dynamic adaptation: rate adaptation which reduces the data flow rate, data transformation which transforms the format of the data, and protocol conversion which switches between different reliable multicast protocols depending on application-level information.

In addition to the above architecture, the model also describes how an RMX client locates an RMX point of contact inside a network using the Scalable Network Service (SNS) architecture also developed in UC Berkeley [Chawathe, Brewer 1998].

2.4.3 Related Concepts

The RMX model is an attempt to achieve adaptability at the network level for reliable multicast sessions, which is related to the overall adaptability problem that the MP system tries to solve but in a different context, i.e. that of the CORBA environments.

The model brings forward a number of related concepts and features apart from the common adaptability goal. It uses a proxy-based design that adjusts the data flow

between the sender and the receivers depending on the different environments, the state of the network, and application-level characteristics.

However, the term proxy is used to convey the simple meaning of an intermediary between the client and the server, which is somehow different than the view adopted in middleware systems that a proxy as a representative of a service having the exact interface and being local to the client.

In the context of CORBA, RMX would fall in the same classification as an interceptor that has access to the low level data bytes of the application. Nonetheless, RMX also has access to application level information, which adds more complexity to it than a mere interceptor.

The model also touches on performance-enhancement techniques like compression, and transport-layer-specific techniques like protocol conversion and data flow rate control. Compression in particular was used in the MP project as an example of controlling the output from the client to the server and vice versa. The security aspect was left out of the RMX model.

2.5 SPIN Operating System

2.5.1 Overview

SPIN operating system was developed by a group of researchers [Bershad et al 1995] from UC Berkley as a project sponsored by the Advanced Research Projects Agency (ARPA), the National Science Foundation (NSF) in 1995.

The project was aimed at designing an operating system that could be dynamically specialized and extended so that it could (safely) meet performance and functionality requirements of the applications using it.

Such requirements are often poorly matched by operating systems and therefore result in either the application not working well, or not working at all. For example, modern operating systems employ disk buffering and paging algorithms that could pose potential performance degradation for certain types of applications like database applications. Also general-purpose network protocols are quite so often incapable of supporting the

high-throughput parallel applications. Other applications have their own special needs that are not always met by operating systems, like multimedia clients and servers, fault tolerant systems, and real-time-data applications.

SPIN is described as an extensible operating system that can provide an extension infrastructure, represented by a set of interfaces, to its applications in such a way that will best meet the demands of those applications. These extensions allow the application to "specialize" the underlying operating system and to grant fine-grained access to system resources in a safe and low-cost way. This access has to be controlled to ensure safety, whereas good performance is only achieved by low communication overhead between the extensions and the operating system kernel.

2.5.2 SPIN System Design

The design of SPIN relies basically on four techniques implemented at the level of the language in which it was written, or it runtime. These techniques are *Co-location,* which enables communication between the system and the set of extensions to have a low overhead. Also *enforced modularity,* through which modules are isolated from one another by boundaries that are well defined by their interfaces. Extensions were written in Modula-3[1], a modular programming language. The other technique used was the *logical protected domains*[2]*,* which ensured that extensions were executed in separate namespaces protecting their effects from other programs. And finally, extensions were only allowed to execute in response to system events, which described potential actions in the system, like a virtual memory page fault or the scheduling of a thread. This is termed as the *dynamic call binding.*

However, it should be noted that these techniques do not guarantee extensibility. It is only guaranteed by the infrastructure functionality provided by the set of extensions and accessed through the exported interfaces.

[1] v-Promela is a recent Visual Object-Oriented language proposed for SPIN, see [Leue et al 1999].
[2] A good example of logical protected domains is the sandbox security model in Java.

In general, SPIN system design encompasses two models, the protection model and the extension model. The protection model controls the set of operations that can be applied to resources. It ensures that a process can only access the memory within a particular set of virtual addresses. It makes use of the notion of capabilities, which are unforgeable references to a resource. As mentioned above, the protected domains are used in defining the set of accessible names available to an execution context.

The extension model provides a controlled communication facility between the extensions and the core system operating system allowing for a variety of interaction styles.

These two models provide the framework for managing interfaces between services within the system.

2.5.3 Related Concepts

The SPIN operating system is an adaptable, dynamic, flexible system providing different functionality extensions to the applications using it. In other words, it aims at adapting the needs of those applications although this adaptability is more at the high level of applications rather than the low level environment adaptation that represents the goal of the MP project. However, the extensibility represented by the set of interfaces, each providing a different functionality appears in the form of the mobile proxies, each of which carries with it a different form of specialization when sent to the client. The difference should be noted though in the two systems, since SPIN is an operating system, whereas MP is part of an integrating middleware, i.e. CORBA.

2.6 Spring Operating System

2.6.1 Overview

Spring is a highly modular, distributed, multi-threaded operating system that not only supports object-oriented applications but also is object-oriented in itself.

The system was developed in Sun Microsystems Laboratories by a group of researchers [Mitchell et al 1993] who were working on a clean sheet design for a new operating system that would have the potential of replacing Unix. The new system would keep as

many of the good features of Unix as possible, like good performance, memory protection, and network interoperability. However, it would improve in areas that suffered from problems, like the cost of maintenance, the inflexible security aspect, and the difficulty of supporting time-critical media.

Because the design was a fresh one, the opportunity was there to make full use of the best available technology to come out with an open and flexible system, that is able to evolve and be extended by different vendors. Such a system would have to pay attention to the different components that construct it and the interfaces between them. This requirement immediately dictated a system structured around the notion of objects that are defined by strong interfaces specifying what these objects do and leaving the actual implementation open. These interfaces allowed operation invocations to be type safe, secure, and uniform, whether the client invoking a method is in the same address space, in another address space, or on a different machine altogether.

Further, in order that these interfaces are defined without tying them to a particular programming language (and hence undermining the openness of the system), an Interface Definition Language (IDL) was used. This language is basically the same as the one adopted by OMG.

2.6.2 Spring System Structure

Spring is designed using the microkernel approach. The kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management.

A typical Spring node runs several system servers in addition to the kernel. Two of these run in the kernel mode: The *nucleus* and the *virtual memory manager*.

The nucleus manages process and inter-process communication, and it supports three basic abstractions: *domains*, which are analogous to Unix processes, *threads*, which execute within domains, and *doors* which support object-oriented calls and provide entry points between domains.

The virtual memory manger controls the memory management hardware. The rest of the system services, including naming, paging, network IO, file systems, keyboard management, etc., are implemented as user-level servers.

These services are available on all the nodes in a distributed system and basically provide access to the system resources they manage through their interfaces allowing clients to use those resources in a safe way by invoking operations on these interfaces.

Also appearing in the Spring structure are *network proxies* that support invocations among objects across the network. However, they are normal user-mode server domains that receive no special support from the nucleus. A Spring machine might have any number of them, that speak different network protocols.

Spring proxies transparently forward door invocations between domains on different machines with neither the client nor the server being aware of their existence. Network addresses and handles are used in identifying these proxies and the doors using them.

Figure 2.7 shows how an invocation is routed from the client to the server through doors and network proxies.

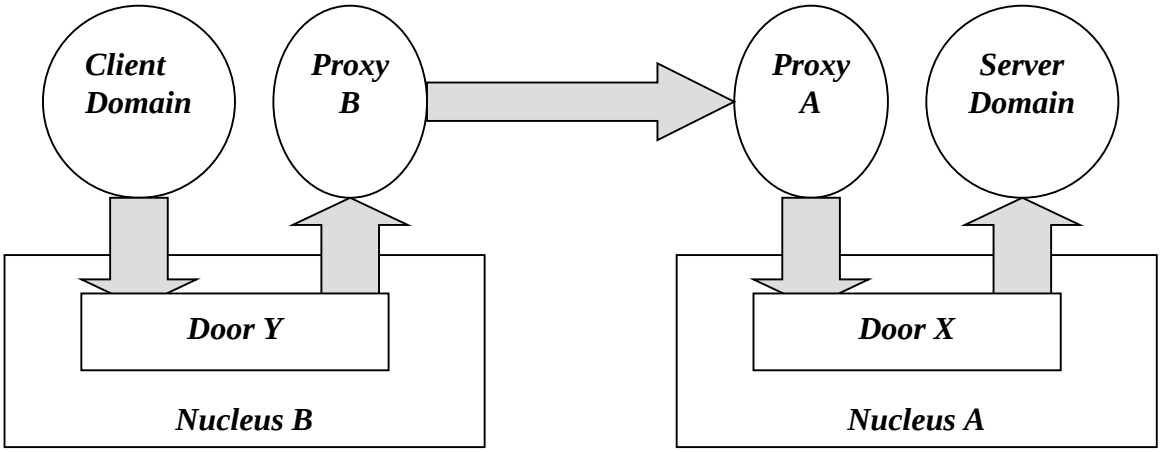

Figure 2.7: An invocation in Spring Operating System [Mitchell et al 1993]

[Nelson et al 1993] provides a further outlook into the structure of the Spring file system and the Spring virtual memory system respectively.

2.6.3 Related Concepts

The Spring operating system is another example of how proxies can be used as intermediaries between client/server communication. Although Spring proxies have a limited functionality regarding adaptability at the low level of data bytes, yet it could be argued that they are working at the protocol level simply by having the system use the suitable proxy depending on what protocol the server speaks. This however is the only common concept that would fall into the same set of concepts that the MP system relies on.

Chapter Three
System Design

In this chapter, a high level description of the MP system design is given with all its major parts and the way these parts combine to make up the system.

Also described are the performance and security techniques employed in providing distributed applications that use the system, with the extra functionality needed by those applications to enhance the interaction between different components of the application.

The chapter begins with an overview of the general scene in which the MP system works and the other entities with which it interacts.

One of these entities is the ***Environmental Repository***, which alongside the ***Environmental*** and ***Decision*** objects, will also be overviewed but only in general without going into further details, as it does not form a part of the project, but is rather a separate system that deserves its own research.

3.1 Overview of the General Scene

The MP system basically provides a mechanism by which applications using the CORBA middleware can adapt to different environments and meet the different requirements of the applications. However, in order for the MP system to meet its expected goals and become fully functional, other parties are assumed to exist.

The overall scene is depicted in figure 3.1 with basically three major entities that the MP system has to interact with. These are:

- The Application
- The Object Request Broker (ORB)
- The Environmental Repository

3.1.1 The Application

This is just an ordinary CORBA-integrated application, comprising a client process that uses a server object reachable through the ORB. The client makes all the method invocations through the local proxy that represents the server object[1].

The server process will also be the provider of a number of adaptable proxies[2]. These proxies carry the same exact interface of the server process but also implement a certain interface that will make them adaptable.

The current version of the MP system only supports applications using OrbixWeb from Iona.

3.1.2 The ORB

The ORB was briefly discussed in section 2.2 being the integrating bus that carries the requests sent from clients to different server objects living in the CORBA environment.

3.1.3 The Environmental Repository

Finally, there is the Environmental Repository, a new concept suggested throughout this project that is meant to fill two roles. The first is being a holder for the description of different services and the environments in which these services live. The second is to compare this description with that of the clients willing to avail of these services in a *decision-making* process.

The description of the services and their environments could be some form of Meta data that must be standardized in order for the description to be valid across different repositories and the clients interacting with them.

The Environmental objects will be used to represent the description of different environments. This description could be a set of parameters or variables that control the state of these objects and that reflect the state of the environment they represent. They will also contain ***Application Level Data*** (ALD) necessary for the decision-making process. The outcome of the latter process would then be embedded in a Decision object.

---

[1] In CORBA jargon, this is the *stub*. In this project, we will refer to it as the *default proxy*.

[2] These are also termed the mobile proxies, because of their mobility.

The Environmental Repository might use any techniques in reaching its decision, ranging from simple if-then rules, to high-level intelligent methods like ***Case Base Reasoning*** (CBR).

It is important to note that the MP system will be part of both the client and the server processes. These two parts will be referred to as the ***Client MPS*** and the ***Server MPS*** respectively. They basically provide a set of APIs that the client and the server processes will have to call in order to initiate and end the adaptation process.

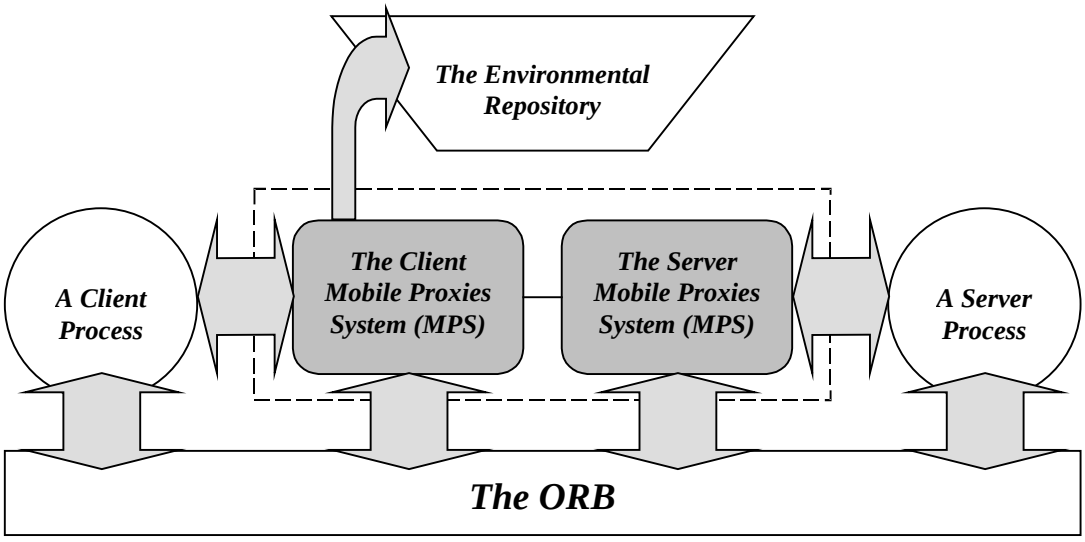

Figure 3.1: The Mobile Proxies System and other entities

3.2 Five Steps to Adaptability

3.2.1 Step One

The first step in the interaction process between different entities starts with the Client MPS contacting the Environmental Repository on behalf of the client process and supplying it with an Environmental Object. This object will describe the client's environment as well as any other parameters that would be useful in deciding the necessary functionality that will control the flow of data between the client and server. This point marks the beginning of the adaptation process.

3.2.2 Steps Two and Three

The repository then undergoes a decision-making process that ends with a decision made as to what functionality should be used. This decision will then be passed to the Client MPS in the form of a Decision object. This is shown in figure 3.2.

The Decision object will normally hold the decision as a URL pointing to the location of the adaptable proxy at the server site. Along with the URL will also be its digital signature, which will be used in verifying the source of that URL. The object may also hold other information related to the server, such as any digital keys or certificates that would be necessary for security purposes, such as authentication or message confidentiality, specially that code mobility is involved.

Again the emphasis here is on the fact that these objects, as well as the Environmental ones, have to have a standard format that will be compatible everywhere, regardless of the application or the environment it's built in.

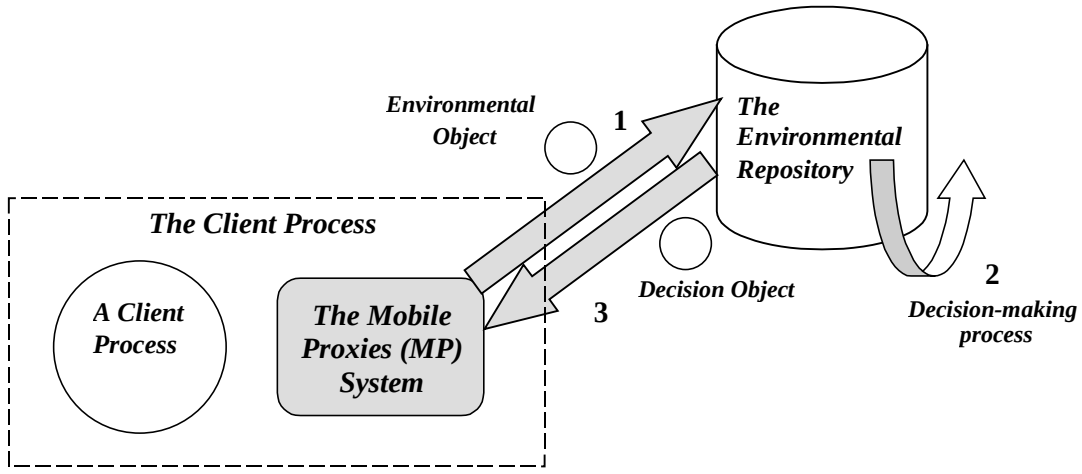

Figure 3.2:

4. The Client MPS sending an Environmental object

5. The decision-making process in the Environmental Repository

6. The Client MPS receiving a Decision object

3.2.3 Step Four

The next step involves the Client MPS downloading the adaptable proxy along with any code referenced by it from the server process and then installing it in the client process. This step is depicted in figure 3.3.

The code mobility aspect was preferred to the other choice, i.e. the client having all the necessary adaptable proxies locally stored (just as it has the default proxy) and loaded at runtime into the JVM. Being mobile meant that the same advantages mentioned in section 2.3.3 would be retained, as well as, adding an extra advantage, i.e. that of extensibility.

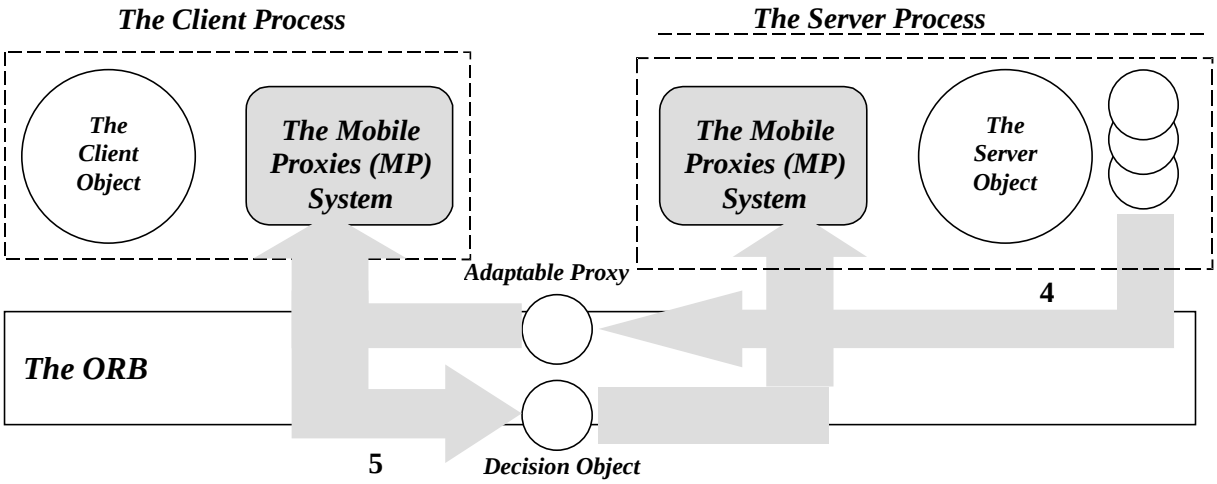

Figure 3.3:

6. **The Client MPS downloads the adaptable proxy, then**
7. **It sends the Decision object to the Server MPS**

Extensibility here refers to the fact that the server is free to implement as many of the adaptable proxies as it wishes without having to notify the client. New proxies can be added and older ones removed depending on the necessity. This also allows for a lightweight client that does not have, at compilation time, more than the default proxy generated by the CORBA IDL compiler.

3.2.4 Step Five

After downloading the necessary code, the Client MPS will send the Decision object across to the Server MPS through the ORB. This will distribute the knowledge of the outcome of the decision-making process between the client and the server processes and so an agreement will have been reached as to what functionality should both sides use to best match their needs.

The Client MPS will then call a method in the adaptable proxy to install the necessary interceptor in the client process. The same thing takes place on the server side, with the Server MPS installing another interceptor that complements the client-side interceptor.

These interceptors are only used to expose the low-level data bytes running between the two processes. A number of objects each implementing a different functionality is then used to modify and control these bytes. The functionality of these objects is defined through a set of interfaces.

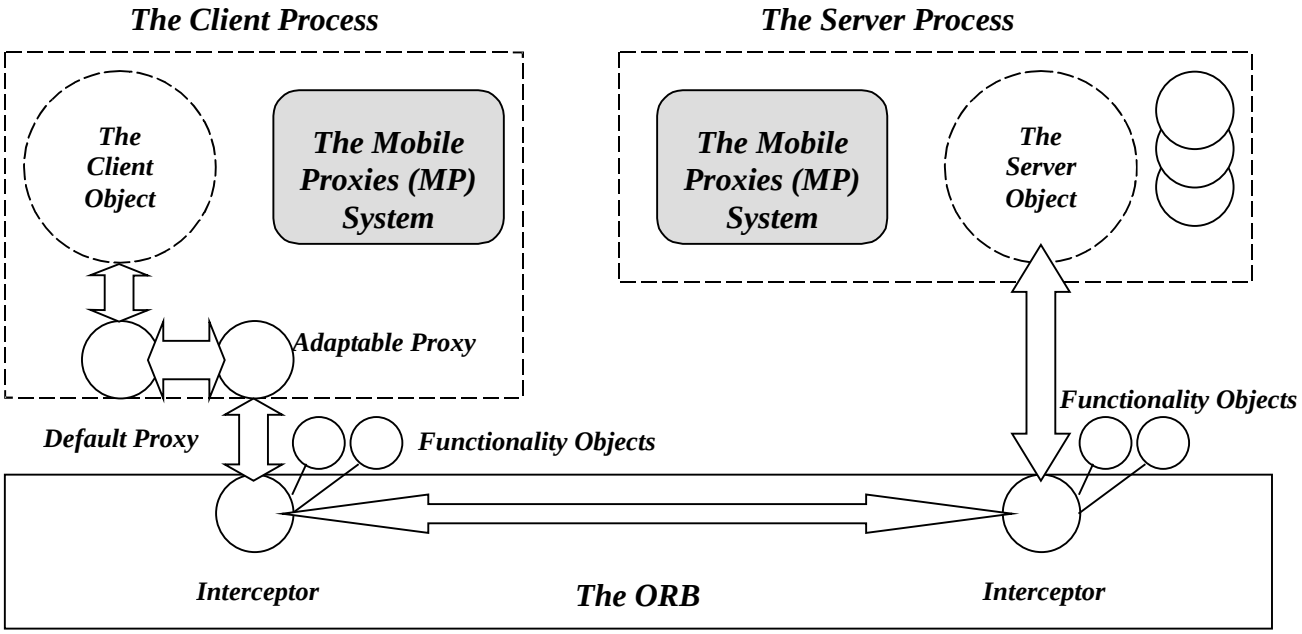

Figure 3.4: The interaction between the client and the server using the MP system

It is important to keep in mind that all these objects are downloaded as necessary from the server site at runtime.

This step denotes the end of the adaptation process and invocations can now take place and data can be exchanged between the client and server as depicted in figure 3.4.

3.2.5 The Problem of Concurrency

One important issue that the Server MPS has to deal with is the concurrency issue. It is possible that more than one client at a time opens a connection to the server process. The Server MPS should be able then to deal with these clients allowing each to implement a different functionality and at the same time reversing the effects of that functionality when the modified data is received at the server side.

One way of achieving this is for the Server MPS to spawn a different child thread for every new client that connects to the server process. This will free the Server MPS to deal with new clients coming up. It will also allow the child thread to be specialized for only one client.

The child thread will encompass all the objects used to treat the incoming data as well as modifying the outgoing ones. It will also have a reference to the server object to which method invocations are made. This however implies that the server object should in itself be able to deal with multiple requests, something that must be taken care of at development stage.

Another way of dealing with multiple clients would be for those clients to transmit some knowledge with the data, which would inform the Server MPS of the way that data was modified. This knowledge will be inserted by the Client MPS at each client process and would indicate the source of the data. This should be sufficient in determining how the modified data can be restored to its normal state taking into consideration the fact that that the Server MPS already has an idea about the different clients connected and the different decisions reached for each client (represented by the Decision objects.)

The Server MPS will still have to know how the outgoing data would be modified according to its client.

These two solutions to the concurrency problem are illustrated in figure 3.5 where in (a), the Server MPS spawns a different thread to deal with each client, whereas in (b), the Server MPS has the necessary knowledge as to the source of the data received.

After the client finishes invoking the server, the connection is closed between the two processes and the Server MPS will have to remove any objects that were installed during the adaptation process.

However, the situation is different in the case of multiple clients. In the first situation the Server MPS will have to kill the child thread that was dealing with that client. In the second situation, the Server MPS will only remove the client's entry containing, at least, the id and the functionality used in modifying the data.

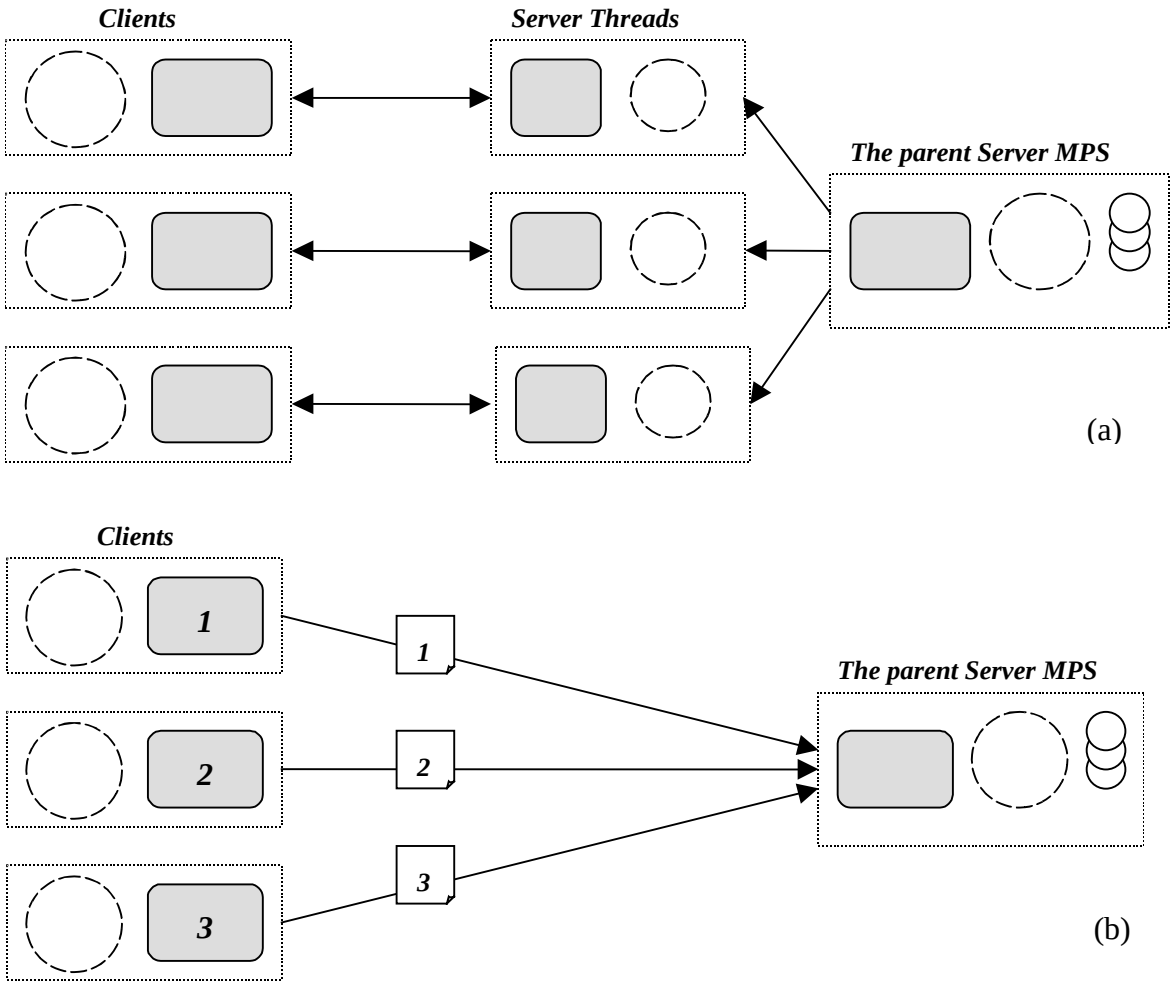

Figure 3.5: Concurrency in the MP system

3.3 The MP System Design

The MP system is generally divided into five modules that combine together to provide a set APIs for the application developer and help achieve the goal of adaptability.
These modules are:

- The **Client Adapter**
- The **Server Adapter**
- The **ORB Adapter**
- The **Proxy Loader**
- The **Class Loaders**

Which are displayed below in figure 3.6 and further described in the following sections.
It is worth noting that the Client Adapter, the ORB Adapter, and the Proxy Loader form the Client MPS, whereas the Server Adapter represents the Server MPS. The Class Loaders are part of both the client and Server MPS.

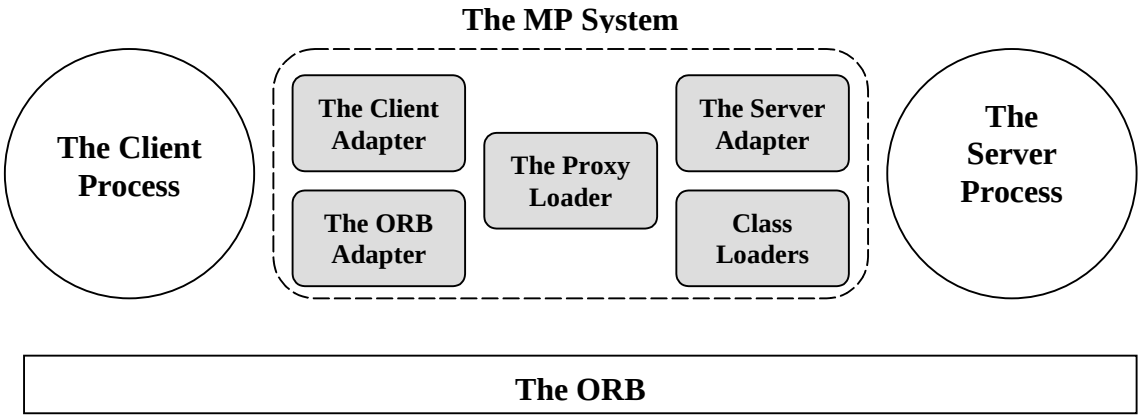

Figure 3.6: The different modules of the MP system

3.3.1 The Client Adapter

This module is nothing more than a front-end to the client process. It is itself part of the client process and it provides to the client a set of APIs that will start and end the adaptation process.

However, this module is also responsible for contacting the Environmental Repository and supplying it with the Environmental object and receiving the Decision object.
It will also carry out a number of other tasks, including sending the Decision object to the Server MPS using the Proxy Loader module, calling the appropriate method(s) on the adaptable proxies that will cause them to adapt the environment. And finally, making sure that the adaptation process was successful while informing the client of any exceptions.

3.3.2 The ORB Adapter

The ORB adapter deals with the ORB and provides some means for it to synchronize the adaptation process, especially at the point where an adaptable proxy is downloaded from the server.
It allows the ORB to inform the Client MPS of the point in time when a proxy is needed either because a reference to the server object has entered the client process or the client has decide to bind to that object. The Proxy Loader will act at this point and download an adaptable proxy that will override the default proxy at the client.
This module is only part of the client process and the Client MPS.

3.3.3 The Proxy Loader

The Proxy Loader has a major task of downloading the adaptable proxies from the server. It also provides the means for the Client Adapter to send the Decision object to the Server MPS. In turn, it uses the Class Loaders in its downloading procedure.
This module is only part of the client process and the Client MPS.

3.3.4 The Class Loaders

This module is only a proprietary class loader that overrides the default class loader, which is usually provided with the JVM.
Being part of both the client and server processes, it has to be able to load Java classes across the network as well as locally. This ability is needed because of the fact that the client does not have the class files for the adaptable proxies. And although the server

does have them, the proxy used remains anonymous to the latter until the point when the Client Adapter sends the Decision object to the Server MPS.
This module also has to deal with the security issue rising from code mobility. This is worked out by using digital signatures to authenticate the mobile code.

3.3.5 The Server Adapter

The Server Adapter is the equivalent of the Client Adapter, but at the server process. It performs all the necessary tasks, including receiving the Decision object, calling the necessary method(s) on the adaptable proxies causing them to adapt the environment. Also keeping track of all the connected clients and spawning different threads to deal with each, or monitoring the incoming and the outgoing data, depending on the solution adopted for the concurrency problem mentioned in the previous section.
This module constitutes the Server MPS.

The previous classification of the different modules of the system does not imply that these are separate Java classes, but rather is based on the different roles performed by each of the modules.
The system design was meant to be as flexible and lightweight as possible so that its performance cost would be kept to the minimum and at the same time allowing for the different modules to be replaced and upgraded as necessary.

3.4 The Functionality Design

Basically, the MP system is targeted at achieving adaptability in two contexts, i.e. communication performance and security.
The performance of any application can be improved by in a number of ways. One of these ways is to reduce the delay incurred when data is transmitted from one machine to another across the network.
This delay varies among different networks and is a function of a number of factors, including the network bandwidth and the amount of traffic flowing at a certain point in time.

One of the factors linearly affecting the transmission delay is the size of data that is being sent over the network. Size reduction could lead to considerable gains in performance since the transmission time would be reduced. This reduction however represents extra processing time, hence the trade off in this situation would be the transmission time versus the processing time. More discussion of this issue is provided in chapter five.

The security side of the system works in the three areas often termed *CIA*. These are: *Confidentiality*, *Integrity*, and *Authenticity*.

Confidentiality refers to the ability of the application to hide or protect its data from unauthorized access, otherwise known as *Eavesdropping*. Protecting against eavesdropping becomes even more difficult when that data is exposed to the outside world while transmitted over the network.

Encryption is normally the solution to securing confidentiality. Encryption usually falls into three broad categories that are populated by different algorithms: *Symmetric*, *Asymmetric*, and *Hybrid* encryption. Examples of these include the **Data Encryption Standard** (DES) and the **International Data Encryption Algorithm** (IDEA) which are symmetric algorithms. The **Rivest-Shamir-Adleman** (RSA) is an example of an asymmetric algorithm. On the other hand, **Pretty Good Privacy** (PGP) and **Secure Sockets Layer** (SSL) use a combination of the previous two and so they are often termed hybrid algorithms.

Integrity means that the data sent has not been tampered with or altered. Message Digests are one good and efficient way of achieving this through providing digital fingerprints. Examples of message digest algorithms are the **Secure Hashing Algorithm** (SHA) and MD5.

Finally, Authenticity is to verify the source of the data received, and prove its real identity. Digital Signatures are used in proving the authenticity of messages and DSA is one algorithm that achieves this.

Two good references on the different security concerns and solutions are [Knudsen 1998] and [Oaks 1998].

3.4.1 The Functionality Tree

The design of the system functionality was made as open as possible in order for any enhancements or new functions to be added in subsequent versions of the system and still remain compatible with the previous ones.

The design takes the form of a hierarchical tree that can be expanded horizontally and vertically allowing for new elements to be added at each level. This tree is shown in figure 3.7.

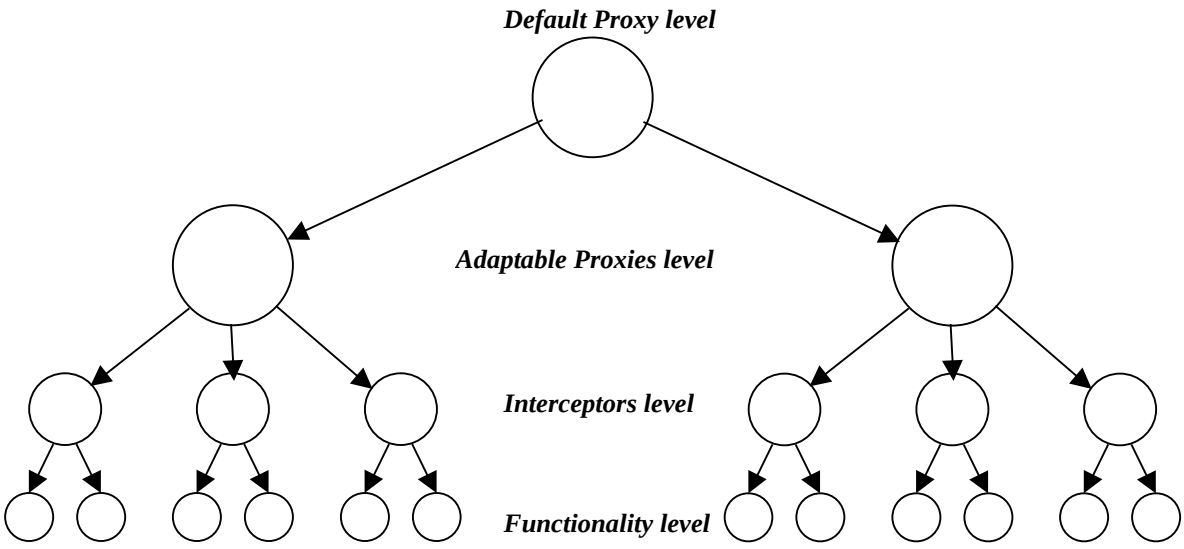

Figure 3.7: The MP system functionality tree

3.4.2 The Four Levels

The first level is the default proxy level. This level contains the normal default CORBA proxy generated by the IDL compiler and statically linked to the client. It mirrors the service interface.

The next level represents the adaptable proxies, which are extra proxies for the same service but that have the ability to adapt to the requirements of the client and the needs of its environment. This ability is defined by a certain interface that has to be implemented by any proxy willing to become adaptable.

These proxies will be stored at the server site and will be downloaded at runtime by the client, overriding the default proxy of the first level. There can be any number of them depending on the different adaptability semantics needed.

The third level defines the interceptors used in intercepting the data bytes going out and coming in of each process. These are the Message-level interceptors defined in CORBA v2.3 specification and discussed in section (2.2).

In the design of the MP system, these interceptors are only used to expose the low-level data bytes without imposing any modification on them. This again is meant to keep the system as open as possible, and to allow for different types of modifications to be pipelined and therefore increase the flexibility of the adaptation process.

The fourth and the last level encompass the different functionality-implementing objects. These objects are defined by a set of interfaces each reflecting a particular technique like, compression, encryption, digital signatures and others.

These interfaces can be implemented in a variety of ways and for any number of algorithms.

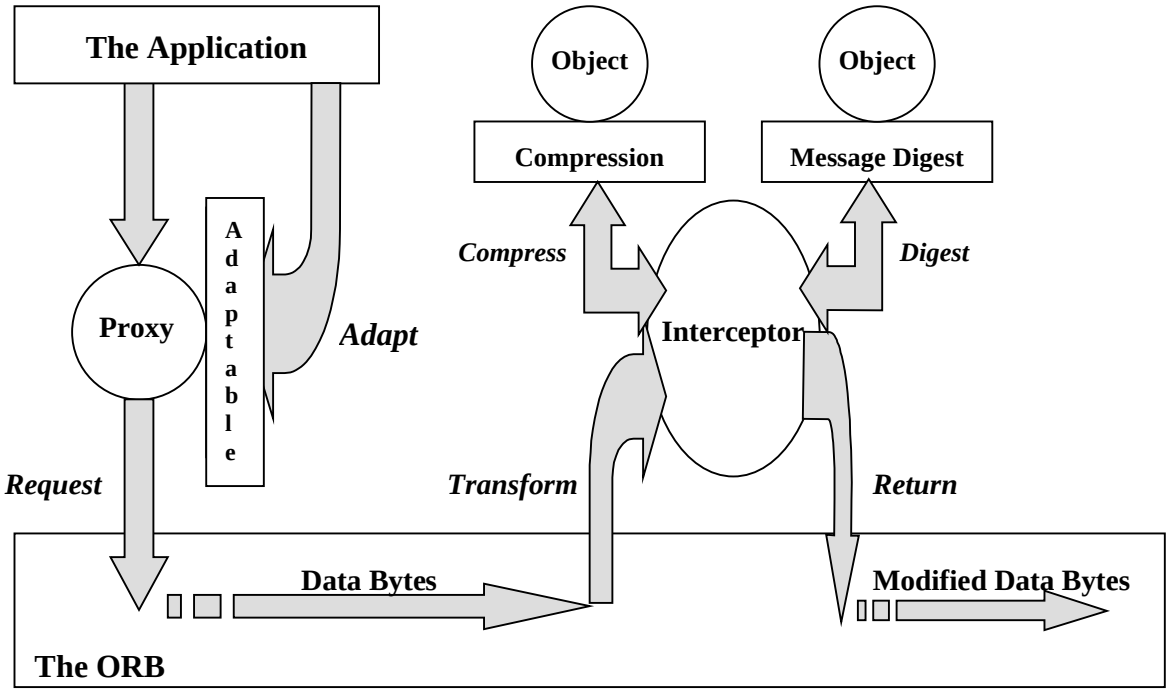

Figure 3.8: Adaptable proxies, interceptors and functionality objects in action

Figure 3.8 illustrates one example in which the different elements of the functionality tree are seen in action. An adaptable proxy carries certain semantics like compression and message integrity. The proxy will then install an interceptor that exposes the data to be compressed and digested. This interceptor will in turn call methods in specific objects that actually perform compression and message digesting.

In the next chapter, the implementation of the parts of the system and its functionality is discussed in detail.

Chapter Four
System Implementation

In this chapter, a complete description of the implementation of the MP system is given. The system design discussed in the previous chapter is revisited from a different angle with each module and the classes it evolved to.
Provided also is a stub implementation of the Environmental Repository and Decision object, leaving the implementation of the Environmental object for further research as this object was not crucial to get things up and running.
Finally, the interaction between the different classes is outlined so that a clear picture will be drawn as to how the system achieved its goals.

Java 1.1 was chosen as the language of implementation for all the features it provides and that were described in section 2.3.3. The export restrictions imposed on the Java Cryptography Extension (JCE) from Sun however meant that an extra package had to be used to implement the encryption functionality of the system. The JCE package from the Australian Business Access Pty Ltd was chosen for many reasons not least of being the clean room implementation it provides of Sun's JCE. All about the package can be found in this page [www5].
OrbixWeb v3.1c from Iona Technologies was adopted as a product of CORBA that combines most if not all of the concepts and features that the MP system was based on.

4.1 Overview of the Implementation

A number of packages were developed as an implementation of most of the entities involved in the project. These combine together to form the MP v1.0 system. However, not all of the design features and requirements were satisfied in this version. These limitations will be referred to as appropriate in the description of the packages.

The first of these packages is the **MobileProxies package**, which encompasses all the five modules that form the basic structure of the system as described in section 3.3. This package gives an implementation of the core MP system classes.
The next three packages include the **MobileProxies.interceptors** package that defines the interceptor classes, the **MobileProxies.interfaces** package, which contains the interfaces defining the functionality objects, and the **MobileProxies.functionality**, which has all the functionality objects.
These packages implement the two low levels of the functionality tree (figure 3.7).
The last package is not really part of the MP system but rather is employed by the system. The **MobileProxies.repository** package gives a stub implementation of the Environmental Repository and the Decision object, leaving out the Environmental object.

In the following sections, a description of each package is provided and the role its classes satisfy in the overall system design as well as the interaction between them.

4.2 The MobileProxies Package

The package consists of ten classes and one interface. Four of the classes are used by the Client MPS, another four by the Server MPS, and the remaining two by both of them. The interface defines couple of methods that have to be implemented by any adaptable proxy.
These classes will be overviewed in the following sections, where each class is fitted into the system module it belongs to, and so a complete picture of the core MP system structure is portrayed.

4.2.1 Class MobileProxies.ClientEnvAdapter

This class represents the Client Adapter module. It basically provides a couple of methods for the client application to invoke in order to start and end the adaptation process. These are the *startAdapting* and the *endAdapting* methods.
In the v1.0 of the MP system this class has a Constructor that reads the Decision object from a file called *Client_Decision.object* which will have been generated by the

Environmental Repository. The file contains a serialized Decision object with the decision as what functionality should be used to best adapt the client and the server processes. This object is then passed to the adaptable proxy that will be downloaded using the Proxy Loader.

The startAdapting method will create the ORB Adapter module represented by the MobileProxies.ProxyLoader class passing it the identifier of the IDL interface which can be obtained by calling the *id* method of the Helper class generated by the IDL compiler.

The creation of this class and the calling of the startAdapting method have to be done prior to any reference to the server object entering the client's address space.
The second method that interfaces to the client application is the endAdapting method. This method will effectively end the establishing of all the required objects and hence the adaptation process and will open the door for the client application to start using the server object (in an adaptive way).

4.2.2 Class MobileProxies.MobileProxyFactory

This class is used to provide a front end to the ORB. It constitutes the ORB Adapter module that is part of the Client MPS and provides the ORB with means to synchronize the whole adaptation process by signaling the point in time when a reference to the server object enters the client address space. At this point, the ORB will effectively request a proxy object to be created, leaving the decision as to how to do that to the implementation of this class.
This was made possible by allowing this class to inherit from IE.Iona.OrbixWeb.Features.ProxyFactory class, which provides a certain method called *New* that is called by the ORB whenever a proxy object is needed. It is then all left to the particular implementation of the method. A description of the IE.Iona.OrbixWeb.Features.ProxyFactory can be found in [ORG 1998].
Currently, this method uses the Proxy Loader module to download an adaptable proxy that is practically a subclass of the default proxy provided by the IDL compiler.

4.2.3 Class MobileProxies.ProxyLoader

This class implements the Proxy Loader module in the Client MPS. It is mainly used by the previous two classes, where it provides a method for each. The first method is called *sendURL* and is used to by the ClientEnvAdapter to send the Decision object to the server side. The second is called the *downloadProxy* and is used by the MobileProxyFactory to download an adaptable proxy.

The sendURL method uses the ORB. It constructs an org.omg.CORBA.Request object and populates it with the necessary information, including the URL of the adaptable proxy represented as a string.

The downloadProxy method uses the Class Loaders module in downloading the adaptable proxy over a URL connection.

It is important to note that unless the string representing the URL of the adaptable proxy is set before hand (this is normally achieved by calling this method before the sendURL method), an exception will be thrown.

4.2.4 Class MobileProxies.ServerEnvAdapter

This class is a subclass of the IE.Iona.OrbixWeb.Features.Filter class [ORG 1998], which is the parent class of all the Request level interceptors, or otherwise known as Filters in the OrbixWeb terminology. More about Filters can be found in [OPG 1998].

The class is part of the Server Adapter module. The server does not have to call any methods on this object, in fact all what the server has to do is to create an instance of this class.

Being a Filter, this class can intercept all the requests arriving at this server process. It does that for the purpose of trapping the sendURL method that would have been called by the ProxyLoader at the client side. As soon as this request arrives, the sendURL method of this class is called and the string representing the URL of the adaptable proxy is made available. This informs the server process of the name of the adaptable proxy used in the client process.

4.2.5 Class MobileProxies.ServerIOCallback

This class implements the IE.Iona.OrbixWeb.Features.ioCallback [ORG 1998] interface that gives it the ability to monitor any connections that are established or closed by clients.

This ability is provided by two methods, the *CloseCallBack* and the *OpenCallBack* that are defined in the ioCallBack interface.

The current implementation provided by MP v1.0 install a MobileProxies.DummyTransformer (which will be described next) whenever a client closes its connection with the server process.

The class is also part of the Server Adapter module.

4.2.6 Class MobileProxies.DummyTransformer

This class is a Message level interceptor that simply does nothing to the incoming or the outgoing data. In OrbixWeb, Message level interceptors are called Transformers and they are subclasses of the IE.Iona.OrbixWeb.Features.IT_Transformer class [ORG 1998]. These Transformers are called by the ORB just prior to the transmission of the byte array representing the marshaled request and just at the arrival of that array at the other side.

This provides for the accessibility that is required to control the data traffic in a way that achieves adaptability (section (1.2).)

However, there can only be one Transformer installed in a process at any moment in time limiting the prospect of multithreading. This will limit the first solution to the concurrency problem as mentioned in section (3.2.5) to the creation of different process each to deal with a single client as opposed to the spawning of threads.

This problem was left unsolved in MP v1.0, and therefore this class was built to remove any specialization caused by the previous client and so allowing for different clients to queue at the server process.

4.2.7 Class MobileProxies.MultiClassLoader

This is an abstract class that inherits from the java.lang.ClassLoader class and so allowing the definition of a new way for loading classes into the JVM. The class was defined as abstract to give the sub classes a free implementation of how the actual class

bytes will be provided and from which resources. Apart from overriding the *loadClass* method of the java.lang.ClassLoader class, it provides one abstract method that is implemented by its subclasses, that is the *loadClassBytes* method.

When attempting to load a class, a subclass will look first in a locally stored pool of classes using a Hashtable. This technique was employed to speed up the loading process. If the required class was not found, or in other words, if it was not loaded previously, it looks in the list of system classes in case it is a part of the standard JDK. Finally, if everything fails, it will use the loadClassBytes method.

The class also provides a couple of other methods that are used in putting the class names into an understandable format for this class.

Finally, this class is part of the Class Loaders module. Both the client and the Server MPS require it.

4.2.8 Class MobileProxies.FileClassLoader

This is a subclass of the MultiClassLoader that allows classes to be loaded into the JVM directly from the local file system. It is needed mainly by the Server Adapter, and particularly, by the ServerEnvAdapter class since the URL of the adaptable proxy employed remains anonymous until the Server MPS receives it from the Client MPS.

It defines the abstract method loadClassBytes inherited from its superclass such that the bytes are read from a java.io.FileInputStream object. This gives the class a local dimension.

The class falls naturally into the Class Loaders module but it's only part of the Server MPS.

4.2.9 Class MobileProxies.URLClassLoader

This is the opposite of the previous class allowing class bytes to be downloaded across the network. It provides the mechanism with which all the classes needed by the Client MPS can be downloaded. It is also an instance of the MultiClassLoader with a URL-connection-based implementation of the loadClassBytes method.

The URLClassLoader has to consider one important issue, i.e. security. This is necessary due to the code mobility and the threats associated with it. Therefore, digital signatures

were used in authenticating all the downloaded classes having got the public key of the server process in advance.
The class is the last entity in the Class Loaders module. However, it is only part of the Client MPS.

The nine classes just described do not provide any functionality that will achieve adaptability. In fact they only provide the basic tools used in installing the different functionality choices employed in modifying the data bytes.
The size of the nine classes does not exceed 19 KB of executable class files that are distributed between the client and the server processes.

4.2.10 Class MobileProxies.PrivateKeys

The PrivateKeys class contains the private and secret keys necessary for signing and decrypting any data sent to other process.
It is part of the Client and the Server MPS and it is used directly by the ClientEnvAdapter and the ServerEnvAdapter.

4.2.11 Interface MobileProxies.Adaptable

All the adaptable proxies that are part of the application package implement this interface. It defines couple of methods called adaptClient and adaptServer that will be invoked by the Client MPS and the Serve MPS respectively.
The methods when implemented will carry the adaptability semantics at an abstract level allowing for the functionality objects to accomplish the low-level details.

4.3 The Functionality Tree Implementation

The functionality tree designed and described in section (3.4.1) was implemented at different levels in a way that allowed for new classes to be added in the future.
The top two levels, by definition, are part of the application package that uses the MP system. Therefore, these will not be considered as part of the MP system, although they

still remain part of the functionality tree. The description of these two top levels will follow in the next chapter, where an example application was used to evaluate the system. Meanwhile, emphasis will be on the two low levels, i.e. the Interceptors and the Functionality levels.

4.3.1 The Interceptors Level

The Interceptors level was implemented in the **MobileProxies.interceptors** package. The interceptors are subclasses of the **IE.Iona.OrbixWeb.Features.IT_Transformer** class [ORG 1998]. In other words, they are the Transformers of OrbixWeb described in [OPG 1998], but each with a specialized *transform* method that overrides the superclass method.

The current version of the MP system provides five Transformers in this package. These are:

1. The **MobileCompressionTransformer,** which compresses and decompresses the data bytes.
2. The **MobileSignatureTransformer,** which signs and verifies the data bytes.
3. The **MobileSymmEncryptionTransformer,** which encrypts and decrypts the data bytes symmetrically.
4. The **MobileAsymmEncryptionTransformer,** which encrypts and decrypts the data bytes asymmetrically.
5. The **MobileMessageDigestTransformer,** which adds and removes the message digest of the data bytes.

These Transformers will be installed as required by the adaptable proxies and they will apply the functionality they represents to the data traffic flowing to and from the process. Because the number of these Transformers that can be installed in a process is limited to only one, the actual modification of the data bytes will not take place here in this level.

Rather these Transformers will *own* and use another set of objects, namely that described in the next level, and known as the functionality objects.

4.3.2 The Functionality Level

This level was implemented as a set of interfaces, each of which defined certain functionality. The choice of interfaces was made to facilitate populating this level with as many objects implementing the interfaces as possible. It also leaves these implementations open.

These interfaces are part of the **MobileProxies.interfaces** package. The current version of the MP system supports five interfaces. These are:

1. The **I_CompressionTransformer,** which defines the compression functionality.
2. The **I_SignatureTransformer,** which defines the digital signatures functionality.
3. The **I_SymmEncryptionTransformer,** which defines the symmetric encryption functionality.
4. The **I_AsymmEncryptionTransformer,** which defines the asymmetric encryption functionality.
5. The **I_MessageDigestTransformer,** which defines the message digests functionality.

Each of these functionality interfaces defines certain methods that have to be implemented by any objects willing to provide that functionality.

For this purpose, five implementation classes were developed. These were:

1. The **MobileDeInflater** class, which implements the I_CompressionTransformer and uses the java.util.zip.Deflater and the java.util.zip.Inflater classes.
2. The **MobileDSA** class, which implements the I_SignatureTransformer and uses the *Secure Hashing Algorithm* (SHA) and the *Digital Signature Algorithm* (DSA).
3. The **MobileDES** class, which implements the I_SymmEncryptionTransformer and uses the *Data Encryption Standard* (DES) with *Electronic Code Book* (ECB) mode of cipher and the *PKCS#5* padding scheme.
4. The **MobileRSA** class, which implements the I_AsymmEncryptionTransformer and *Rivest-Shamir-Adleman* (RSA) algorithm with ECB mode of cipher and PKCS#5 padding.
5. The **MobileCRC32** class, which implements the I_MessageDigestTransformer and uses the java.util.zip.CRC32 class that computes the CRC-32 checksum.

These classes are part of the MobileProxies.functionality package.

4.4 The Environmental Repository and The Decision Objects

Five classes were developed as stub implementation of these two entities. The Environmental object was left out as this is only useful in the first step explained in section 3.2.1 as part of the client-repository negotiation process, which is not considered to be part of the mechanism the MP system provides.

The classes were defined in the MobileProxies.repository package.

4.4.1 The Decision Object

This object is represented by three classes. These are:

1. The **SignedURL** class, which represents a signed URL object.
2. The **OtherIdentity** class, which holds necessary information about the other process that this particular process is interacting with. In MP v1.0, this class holds the public keys of the other party that are necessary for encrypting and verifying any data received from there.
3. The **DecisionObject** class, which contains an array of SignedURL objects representing the code that will be downloaded from the server process to the Client MPS. It also contains an OtherIdentity object representing the server object.

   It is worth noting that this same class was used also by the ServerEnvAdapter to obtain the necessary information about the client reflected in the OtherIdentity object. The SignedURL array however was taken as null.

4.4.2 The Environmental Repository

This repository is represented by the remaining two class:

4. The **KeyGenerator** class, which generates any necessary keys, embedded in the DecisionObject and that represents a process.
5. The **DecisionObjectGenerator** class, which is the main decision-generating entity. It generates the DecisionObject and serializes it to a file to be read by the ClientEnvAdapter.

4.5 The Runtime Interaction

The Client application starts running the MP system by creating an instance of the **MobileProxies.ClientEnvAdapter** class and calling the startAdapting method. This creation of the ClientEnvAdapter causes it also to read the **MobileProxies.repository.DecisionObject** object from a file called the **Client_Decision.object**.

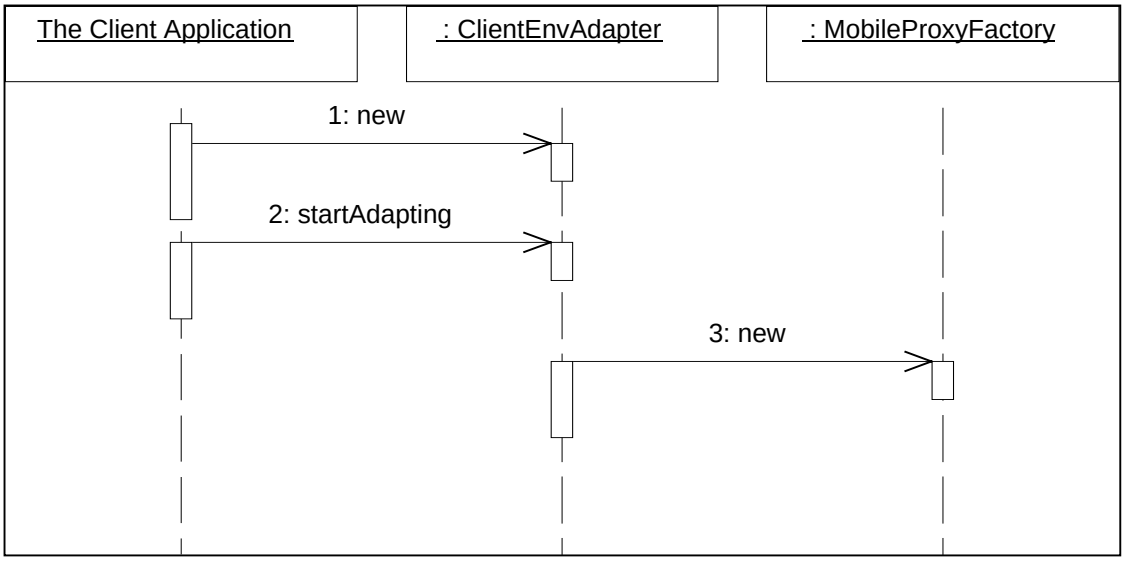

Figure 4.1: Runtime Interaction 1

This is just a stub implementation of the actual negotiation process that will occur between the ClientEnvAdapter and the Environmental Repository.

The ClientEnvAdapter will also read the **MobileProxies.PrivateKeys** object to determine the private and secret keys belonging to the client application. It will then create a **MobileProxies.MobileProxyFactory** object as soon as the startAdapting method is invoked, supplying it with the PrivateKeys and the DecisionObject objects.

This point in time is shown in figure 4.1 below. The MP system now is ready to adapt the client application as soon as a proxy to the server object is required. This requirement may be initiated by invoking the bind method of the Helper class belonging to the IDL interface of the server object. Alternatively, a reference to that object may enter the

address space of the client application through whatever means (like for example, the return result of calling a method on another interface).

The next step follows with the client application binding to the server object. This causes the ORB to call a special method in the MobileProxyFactory object which will requires the creation of a proxy object (called stubs in the CORBA terminology.) The MobileProxyFactory object will act by creating a **MobileProxies.ProxyLoader** object and invoking a special method that will download an adaptable proxy from the server site using a **MobileProxies.URLClassLoader** object.
As soon as the proxy is downloaded and created, a reference will be available and both the ClientEnvAdapter object and the ORB will be informed of it. Figure 4.2 depicts the last step.

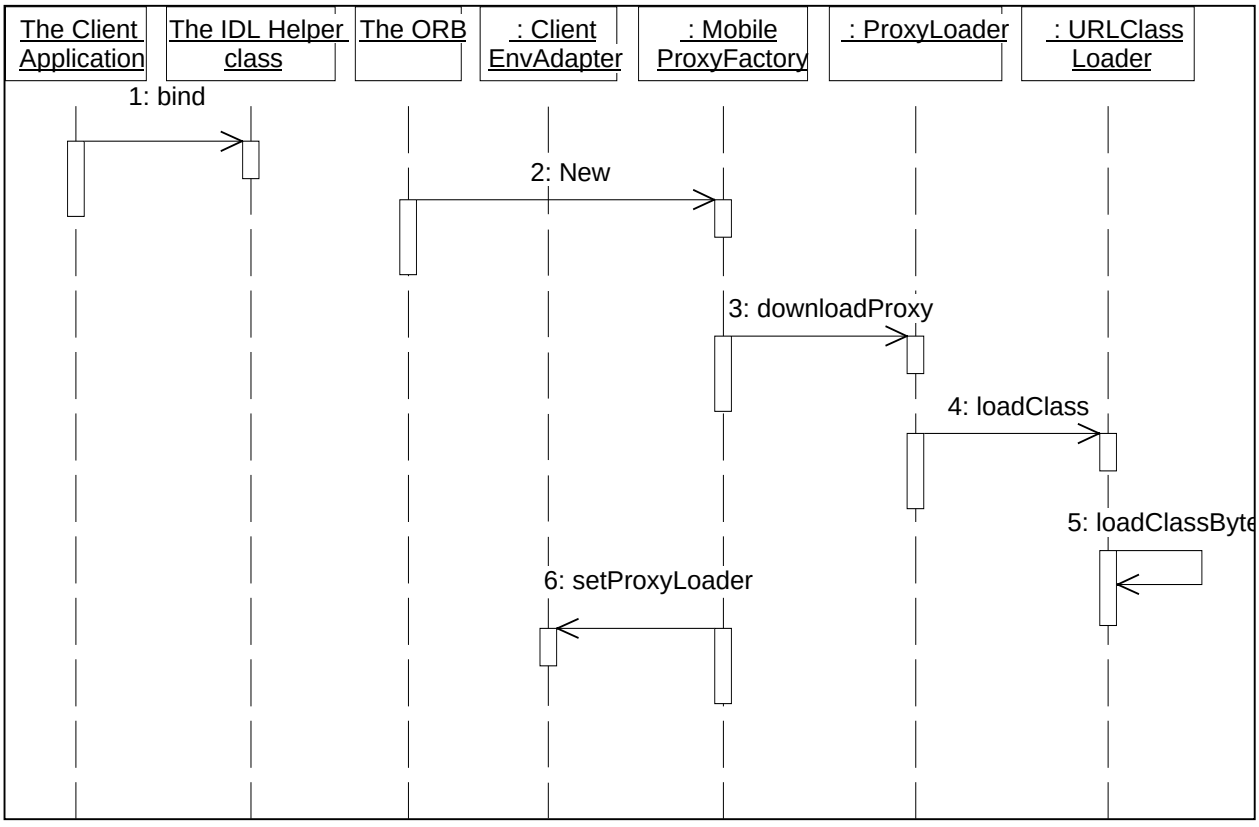

Figure 4.2: Runtime Interaction 2

After the proxy object has been created in the client process, the client application will call the last API, which will completely establish the adaptability requirements between the client and the server processes. Invoking the endAdapting method of the ClientEnvAdapter object will initiate a number of invocations as follows: The ClientEnvAdapter will invoke the sendURL method of the ProxyLoader object, which will in turn use the ORB in sending a request to the server process with the same method name. This will be intercepted at the Request level by the ServerEnvAdapter object, which is created as soon as the ORB starts the server process. The request will also carry a string representing the URL of the adaptable proxy that has been decided for the client.

This achieves the purpose of section (3.2.4), where the importance of distributing the knowledge of the outcome of the decision-making process in the Environmental Repository was emphasized.
The above sequence of invocations is illustrated in figure 4.3.

The ClientEnvAdapter and the ServerEnvAdapter then will invoke the corresponding *adapt* methods in the **Adaptable** proxy, therefore initiating the creation and installation of all the necessary interceptors and functionality objects.
The above description highlighted the major steps that the MP system follows up to the point where the adaptable proxy is used.

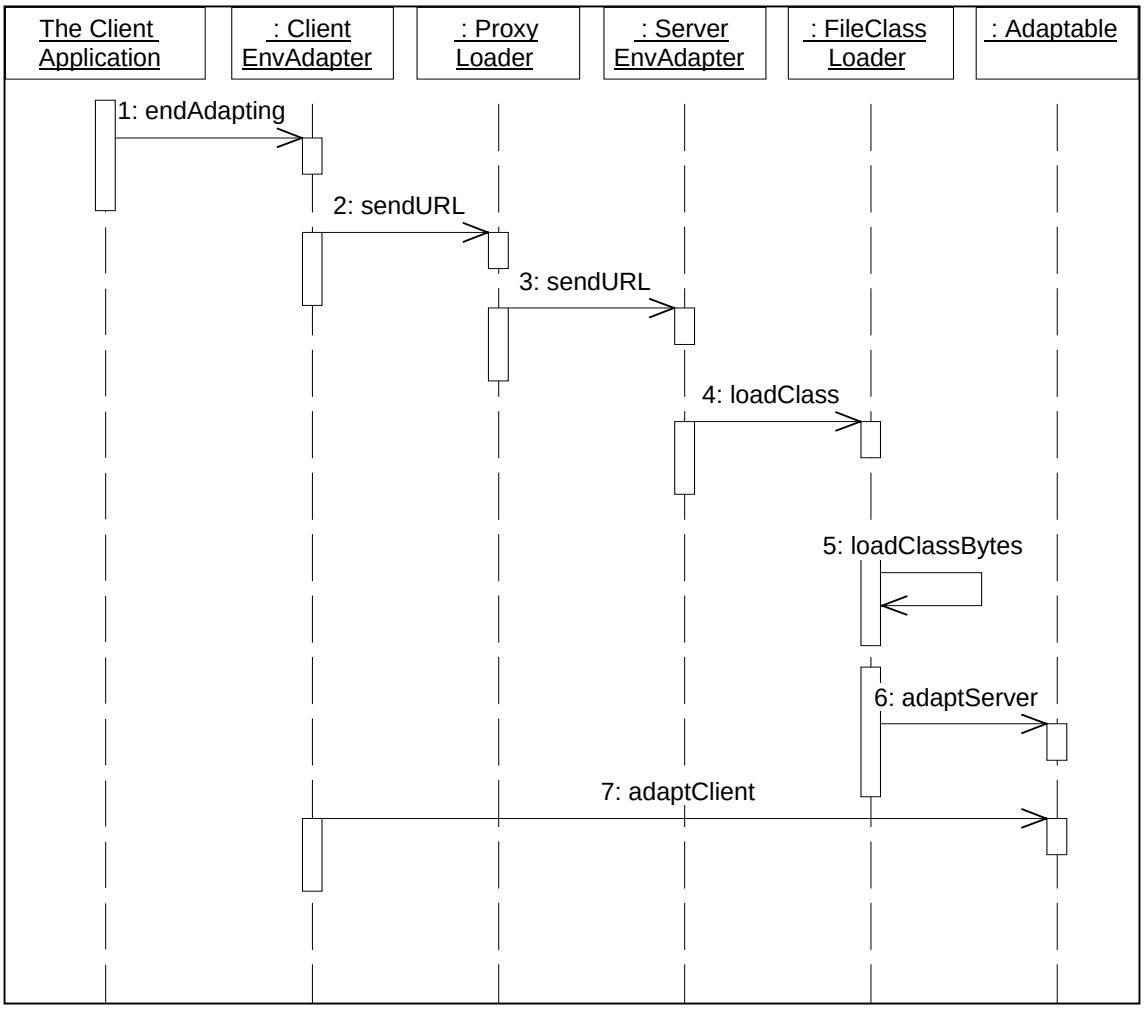

Figure 4.3: Runtime Interaction 3

The next step will be completely determined by that specific proxy and how it converts the abstract adaptability semantics into a reality implementation.

The next set of invocations is viewed in figure 4.4 with an example, where the compression functionality is used in realizing adaptability.

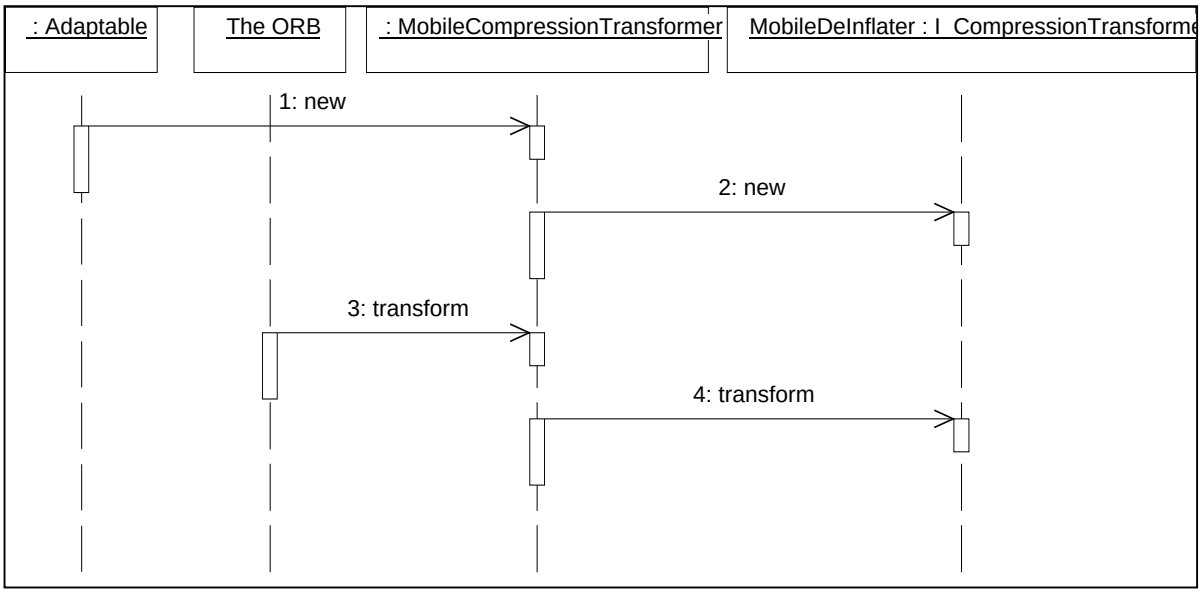

Figure 4.4: Runtime Interaction 4

Assuming that the adaptable proxy carries the compression functionality, the **adapt** methods will cause it to create and install one of the classes belonging to the MobileProxies.interceptor package that compresses and decompresses the data bytes.
Whenever data enters and leaves the process, the ORB will call the **transform** method of that Transformer.
The Transformer then will use another object defined by a MobileProxies.interfaces interface and implemented in any package, including the current MobileProxies.functionality package, to apply the compression functionality.
At this point, all the data flowing between the two processes will be compressed when sent from one process and decompressed at the other. This means that the interaction between the two processes has been specialized to apply certain techniques that will, in the case of compression, increase the performance of the application by minimizing the amount of traffic exchanged over the network.

In the next chapter, an application will be used to evaluate the MP system and the different functionality extensions it currently provides.

Chapter Five
System Evaluation

The evaluation of the MP system was carried out to determine the level at which the system has reached in achieving its intended goals.

As we mentioned in section 1.2, the system was meant to tackle the adaptability issue from the performance and security points of view.

The performance side of the evaluation is meant to show how feasible the system is regarding the overhead incurred in sending the data over the network. Whereas the security goal aims at achieving the three goals of security: Confidentiality, integrity, and authentication.

An application has to be chosen so as the evaluation process would rightly assess the system. Such an application has to test the different functionality extensions that are provided by the system taking into consideration the goals that are sought from every extension.

In the following sections, a brief description of the application package will be provided along with the different proxies it includes. Samples of the results obtained while using different proxies will then be assessed and conclusions drawn as well as determining how successful the system was in a particular functionality. The cascading of different functionality objects was left out, as the number of combinations that could be constructed from these objects would be big enough to make the evaluation process a time-consuming one.

5.1 The Application

5.1.1 Overview

One of the factors that control the adaptability between the different applications is the nature of the application itself and the data it is manipulating. Therefore, it becomes

difficult to decide on a specific application and the way it is distributed between the client and the server environments.
The performance goals are quantity measurements, which dictate that performance evaluation should be carried over a wide range of data sizes ranging from zero-byte data to the excess of Megabytes to determine the gain profile. Yet, experience revealed that the nature of the data is also an issue when it comes to performance. This will be further discussed when performance measurements are given in the next section.

The security demands are also determined by the nature of the data and its sensitivity, but also by the nature of the different media that data traverses. This however is a matter of deciding how important that data is and the appropriate functionality that should be applied to best protect it against attacks from the network that separates the client and the server. This requirement does not impose any performance goals, although it is always nice to have a high performance system. So the only requirements remaining are those determined by the level of security needed.

From the above perspectives as to the different performance and security demands, a file transfer application was chosen. This application is a simple example yet it has the ability to deal with files of different sizes and nature.

The design of the application allows for the client to read a file from the local file system, send it to the server on another host, receive an acknowledgement from that server, and meanwhile, perform all the necessary measurements. The server will just write the file to the its local file system. This is shown in figure 5.1.

5.1.2 The Application Package

The **DataTransfer** package was developed as an implementation of the above outlined application design. The package contained eight classes, five of which were proxies that implemented the MobileProxies.Adaptable interface. Also part of the package was the IDL interface that defines the server object.

Briefly, the three application classes are:

- The **DataTransfer.DataClient** class, which is the CORBA client application.
- The **DataTransfer.DataServer** class, which is the CORBA server application.
- The **DataTransfer.DataImpl** class, which is the implementation class of the IDL interface.

The IDL interface was defined by the **DataTransfer.FileTransfer** interface. Also the inheritance method of interface implementation was used with the DataImpl class extending the **_DataTransferImplBase** compiler-generated class. The delegation method could equally be used.

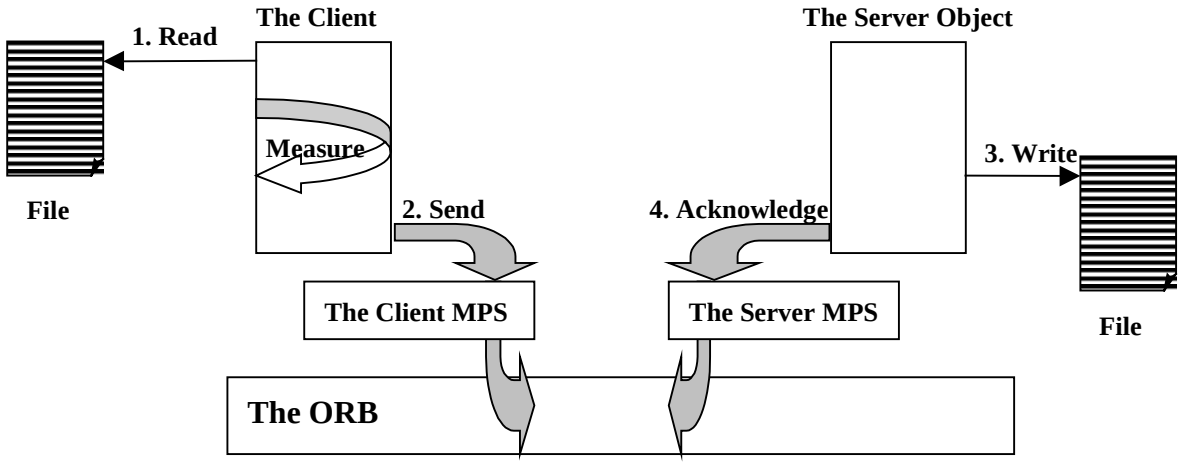

Figure 5.1: The file-sending application.

On the other hand, the five adaptable proxies are:

- The **DataTransfer.MobileCompressionProxy** class, which provides compression semantics.
- The **DataTransfer.MobileSignatureProxy** class, which provides digital signatures semantics.
- The **DataTransfer.MobileSymmEncryptionProxy** class, which provides symmetric encryption semantics.
- The **DataTransfer.MobileAsymmEncryptionProxy** class, which provides asymmetric encryption semantics.

- The **DataTransfer.MobileMessageDigestProxy** class, which provides message digesting semantics.

All these proxies are subclasses of the default proxy generated by the IDL compiler, i.e. the **DataTransfer.FileTransferStub**.

The number of these proxies however should not be considered as a limitation to the total number of proxies that can be developed. Any combination of functionality extensions can be formed as we mentioned earlier. Moreover, with the current version of the MP system, a suitable compiler can be built to generate these proxies automatically from the default one. Although this will make the adaptability process less flexible as the choices of functionality will be limited by the output of that compiler.
The proxies are stored in the file system of the server process and are downloaded, along with the Transformers and the functionality objects, at runtime by the client process using some protocol, like HTTP or FTP, depending on the nature of the server.
In the project, an HTTP server different from the application server was used to provide the mobile code for the client process. The HTTP server was however chosen to be on the same host as the application server.

The system requires from .77 to1.55 Milliseconds to be completely set up with all the Transforms installed and ready for the client to make its invocations. This numbers were calculated by measuring the times required for the startAdapting, the endAdapting, as well as the call to the Constructor of the ClientEnvAdapter class.
For the purpose of measurements, the *java.lang.System.currentTimeMillis* method, which returns the system time in Milliseconds.

The application used three types of files. The first type was a text file with a repetitive nature that was generated “artificially” using a program. These files contained numbers that counted from 0 to 255 and then repeated the format up to the specified size.
The second type of files was a PDF file, which was more natural than the first type. Different files were chopped from one large file according to the required size.

The last type was an image file, with a combination of JPG, GIF, and bitmap-formatted files.
30 files of each type were used with sizes ranging from 0 to 2M bytes. These files helped satisfying the evaluation requirements as they provided a wide range of data sizes as well as a variety of natures.

5.2 Results of the Compression Functionality

5.2.1 Overview

The compression functionality was used to improve the performance level of the application by minimizing the delay while transferring data from one process to another over the network. This delay depends on two components: The processing time, and the transmission time.

The processing time includes any time spent prior to the transmission of the data over the wire, whereas the transmission time is the time spent by the data while travelling over the network. The latter though depends on two factors: The size of the data to be transmitted, and the network speed.
The size is an obvious factor affecting the transmission time linearly. The network speed is a less obvious one, depending on a number of other factors, like the physical bandwidth and the amount of traffic already flowing over the network.
To reduce the transmission time, either the size of data has to be reduced, or the network has to be faster up. Unlike the network speed, which cannot be controlled directly from the application, the data size *can* be controlled and reduced to an optimum level. To do so, compression techniques between the client and server processes have to be employed.

Ideally, If these compression techniques did not take any time, the minimum transmission time would be achieved at the smallest size for the compressed data. However, in reality, compression does take time. This time will add up to the processing time therefore increasing the overall delay and possibly affecting the total performance gain negatively.

Figure 5.2 illustrates the different times spent during the transfer process, including the processing time at both ends, of which the compression time is one factor, and the transmission time suffered over the wire.

At this point a trade off begins to form as to whether use the compression or just send the data directly. This will depend on all the above-mentioned factors as well as on the nature of the data. This nature will determine how well the data is compressed and hence whether or not the compression process was justifiable.

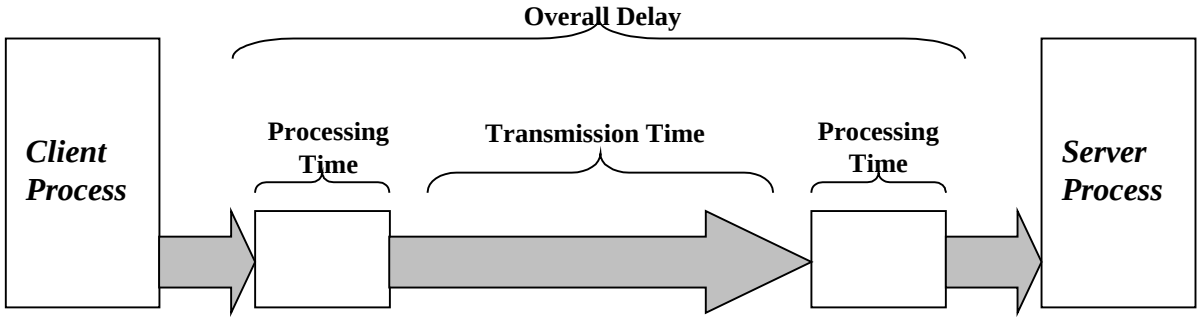

Figure 5.2: Different time delays suffered by the data.

All these factors that are part of the overall delay formula can be described by the Environmental objects. These objects (and hence the factors) will then be compared for different client-server applications and environments and a decision will be reached as whether or not compression techniques should be used.

To evaluate the feasibility of the MP system from the compression point of view, the different files mentioned in the previous section were transmitted from the client running on a Windows NT platform, to the server object running on a Sun Solaris workstation. These however were part of a department-wide local area network, which is considered to be a relatively fast medium. In other words, the transmission time will be small compared to other networks.

The following section will present some of the results that were obtained in chart form. The raw numbers can be found in the Appendix titled Performance Results.

5.2.2 The Results

The first set of measurements was carried out for the PDF-type files. The following chart in figure 5.3 shows a comparison of the time taken by the *sendData* method of the **FileTransfer** interface when invoked by the client, with and without using the compression functionality of the MP system. The chart shows a gain at the point where the size of the file reaches 187.5 KB forward. The gain then is not that high due to the nature of the files which had a compression ratio ranging from 23 to 65 %.

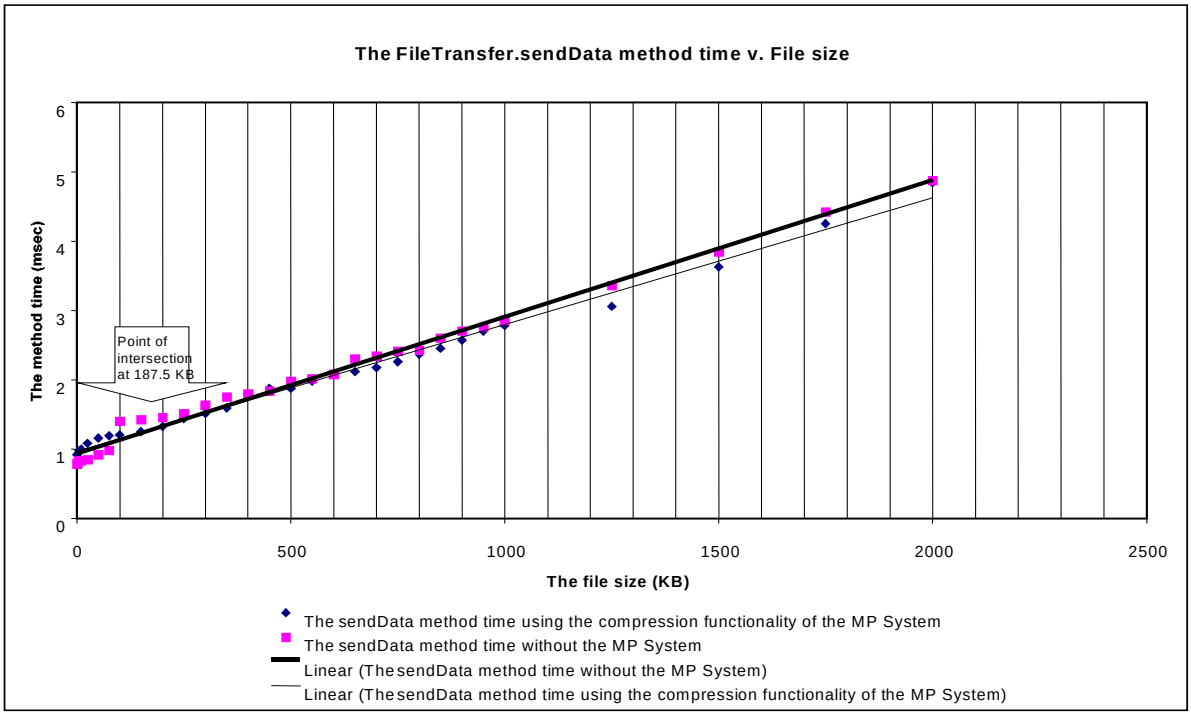

Figure 5.3: The FileTransfer.sendData time with and without the MP system for the PDF files.

The next chart in figure 5.4 shows the profile of the whole DataTransfer.DataClient application time and for the same set of files. It is clear that due to the MP system setup overhead, the gain is occurring at a later point of approximately 1M byte file size.

In addition to these two sets of measurements, a theoretical calculation was made to compare the processing time spent in the compressing/decompressing part, and the transmission time spent over the network. These two elements are compared in figure 5.5. The dark-colour component being the transmission time.

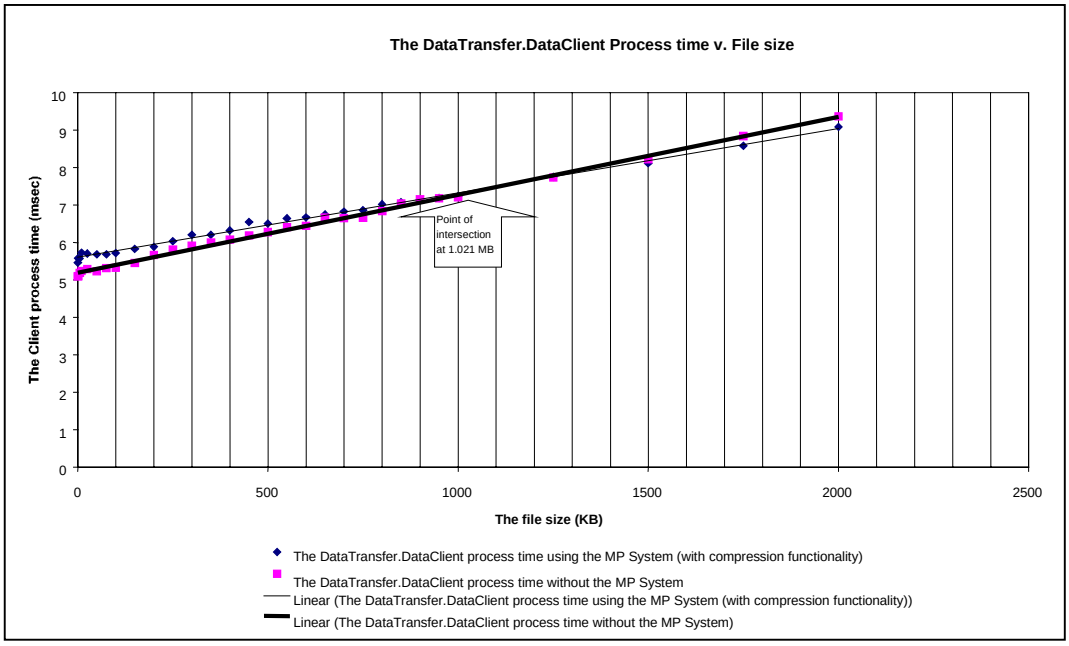

Figure 5.4: The DataTransfer.DataClient application time with and without the MP system for the PDF files.

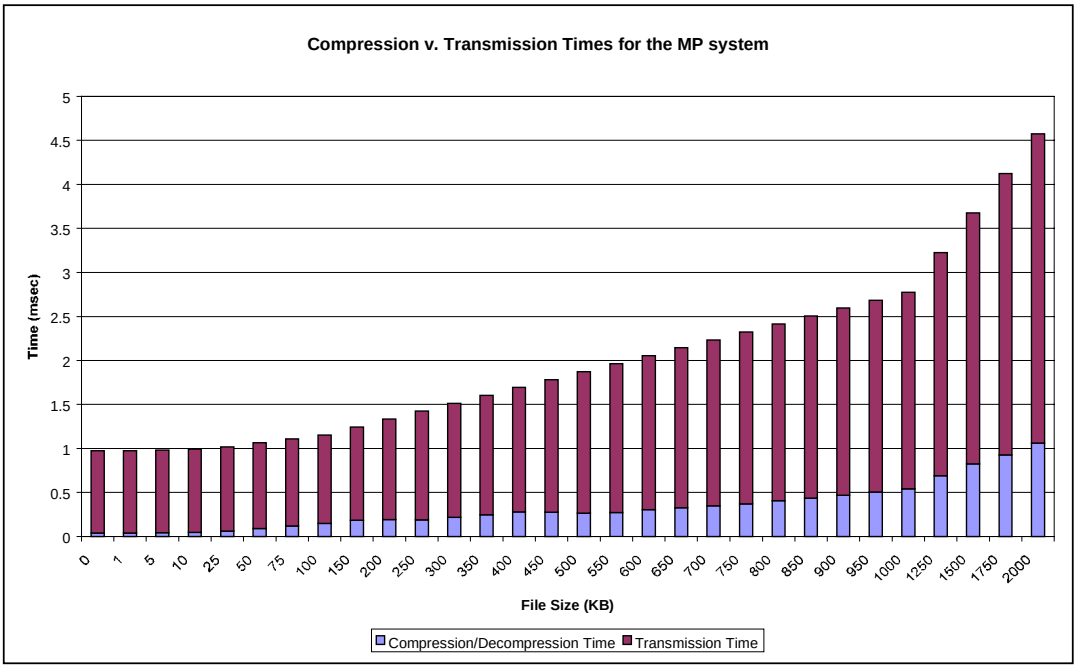

Figure 5.5: The compression time versus the transmission time for the FileTransfer.sendData method using the MP system for the PDF files.

The point to be drawn here is that for slower networks, the transmission time will increase while the compression time will remain the same. So if the data were sent without any compression, the delay would be higher since the time consumed is only composed of the transmission component. In other words, the effect of the slow network will be more on times that are composed of one component (the transmission time) than those composed of two components (the transmission and processing times).

The following table predicts the breakeven point at which the gain occurs for a number of networks, whereas figure 5.6 illustrates the results on a chart. It is worth noting that the measurements were taken on a 10 Mbps network.

Network Speeds v. Breakeven Point for the PDF files using compression

Network	Mbps	Breakeven Point (KB)
Slow modem	0.0024	0
Basic fax/modem	0.0096	0
Fast modem	0.0144	0
Nokia' HSCSD network	0.0576	0
ISDN	0.128	0
Cable modem	0.5	0
T1	1.544	0
Wireless Bridges	2	0
	3	2.5
10Base2	4	15
	6	47
	7.5	82
10BaseT, and Ethernet	10	187.5
	11	388
	11.5	393
Telesat satellite	12	398
	12.5	403
	13	not feasibile
T3	45	not feasibile
SONET	52	not feasibile
Fast Ethernet, 100BaseT, 100BaseT4, 100BaseTX, FDDI	100	not feasibile
ATM, STM-1	155	not feasibile
OC-12	622	not feasibile
Gigabit Ethernet	1000	not feasibile
OC-48	2400	not feasibile
OC-192	9600	not feasibile

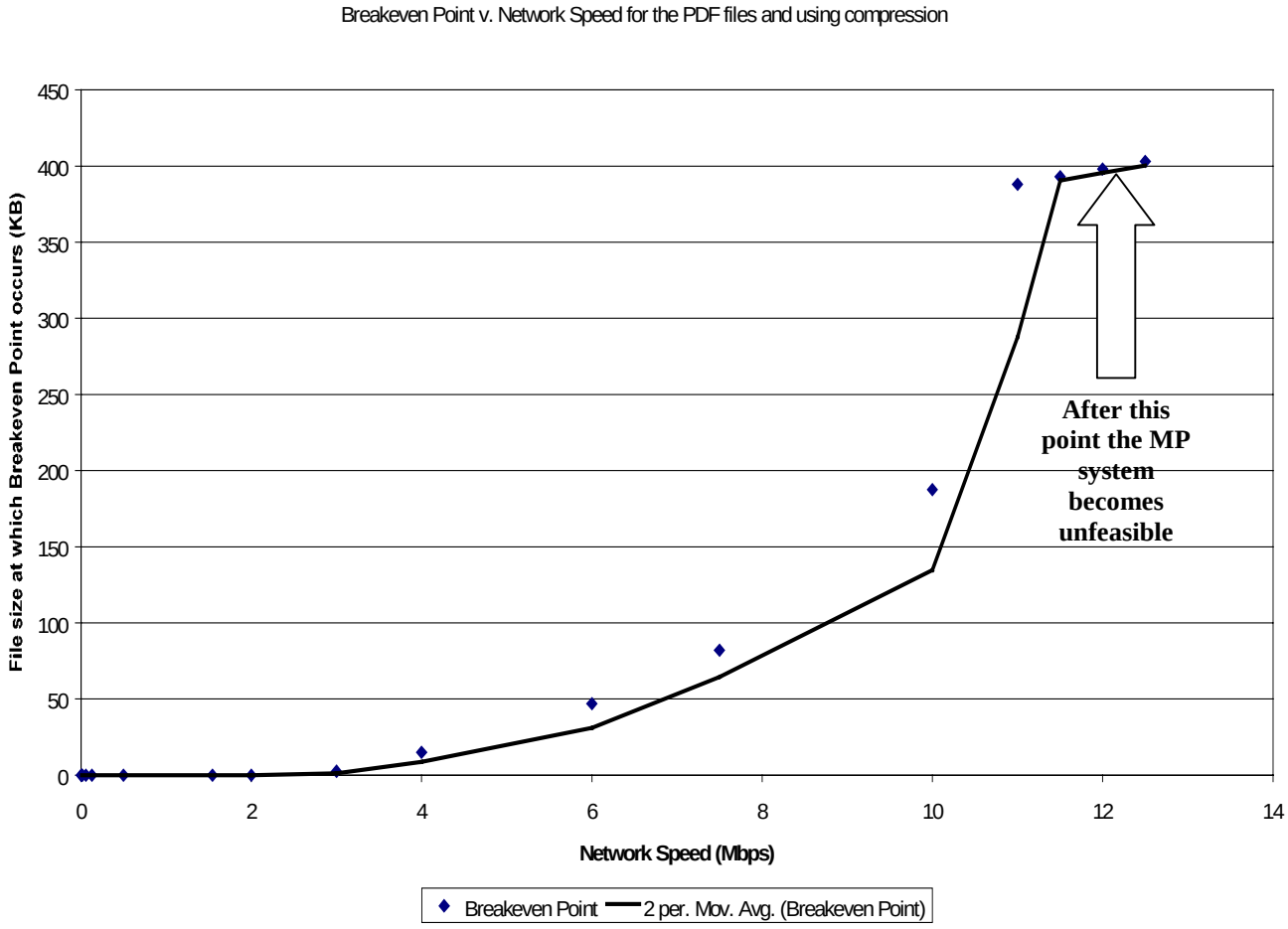

Figure 5.6: The Breakeven point v. Network Speed

The second type of files used was the repetitive-number file. These, owing to their nature, had a higher compression ratio than the previous type with most of the ratios above 90 %. This meant that the transmission time would be more constant because the actual size transmitted is very close across the range of files. Therefore the small files will get the same benefit as the large files. This is in contrast to the previous case, where the transmission time was irregular across the size range, with small files transmitted quicker than the large ones.

Figures 5.7 to 5.9 illustrate the same charts but for the repetitive-number files.

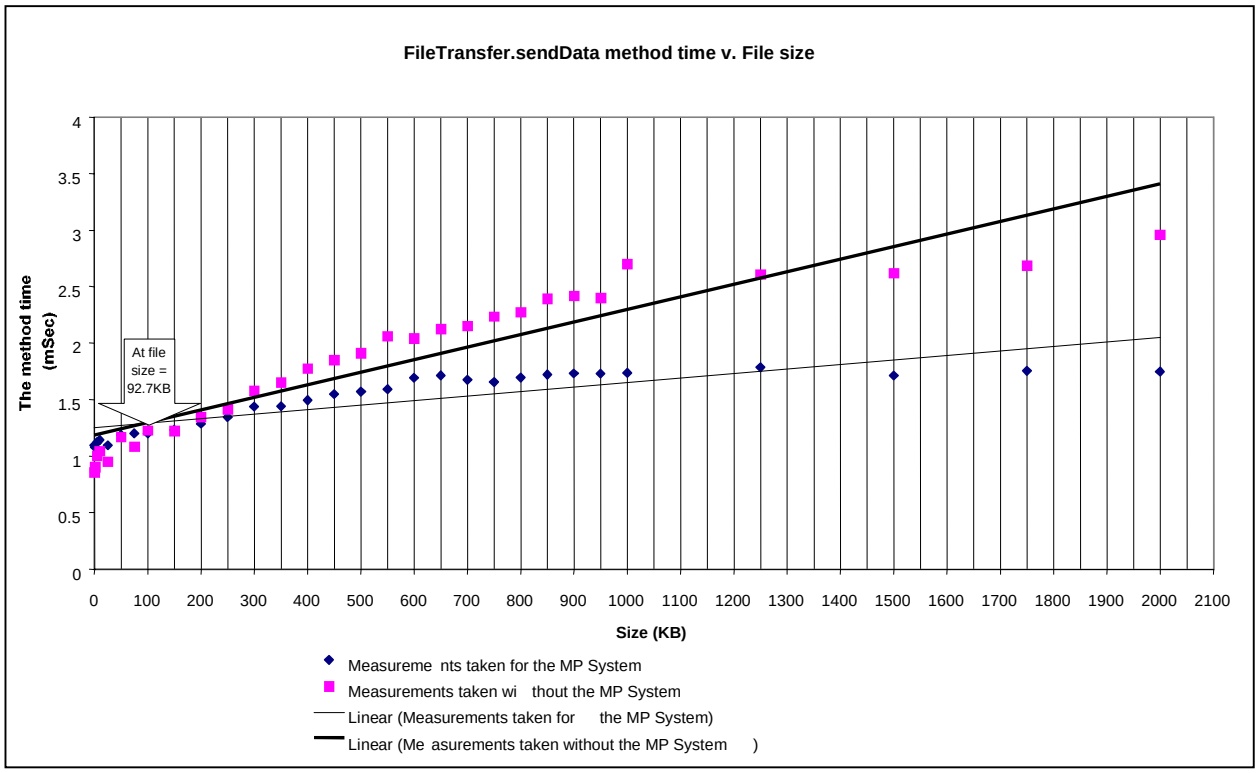

Figure 5.7: The FileTransfer.sendData time with and without the MP system for the repetitive-number files.

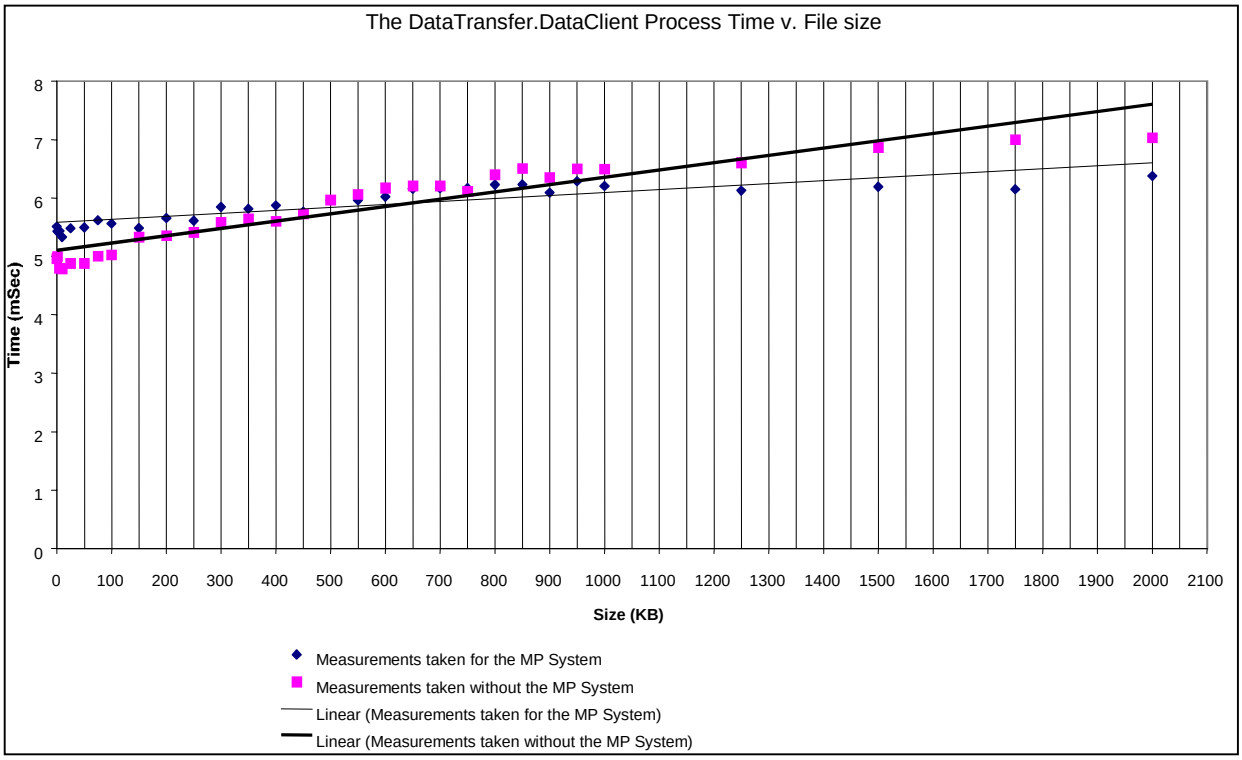

Figure 5.8: The DataTransfer.DataClient application time with and without the MP system for the repetitive-number files.

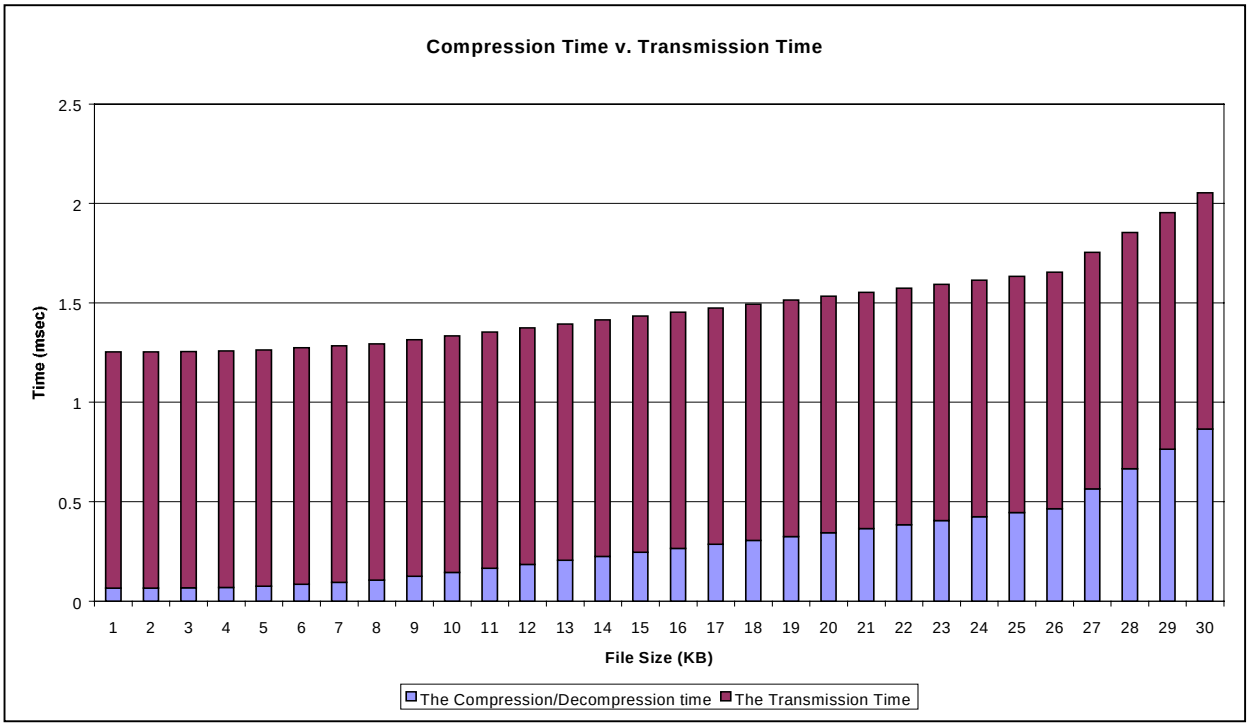

Figure 5.9: The compression time versus the transmission time for the FileTransfer.sendData method using the MP system for the repetitive-number files.

The nature of the image files did not allow them to be compress more than 5 %, which meant that the compression functionality is not suitable for that type of files. Hence adaptability would be achieved by *not* applying the compression functionality.

5.3 Results of the Security Functionality

The security aspect of the system was implemented to cater for application that are separated by untrusted networks as well as protecting any host that is downloading code from hostile actions performed by that code.

All the mobile code downloaded from the server to the client, including the adaptable proxies, the Transformers, and the functionality objects were signed and verified at the

Client MPS before any instances of them were created. These signatures were applied at all times even in the case compression was used.
However, the adaptability goal regarding security was achieved by applying, in addition to digital signatures, encryption (symmetric and asymmetric) and message digests to the file sent from client to the server object.

The digital signatures made sure that the source of the data arriving is authenticated as being the client. Such an authentication would protect against masquerading.
The encryption would be necessary if the data sent is sensitive enough so that no eavesdropping is allowed and the confidentiality of data is preserved.
Finally, message digests are a way of insuring that the data have not been tampered with during the time it's travelling over the network. Thus the integrity of the data is insured.

The performance of these security extensions was measured, except for the message digests, although it is not a goal. However this was done to get some feeling of how expensive these functions are in terms of processing time.
The most expensive one was the asymmetric encryption to the degree that files larger than 450 KB caused the JVM to run out of memory. Less expensive was the symmetric encryption then the digital signatures. The table below gives an idea about the cost of these functions for a sample PDF file of size 500 KB. A complete table can be found in Appendix B.

Security Functionality	The FileTransfer.sendData method time (Millisec.)	The DataTransfer.DataClient process time (Millisec.)
Digital Signatures	10.091	16.53033
Symmetric Encryption	10.809	17.42833
Asymmetric Encryption	55.001	62.586

5.4 Concluding Points

All the measurements were recorded at times when the network is most stable and the amount of traffic is at a minimum. In addition, a number of measurements were taken for each file, with the average being adopted as a representative of that measurement.
It should also be noted that the times were recorded for a dead server process, in which case there is extra time spent by the ORB in starting the process. This choice was preferred to calling the same process each time before it exits so that the worst possible case would be always considered.

It can be claimed that if the appropriate knowledge about the environment, including the state of the network, and the application was made available to the MP system that it can achieve performance gains in the case they are needed.
The system also satisfies security demands depending on the level of security required and the sensitivity of the data sent as well as the nature of the network on which that data travelling. Code mobility however is always made secure.

Chapter Six
Conclusion

The previous chapters demonstrated the different aspects of the adaptability problem in client server environments and a proposal was suggested that provided a solution to this problem in the performance and security contexts.

A survey was then carried out that included a number of technologies related to the goal of the project but that fell short somewhere of achieving that goal.

Therefore a design was laid down for a system that would offer a mechanism for adding extra functionality extensions to the runtime interaction between clients and servers so that the security and communication performance requirements of that interaction would be met.

The design was then implemented and evaluated for a number of functions including, compression, digital signatures, and encryption (symmetric and asymmetric).

In this chapter, a general review of the whole work and the lessons learnt will be presented as well as a perspective to where future work may be directed.

6.1 Review of The Adaptability Problem

The fact that modern distributed systems are composed of a wide range of products and network types meant that these systems should also offer mechanisms to allow for the different products to interact in a way that will satisfy certain application- and environment-level needs.

These needs are often expressed in terms of performance, security, fault tolerance, processing power and other issues that can shape the way applications interact and the outcome of that interaction.

Adaptability is therefore the term that describes how well applications are matched according to some level of acceptance. This level could be determined by certain standards, like the Quality of Service (QoS) standard, or by some user-defined criteria.

Among the different aspects of adaptability, security and communication performance where chosen as the two goals towards which this project was directed.

6.2 Review of the MP System

The MP system was designed as a mechanism that provides secure and performance-enhanced interaction among applications integrated with the Common Object Request Broker Architecture (CORBA) technology.

The system relied on two main concepts: The Proxy principle [Shapiro 1986], and the concept of Interceptors [CORBA/IIOP v2.3 Spec 1999], both of which are used in the CORBA technology.

These two concepts when combined provide a means by which the low level data bytes exchanged by clients and servers may be exposed to the higher levels of applications and hence allowing for some “intelligent” control to be applied. The control means that different requirements of applications from the performance and security points of view are met.

The system consists mainly of five modules distributed between the client and the server. These modules provide the necessary mechanism for a proxy that carries some adaptability semantics to be downloaded from the server’s site and installed in the client’s process.
This proxy will then install an interceptor that exposes the data exchanged between the client and the server, and at the same time, feeding it to one or more functionality objects that are capable of applying certain security- and/or performance-enhancement techniques to the data.

The adaptable proxy is chosen based on a decision taken by a third party, which could be some form of an environmental repository which would hold different environment and application level information for clients and servers.
The security implications rising from the code mobility were taken care of by using digital signatures.

The system was implemented as MP v1.0 and represented by a number of packages, of which the MobileProxies package holds the core system classes.
Java was the language of implementation and OrbixWeb from Iona Technologies was chosen as the integration framework for the system.

6.3 Review of the Evaluation Results

To allow for the MP system to be evaluated and assessed, an application was developed that allowed a CORBA client to send a file as a byte array across the ORB to an IDL-defined server object.
This application was used then to measure the performance of the different functionality extensions provided under the two major goals of the project, i.e. communication performance and security.

The communication performance results revealed a gain that was achieved at a certain file size, although this gain was very much dependent, among other things, on the nature of files transmitted which determined their ability to be compressed.

A comparison was then carried out to determine the gaining point, or otherwise known as the *breakeven* point, for a number of data transmission speeds varying from the slow 2400 bps modem, to the 9600 Mbps OC-192 network (figure 5.6 and the table preceding it.)
This comparison revealed the fact that the MP system offered breakpoints as low as 0 byte files for the slow networks, but then it became unfeasible for the highly fast ones.

These results emphasize the need for clients and servers to negotiate their environment requirements and the nature of the data that is being exchanged which is application-dependent.

The security measurements were solely meant to give an idea of how expensive (in terms of time delay) these functions are.
The results were according to the expected profile that states that digital signature are the least expensive whereas the asymmetrical encryption costs were the highest.

6.4 Future Work

The MP system is one step towards the achievement of adaptability among client-server applications by improving the communication performance and enhancing security measures.
However, further research can be carried out both horizontally and vertically. Horizontally would be to look at other aspects of adaptability like fault tolerance, where a server may or may not use a replicated process depending on the client's fault tolerance needs.
On the other hand, extra vertical research may be carried out, by looking at other entities interacting with the MP system. Of these is the Environmental Repository and the Environmental Objects (as suggested in section 3.1.3) both of which play a major role in the decision-making process that will eventually determine the adaptability semantics used in a particular interaction between the client and the server.

Further research would also include the cascading of different functionality-manipulating objects allowing for a variety of extensions to be available and hence offer extra adaptability semantics.

Appendix
Performance Results

A.1 Compression Performance Results

The following tables show the performance results that were obtained for the compression functionality of the MP v1.0 system.

A.1.1 The PDF Files

The Mobile Proxies System Measurements for the Compression Functionality
(Using java.util.zip.Deflater.BEST_SPEED and java.util.zip.Deflater.DEFAULT_STRATEGY variables)

					FileTransfer.sendData method time				The DataTransfer.DataClient		
File (.pdf)	Normal Size (KB)	Request buffer (Bytes)	Compressed buffer (Bytes)	Compression ratio %	Tx1	Tx2	Tx3	Txav	T1	T2	T3
1	0	116	89	23.28	0.912	0.902	0.932	0.915333	5.369	5.519	5.49
2	1	1140	769	32.54	0.912	0.972	0.912	0.932	5.619	5.529	5.58
3	5	5236	2603	50.29	0.992	0.961	0.982	0.978333	5.58	5.619	5.519
4	10	10356	5187	49.91	1.012	0.982	1.002	0.998667	5.76	5.449	5.98
5	25	25716	10521	59.09	1.142	1.032	1.082	1.085333	5.78	5.799	5.533
6	50	51316	19116	62.75	1.082	1.222	1.172	1.158667	5.539	5.779	5.73
7	75	76916	27506	64.24	1.213	1.202	1.172	1.195667	5.69	5.73	5.629
8	100	102516	36132	64.75	1.232	1.242	1.152	1.208667	5.789	5.719	5.64
9	150	153716	62946	59.05	1.292	1.202	1.262	1.252	5.86	5.679	5.94
10	200	204916	105180	48.67	1.312	1.362	1.312	1.328667	5.909	6.02	5.719
11	250	256116	153462	40.08	1.512	1.412	1.413	1.445667	6	6.13	5.956
12	300	307316	184980	39.81	1.502	1.463	1.583	1.516	6.221	6.24	6.16
13	350	358516	214450	40.18	1.603	1.563	1.613	1.593	6.22	6.12	6.271
14	400	409716	244971	40.21	1.873	1.652	1.853	1.792667	6.171	6.461	6.32
15	450	460916	292243	36.60	1.863	1.933	1.843	1.879667	6.481	6.47	6.681
16	500	512116	343458	32.93	1.883	1.893	1.853	1.876333	6.451	6.741	6.32
17	550	563316	386485	31.39	1.964	2.014	1.973	1.983667	6.531	6.601	6.791
18	600	614516	415477	32.39	2.154	1.993	2.123	2.09	6.681	6.631	6.701
19	650	665716	450626	32.31	2.204	2.074	2.083	2.120333	6.881	6.611	6.742
20	700	716916	485563	32.27	2.224	2.163	2.153	2.18	7.001	6.772	6.701
21	750	768116	520428	32.25	2.223	2.274	2.294	2.263667	6.781	6.952	6.852
22	800	819316	548024	33.11	2.384	2.394	2.323	2.367	7.102	7.101	6.861
23	850	870516	578667	33.53	2.515	2.404	2.444	2.454333	7.052	7.001	7.172
24	900	921716	609023	33.93	2.605	2.605	2.514	2.574667	7.061	7.172	7.162
25	950	972917	636052	34.62	2.764	2.744	2.594	2.700667	7.242	7.201	7.121
26	1000	1024116	664396	35.12	2.654	2.664	3.045	2.787667	7.191	7.242	7.272
27	1250	1280116	817492	36.14	3.105	3.065	3.005	3.058333	7.633	7.722	7.892
28	1500	1536116	978322	36.31	3.606	3.655	3.635	3.632	8.163	8.063	8.113
29	1750	1792116	1158857	35.34	4.216	4.287	4.256	4.253	8.574	8.574	8.594
30	2000	2048116	1318920	35.60	4.877	4.868	4.798	4.847667	9.095	9.164	9.004

Notes:
1- All times in msec unless otherwise mentioned
2- Measurements taken at 2.25 AM, 10 Aug.1999

Measurements taken without the Mobile Proxies System

File (.pdf)	Size (KB)	The FileTransfer.sendData method time				The DataTransfer.DataClient process time			
		Tx1	Tx2	Tx3	Txav	T1	T2	T3	Tav
1	0	0.781	0.791	0.791	0.787667	5.117	5.117	5.037	5.090333
2	1	0.802	0.801	0.802	0.801667	5.078	5.208	5.007	5.097667
3	5	0.831	0.861	0.821	0.837667	5.087	5.267	5.207	5.187
4	10	0.821	0.871	0.822	0.838	5.267	5.238	5.207	5.237333
5	25	0.832	0.851	0.871	0.851333	5.558	5.127	5.177	5.287333
6	50	0.942	0.901	0.921	0.921333	5.268	5.188	5.218	5.224667
7	75	0.982	0.971	0.991	0.981333	5.308	5.287	5.348	5.314333
8	100	1.051	1.542	1.623	1.405333	5.348	5.288	5.338	5.324667
9	150	1.201	1.763	1.312	1.425333	5.468	5.418	5.477	5.454333
10	200	1.513	1.251	1.613	1.459	5.989	5.467	5.538	5.664667
11	250	1.502	1.462	1.582	1.515333	5.969	5.929	5.528	5.808667
12	300	1.532	1.803	1.583	1.639333	5.789	6.059	5.889	5.912333
13	350	1.763	1.813	1.693	1.756333	6.188	5.999	5.798	5.995
14	400	1.793	1.853	1.753	1.799667	6.139	5.929	6.159	6.075667
15	450	1.953	1.773	1.792	1.839333	6.629	6.099	5.838	6.188667
16	500	2.113	1.942	1.893	1.982667	6.449	6.269	6.118	6.278667
17	550	1.973	2.073	2.012	2.019333	6.57	6.169	6.5	6.413
18	600	2.043	2.103	2.093	2.079667	6.599	6.369	6.369	6.445667
19	650	2.243	2.423	2.254	2.306667	6.529	6.789	6.66	6.659333
20	700	2.314	2.274	2.433	2.340333	6.63	6.82	6.51	6.653333
21	750	2.554	2.363	2.323	2.413333	6.75	6.649	6.569	6.656
22	800	2.474	2.423	2.394	2.430333	6.87	6.7	6.94	6.836667
23	850	2.574	2.653	2.593	2.606667	6.92	6.93	7.27	7.04
24	900	2.814	2.613	2.694	2.707	7.16	7.07	7.241	7.157
25	950	2.764	2.834	2.694	2.764	7.13	7.25	7.15	7.176667
26	1000	2.854	2.854	2.874	2.860667	7.14	7.25	7.23	7.206667
27	1250	3.335	3.335	3.414	3.361333	7.721	7.701	7.792	7.738
28	1500	3.865	3.826	3.846	3.845667	8.332	8.092	8.262	8.228667
29	1750	4.446	4.377	4.447	4.423333	8.913	8.82	8.803	8.845333
30	2000	4.797	4.967	4.857	4.873667	9.424	9.293	9.394	9.370333

Notes:

1- All times in msec unless otherwise mentioned

2- Measurements taken at 4.30 AM, 10 Aug.1999

Estimated Compression/Decompression time

File (.pdf)	Normal Size (KB)	Compressed buffer (KB)	T_{esm1}	T_{xav}	$T_{comp1ABS}$
1	0	0.086914063	0.936074	0.9734	0.037326
2	1	0.750976563	0.937402	0.9752	0.037798
3	5	2.541992188	0.940984	0.9824	0.041416
4	10	5.065429688	0.946031	0.9914	0.045369
5	25	10.27441406	0.956449	1.0184	0.061951
6	50	18.66796875	0.973236	1.0634	0.090164
7	75	26.86132813	0.989623	1.1084	0.118777
8	100	35.28515625	1.00647	1.1534	0.14693
9	150	61.47070313	1.058841	1.2434	0.184559
10	200	102.7148438	1.14133	1.3334	0.19207
11	250	149.8652344	1.23563	1.4234	0.18777
12	300	180.6445313	1.297189	1.5134	0.216211
13	350	209.4238281	1.354748	1.6034	0.248652
14	400	239.2294922	1.414359	1.6934	0.279041
15	450	285.3935547	1.506687	1.7834	0.276713
16	500	335.4082031	1.606716	1.8734	0.266684
17	550	377.4267578	1.690754	1.9634	0.272646
18	600	405.7392578	1.747379	2.0534	0.306021
19	650	440.0644531	1.816029	2.1434	0.327371
20	700	474.1826172	1.884265	2.2334	0.349135
21	750	508.2304688	1.952361	2.3234	0.371039
22	800	535.1796875	2.006259	2.4134	0.407141
23	850	565.1044922	2.066109	2.5034	0.437291
24	900	594.7490234	2.125398	2.5934	0.468002
25	950	621.1445313	2.178189	2.6834	0.505211
26	1000	648.8242188	2.233548	2.7734	0.539852
27	1250	798.3320313	2.532564	3.2234	0.690836
28	1500	955.3925781	2.846685	3.6734	0.826715
29	1750	1131.696289	3.199293	4.1234	0.924107
30	2000	1288.007813	3.511916	4.5734	1.061484

T_{esm1} Estimated time for FileTransfer.sendData method without the MP System for the compressed buffer size.

T_{xav} Estimated time for FileTransfer.sendData method using the MP System and for the normal file size.

$T_{comp1ABS}$ Estimated compression/decompression time ($T_{esm1} - T_{xav}$).

A.1.2 The Repetitive-Number Files

Measurements taken for the Mobile Proxies System at 4.00 PM (Saturday)

All times in mSecs unless otherwise mentioned

					The FileTransfer.sendData method				The DataTransfer.DataClient application			
File	Size (KB)	Request buffer (Bytes)	Compressed buffer (Bytes)	Compression ratio %	Tx1	Tx2	Tx3	Txav	T1	T2	T3	Tav
1	0	116	104	10.34	1.111	1.112	1.072	1.098333333	5.678	5.528	5.318	5.508
2	1	1140	386	66.14	1.072	1.091	1.072	1.078333333	5.608	5.368	5.298	5.424666667
3	5	5236	424	91.90	1.072	1.192	1.102	1.122	5.388	5.448	5.458	5.431333333
4	10	10356	471	95.45	1.092	1.212	1.121	1.141666667	5.368	5.237	5.377	5.327333333
5	25	25716	543	97.89	1.121	1.102	1.072	1.098333333	5.508	5.518	5.408	5.478
6	50	51316	655	98.72	1.122	1.322	1.121	1.188333333	5.368	5.668	5.448	5.494666667
7	75	76916	776	98.99	1.322	1.162	1.122	1.202	5.679	5.698	5.478	5.618333333
8	100	102516	869	99.15	1.212	1.202	1.192	1.202	5.528	5.458	5.698	5.561333333
9	150	153716	1076	99.30	1.222	1.332	1.141	1.231666667	5.488	5.598	5.367	5.484333333
10	200	204916	1267	99.38	1.231	1.302	1.332	1.288333333	5.718	5.538	5.709	5.655
11	250	256116	1467	99.43	1.392	1.362	1.281	1.345	5.648	5.537	5.638	5.607666667
12	300	307316	1670	99.46	1.442	1.472	1.402	1.438666667	6.339	5.518	5.678	5.845
13	350	358516	1862	99.48	1.452	1.462	1.412	1.442	5.889	5.859	5.688	5.812
14	400	409716	2062	99.50	1.502	1.492	1.492	1.495333333	6.119	5.738	5.748	5.868333333
15	450	460916	2266	99.51	1.532	1.593	1.522	1.549	5.808	5.668	5.808	5.761333333
16	500	512116	2459	99.52	1.573	1.562	1.572	1.569	5.999	5.859	6.018	5.958666667
17	550	563316	2659	99.53	1.572	1.612	1.593	1.592333333	5.978	5.918	5.979	5.958333333
18	600	614516	2854	99.54	1.713	1.743	1.622	1.692666667	6.009	6.059	5.989	6.019
19	650	665716	3052	99.54	1.652	1.812	1.683	1.715666667	6.399	5.999	6.069	6.155666667
20	700	716916	3252	99.55	1.692	1.662	1.672	1.675333333	6.309	6.088	6.109	6.168666667
21	750	768116	3454	99.55	1.663	1.692	1.612	1.655666667	6.129	6.008	6.36	6.165666667
22	800	819316	3649	99.55	1.702	1.752	1.632	1.695333333	6.249	6.45	5.979	6.226
23	850	870516	3849	99.56	1.812	1.643	1.713	1.722666667	6.108	6.399	6.179	6.228666667
24	900	921716	4051	99.56	1.723	1.732	1.742	1.732333333	6.059	6.189	6.028	6.092
25	950	972917	4245	99.56	1.753	1.732	1.702	1.729	6.71	6.049	6.089	6.282666667
26	1000	1024116	4446	99.57	1.722	1.783	1.702	1.735666667	6.099	6.169	6.339	6.202333333
27	1250	1280116	5436	99.58	1.822	1.793	1.742	1.785666667	6.149	6.139	6.089	6.125666667
28	1500	1536116	6426	99.58	1.712	1.702	1.723	1.712333333	6.199	6.149	6.219	6.189
29	1750	1792116	7420	99.59	1.793	1.752	1.722	1.755666667	6.119	5.978	6.35	6.149
30	2000	2048116	8413	99.59	1.793	1.773	1.682	1.749333333	6.119	6.509	6.489	6.372333333

Measurements taken without the Mobile Proxies System at 8.00 PM (Saturday)

All times in mSecs unless otherwise mentioned

		The FileTransfer.sendData method				The DataTransfer.DataClient application			
File	Size (KB)	Tx1	Tx2	Tx3	Txav	T1	T2	T3	Tav
1	0	0.891	0.791	0.891	0.857667	4.637	5.057	5.187	4.960333333
2	1	0.801	1.022	0.891	0.904667	5.277	4.777	4.937	4.997
3	5	0.911	1.022	1.081	1.004667	4.827	4.957	4.596	4.793333333
4	10	1.282	0.921	0.932	1.045	4.927	4.586	4.857	4.79
5	25	0.941	0.951	0.962	0.951333	4.787	4.887	4.957	4.877
6	50	1.132	1.012	1.362	1.168667	4.847	4.777	5.007	4.877
7	75	1.062	1.062	1.132	1.085333	4.847	5.058	5.108	5.004333333
8	100	1.212	1.102	1.372	1.228667	4.897	5.117	5.067	5.027
9	150	1.192	1.281	1.202	1.225	5.177	5.317	5.498	5.330666667
10	200	1.332	1.361	1.342	1.345	5.508	5.372	5.168	5.349333333
11	250	1.462	1.382	1.402	1.415333	5.298	5.538	5.407	5.414333333
12	300	1.642	1.512	1.582	1.578667	5.719	5.368	5.658	5.581666667
13	350	1.693	1.592	1.673	1.652667	5.678	5.798	5.468	5.648
14	400	1.752	1.792	1.783	1.775667	5.678	5.628	5.498	5.601333333
15	450	1.893	1.793	1.863	1.849667	5.728	5.668	5.779	5.725
16	500	1.963	1.852	1.923	1.912667	6.089	6.139	5.6681	5.965366667
17	550	2.093	2.013	2.083	2.063	6.108	5.938	6.139	6.061666667
18	600	2.053	2.003	2.073	2.043	6.099	6.5	5.918	6.172333333
19	650	2.223	2.083	2.073	2.126333	6.269	6.068	6.289	6.208666667
20	700	2.123	2.293	2.043	2.153	6.088	6.579	5.949	6.205333333
21	750	2.213	2.273	2.224	2.236667	6.079	6.179	6.079	6.112333333
22	800	2.224	2.514	2.083	2.273667	6.55	6.429	6.219	6.399333333
23	850	2.553	2.263	2.363	2.393	6.499	6.459	6.559	6.505666667
24	900	2.354	2.404	2.503	2.420333	6.51	6.299	6.239	6.349333333
25	950	2.353	2.424	2.423	2.4	6.44	6.87	6.189	6.499666667
26	1000	2.924	2.464	2.724	2.704	6.729	6.139	6.62	6.496
27	1250	2.454	2.825	2.553	2.610667	6.52	6.6	6.689	6.603
28	1500	2.734	2.644	2.483	2.620333	6.91	7.19	6.499	6.866333333
29	1750	2.674	2.684	2.704	2.687333	6.84	6.79	7.37	7
30	2000	2.934	2.934	3.014	2.960667	6.98	7.03	7.08	7.03

Estimated Compression/Decompression time v File size

File	Size (KB)	Compressed Size (KBytes)	T_{esm1}	T_{xav}	$T_{comp1ABS}$
1	0	0.10	1.1889	1.2538	0.0649
2	1	0.38	1.1889	1.2542	0.0653
3	5	0.41	1.1889	1.2558	0.0669
4	10	0.46	1.1889	1.2578	0.0689
5	25	0.53	1.1889001	1.2638	0.0748999
6	50	0.64	1.1889001	1.2738	0.0848999
7	75	0.76	1.1889001	1.2838	0.0948999
8	100	0.85	1.1889001	1.2938	0.1048999
9	150	1.05	1.1889001	1.3138	0.1248999
10	200	1.24	1.1889001	1.3338	0.1448999
11	250	1.43	1.1889002	1.3538	0.1648998
12	300	1.63	1.1889002	1.3738	0.1848998
13	350	1.82	1.1889002	1.3938	0.2048998
14	400	2.01	1.1889002	1.4138	0.2248998
15	450	2.21	1.1889002	1.4338	0.2448998
16	500	2.40	1.1889003	1.4538	0.2648997
17	550	2.60	1.1889003	1.4738	0.2848997
18	600	2.79	1.1889003	1.4938	0.3048997
19	650	2.98	1.1889003	1.5138	0.3248997
20	700	3.18	1.1889003	1.5338	0.3448997
21	750	3.37	1.1889004	1.5538	0.3648996
22	800	3.56	1.1889004	1.5738	0.3848996
23	850	3.76	1.1889004	1.5938	0.4048996
24	900	3.96	1.1889004	1.6138	0.4248996
25	950	4.15	1.1889004	1.6338	0.4448996
26	1000	4.34	1.1889005	1.6538	0.4648995
27	1250	5.31	1.1889006	1.7538	0.5648994
28	1500	6.28	1.1889007	1.8538	0.6648993
29	1750	7.25	1.1889008	1.9538	0.7648992
30	2000	8.22	1.1889009	2.0538	0.8648991

T_{esm1} Estimated time for FileTransfer.sendData method without the MP System for the compressed file size.
T_{xav} Estimated time for FileTransfer.sendData method using the MP System and for the normal file size.
$T_{comp1ABS}$ Estimated compression/decompression time (T_{esm1} - T_{xav}).

A.2 Security Performance Results

The security performance results were taken for three of the security. These results are only meant to show how much security costs in terms of performance.

The Mobile Proxies System Measurements for the Digital Signatures Functionality (Using DSA algorithm)

		The sendData method time				The DataClient process time			
File (.pdf)	Size (KB)	Tx1	Tx2	Tx3	Txav	T1	T2	T3	Tav
1	0	5.257	6.249	4.346	5.284	12.417	13.229	11.357	12.33433
2	1	6.139	5.288	6.45	5.959	12.678	12.098	13.48	12.752
3	5	7.441	6.619	5.538	6.532667	14.361	13.269	12.198	13.276
4	10	5.536	6.499	6.299	6.111333	12.006	13.71	12.708	12.808
5	25	6.96	6.47	6.469	6.633	14.13	13.43	12.878	13.47933
6	50	6.85	7.842	4.606	6.432667	13.359	14.341	11.736	13.14533
7	75	6.189	8.002	5.267	6.486	12.578	14.551	12.618	13.249
8	100	8.552	8.562	8.652	8.588667	15.081	15.692	15.552	15.44167
9	150	7.832	8.191	8.873	8.298667	14.641	15.161	15.712	15.17133
10	200	9.624	9.153	10.375	9.717333	16.444	16.133	16.894	16.49033
11	250	8.923	9.263	10.956	9.714	15.963	15.713	17.285	16.32033
12	300	10.184	10.095	9.043	9.774	17.124	17.045	15.863	16.67733
13	350	10.495	9.644	9.624	9.921	17.085	16.073	16.314	16.49067
14	400	10.815	9.694	9.764	10.091	17.505	16.063	16.023	16.53033
15	450	11.276	10.315	10.305	10.632	17.645	17.565	17.185	17.465
16	500	10.745	10.215	10.926	10.62867	17.625	17.526	17.906	17.68567
17	550	10.765	11.046	10.465	10.75867	17.515	17.485	18.808	17.936
18	600	11.106	11.336	10.785	11.07567	17.144	17.465	16.774	17.12767
19	650	11.306	12.048	11.707	11.687	18.407	18.647	18.677	18.577
20	700	12.378	12.238	12.769	12.46167	19.388	19.258	19.188	19.278
21	750	13.309	12.377	13.379	13.02167	19.949	19.588	19.628	19.72167
22	800	12.719	13.73	12.628	13.02567	19.969	20.399	19.708	20.02533
23	850	13.409	13.569	13.119	13.36567	20.86	19.969	19.939	20.256
24	900	15.032	14.661	13.239	14.31067	21.711	21.881	19.589	21.06033
25	950	13.99	15.623	14.831	14.81467	20.71	21.962	23.043	21.905
26	1000	14	14.681	14.25	14.31033	20.569	21.961	20.85	21.12667
27	1250	16.103	17.685	16.093	16.627	23.373	24.555	23.073	23.667
28	1500	18.606	18.787	18.377	18.59	26.879	26.759	25.306	26.31467
29	1750	20.84	20.67	20.33	20.61333	27.509	27.549	27.49	27.516
30	2000	22.513	23.354	24.104	23.32367	30.694	30.584	29.323	30.20033

Notes:
1- All times in msec unless otherwise mentioned
2- Measurements taken at 12.15 AM, 26.8.1999

The Mobile Proxies System Measurements for the Asymmetric Encryption Functionality
(Using RSA algorithm with PKCS1 padding)

File (.pdf)	Size (KB)	The sendImage method time				The ImageClient process time			
		Tx1	Tx2	Tx3	Txav	T1	T2	T3	Tav
1	0	6.9	5.799	4.927	5.875333	14.271	12.108	12.718	13.03233
2	1	6.52	5.087	7.441	6.349333	12.959	12.107	14.27	13.112
3	5	7.231	6.94	7.571	7.247333	13.92	13.99	13.8	13.90333
4	10	7.661	6.449	6.129	6.746333	15.001	13.329	12.768	13.69933
5	25	8.863	8.602	9.223	8.896	15.743	15.612	15.793	15.716
6	50	10.866	11.486	10.956	11.10267	17.786	18.076	17.095	17.65233
7	75	14.521	15.463	15.162	15.04867	21.381	21.762	22.072	21.73833
8	100	17.305	17.956	18.447	17.90267	24.976	25.026	25.457	25.153
9	150	23.904	23.814	24.405	24.041	30.884	30.133	30.804	30.607
10	200	30.714	30.925	29.934	30.52433	37.303	37.504	37.244	37.35033
11	250	38.756	36.352	36.222	37.11	45.315	43.602	42.822	43.913
12	300	43.011	42.641	42.471	42.70767	50.632	49.501	48.941	49.69133
13	350	49.221	49.601	49.121	49.31433	56.421	55.66	56.201	56.094
14	400	55.42	55.029	54.554	55.001	63.121	61.639	62.998	62.586
15	450	At this file size, the Request buffer becomes							
16	500	too large that a java.lang.OutofMemoryError error is rased.							
17	550								
18	600								
19	650								
20	700								
21	750								
22	800								
23	850								
24	900								
25	950								
26	1000								
27	1250								
28	1500								
29	1750								
30	2000								

Notes:
1- All times in msec unless otherwise mentioned
2- Measurements taken at 6.10 AM, 26.8.1999

The Mobile Proxies System Measurements for the Symmetric Encryption Functionality
(Using DES algorithm with PKCS5 padding)

File (.pdf)	Size (KB)	The sendImage method time				The ImageClient process time			
		Tx1	Tx2	Tx3	Txav	T1	T2	T3	Tav
1	0	5.958	4.617	6.389	5.654667	12.447	11.176	12.879	12.16733
2	1	5.427	5.898	5.698	5.674333	11.727	12.187	12.589	12.16767
3	5	5.198	7.01	5.779	5.995667	11.537	13.349	12.799	12.56167
4	10	6.138	6.64	5.988	6.255333	12.548	13.369	12.618	12.845
5	25	6.95	5.267	7.631	6.616	14.05	12.227	14.19	13.489
6	50	6.95	6.71	6.56	6.74	13.93	12.949	13.71	13.52967
7	75	6.249	7.08	6.019	6.449333	14.441	14.23	12.308	13.65967
8	100	7.841	6.128	7.33	7.099667	14.08	13.039	13.669	13.596
9	150	6.509	8.813	7.591	7.637667	13.149	15.362	14.51	14.34033
10	200	9.273	9.814	9.193	9.426667	16.563	16.784	16.053	16.46667
11	250	8.843	10.352	8.743	9.312667	15.793	16.754	15.753	16.1
12	300	10.085	9.936	10.215	10.07867	18.416	17.836	17.365	17.87233
13	350	11.467	10.726	10.385	10.85933	18.407	17.195	17.335	17.64567
14	400	10.856	10.465	11.106	10.809	17.695	17.355	17.235	17.42833
15	450	11.997	11.306	11.667	11.65667	18.026	17.385	18.186	17.86567
16	500	12.208	13.639	13.209	13.01867	19.078	20.42	20.079	19.859
17	550	15.232	13.98	12.838	14.01667	21.731	20.78	19.478	20.663
18	600	14.811	14.39	14.861	14.68733	21.461	22.532	21.791	21.928
19	650	15.272	14.711	15.762	15.24833	22.141	21.871	22.492	22.168
20	700	15.823	15.873	15.913	15.86967	22.843	22.983	22.402	22.74267
21	750	15.162	15.782	16.314	15.75267	21.722	22.743	23.013	22.49267
22	800	17.625	17.025	16.623	17.091	24.825	24.185	22.933	23.981
23	850	16.845	18.196	18.176	17.739	23.254	25.036	24.645	24.31167
24	900	18.537	17.305	18.016	17.95267	25.096	24.606	24.845	24.849
25	950	19.007	18.63	18.607	18.748	25.486	25.78	25.177	25.481
26	1000	19.918	19.808	20.199	19.975	26.418	26.448	26.879	26.58167
27	1250	23.313	22.623	24.315	23.417	29.583	29.803	31.525	30.30367
28	1500	27.63	27.29	27.089	27.33633	33.728	34.3	33.117	33.715
29	1750	30.574	32.426	31.485	31.495	37.284	39.146	38.385	38.27167
30	2000	33.968	33.779	36.142	34.62967	41.26	40.058	43.432	41.58333

Notes:
1- All times in msec unless otherwise mentioned
2- Measurements taken at 4.00 AM, 26.8.1999

References

[Bershad et al 1995] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. *Extensibility, Safety and Performance in the SPIN Operating System*. ACM, 1995.

[Byous 1999] J. Byous. *Jini™ Technology Grants the Ultimate Wish.* Jini™ Technology Homepage, 1999.

[Chawathe, Brewer 1998] Y. Chawathe, and E. Brewer. *System Support for Scalable and Fault Tolerant Internet Services.* Proceedings of Middleware, 1998.

[Chawathe et al 1998] Y. Chawathe, S. McCanne, S. Fink, and E. A. Brewer. *A Proxy Architecture for Reliable Multicast in Heterogeneous Environments.* ACM Multimedia, 1998.

[CORBA/IIOP v2.3 Spec. 1999] *CORBA/IIOP v2.3 Specification*. The OMG, 1999, Chapter 21.

[Jini™ Architecture Spec. 1999] *Jini™ Architecture Specification*. Sun, 1999.

[Katz 1994] R. H. Katz. *Implementing Communication through "Situation Awareness" Adaptation and Mobility in Wireless Information Systems.* IEEE Personal Communications Magazine, First Quarter 1994, Volume 1, Number 1.

[Knudsen 1998] J. Knudsen. *Java Cryptography*. O'Reilly & Associates, Inc., 1998.

[Leue, Holzmann 1999] S. Leue, and G. Holzmann. *v-Promela: A Visual, Object-Oriented Language for SPIN*. Proceedings of the Second IEEE International Symposium on Object-oriented Real-time Distributed Computing ISORC '99, Saint Malo, France, 1999.

[Mitchell et al 1993] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. *An Overview of the Spring System.* Sun Microsystems Inc., 1993.

[Nelson 1981] B. J. Nelson. *Remote Procedure Call*. Carnegie-Mellon University report CMV-CS-81-119, 1981.

[Nelson, Khalidi 1993] M. N. Nelson, and Y. A. Khalidi. *The Spring Virtual Memory System*. Sun Microsystems Laboratories Inc., 1993.

[Nelson, Khalidi 1993] M. N. Nelson, Y. A. Khalidi, and P. W. Madany. *The Spring File System*. Sun Microsystems Laboratories Inc., 1993.

[Oaks 1998] S. Oaks. *Java Security*. O'Reilly & Associates, Inc., 1998.

[OPG 1998] *OrbixWeb Programming Guide*. Iona Technologies PLC, 1998.

[ORG 1998] *OrbixWeb Reference Guide*. Iona Technologies PLC, 1998.

[Shapiro 1986] M. Shapiro. *Structure and Encapsulation in Distributed Systems: the Proxy Principle*. IEEE, 1986.

WWW Resources

1. http://www.act.navy.mil/Adaptability_Flexibility.htm
2. http://www.sun.com/jini/overview/
3. http://www.cs.washington.edu/research/projects/spin/www/
4. http://www.omg.org
5. http://www.aba.net.au/solutions/crypto/jce.html