A Proxy Server Infrastructure for Adaptive Mobile Applications

by

Jianwen Wang
B.Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements of the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

September 1999

© Copyright 1999, Jianwen Wang

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

A Proxy Server Infrastructure for Adaptive Mobile Applications

Submitted by **Jianwen Wang, B.Sc.**,
in partial fulfillment of the requirements for
the degree of Master of Computer Science

Dr. Thomas Kunz, Thesis Supervisor

Dr. Evangelos Kranakis, Director, School of Computer Science

Carleton University

September 1999

Abstract

This thesis derives on a proxy server infrastructure for adaptive mobile applications in wireless mobile environments. Instead of the traditional client-server model, the client-proxy-server model is used in our work. The heart of this design is to use a proxy component supporting application(s) handoff between proxy servers at runtime without users' knowledge. The purpose of the design is to either balance the load between proxy servers or avoid the server(s) with high latency. Environment information, such as latency, memory, and processor queue length, are collected and used to make the migration decision. Technologies, such as Java and Voyager platform, for mobile computing are employed to develop the system.

A mobile MPEG player application is developed for testing purpose. A wireless environment is emulated for experiments. Several experiments for different cases have been done.

Acknowledgments

I would like to take this opportunity to thank my supervisor, Dr. Thomas Kunz, for his continuous support and valuable advises in completing my thesis. I would like to thank Bell Mobility Cellular Inc. and NSERC for sponsoring this research work. I would like to thank my family for their encouragement, generous help, and love. Also, I would like to thank all members in Mobile Computing Group and fellow students in Broadband Network Lab. for their ideas and useful comments.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation and General Architecture

1.1.1 Motivation

With the modern communication technology and economic development, wireless communication becomes more important. Those mobile devices such as laptops, palmtops, or functionally enhanced cell phones are already widely accepted and used in people's life. Compare to the wired networks, wireless networks have some weaknesses. For example, in a wireless communication environment, spurious disconnections happen frequently due to handoff process or shadowed areas, wireless links have much lower bandwidth than in wired networks, and available bandwidth varies because of the characteristic of user's mobility between different networks. Due to these reasons, wireless mobile computing will always be characterized by a scarcity of resources [42]. Meanwhile, mobile devices such as laptops or palmtops also have limitations. Unlike powerful desktop machines, mobile devices have limited battery power supply, a small user interface, limited storage capacity, and less computational power. Under such a harsh computing environment, researchers believe that the traditional client-server model for distributed applications are not well-suited.

Mobile applications execute in an environment that is characterized by a high degree of variability [25]. Depending on a mobile user's current location, the network states such as bandwidth, latency, and error rates, and other QoS characteristics such as system load can change drastically.

In order to fully utilize the current available environment resources and offer mobile users the best services, the distributed applications should be designed in such a way that they have capabilities of moving its objects around and changing its behaviors according to the characteristics of the current execution environment. In other words, mobile applications should adapt to the current execution environment.

An extra network component, proxy, is added to the traditional client-server distributed computing model. It is named client-proxy-server model and will be introduced in Chapter 2. This new model is more suitable to the mobile computing environment. An application is separated into two parts. One is running on the client, and the other is running on the proxy. Along with the new model and mobile application concepts introduced, many related issues are raised:

- How to design and implement adaptive mobile applications?
- What will happen if the current proxy server is overloaded?
- Under what kind of situations will the migration process between proxy servers (HandOff) be triggered?
- What kind of infrastructure is used to support running adaptive mobile applications and object migration between machines?
- How to collect system information at runtime?
- Will the proposed system be suitable for large area deployment?

In the following chapters of this work, we will discuss most of above questions in detail. An infrastructure is proposed in order to run adaptive mobile applications and support object mobility between proxy servers.

1.1.2 General Architecture

The North American Analog Cellular Phone System (AMPS) can be characterized as consisting of base stations and terminals, combined with highly centralized and very intelligent processing in the switching system. Modern GSM, CDMA, and PCS networks apply for the same architecture. Base stations have little local processing capabilities, such as broadcasting, receiving, and forwarding data packets to or from mobile host. Each of these base stations covers a certain geographic area, which is typically named a cell [7]. While mobile user roams from one cell to another, the base station is also changed. The basic cellular system architecture is shown in Figure 1.1.

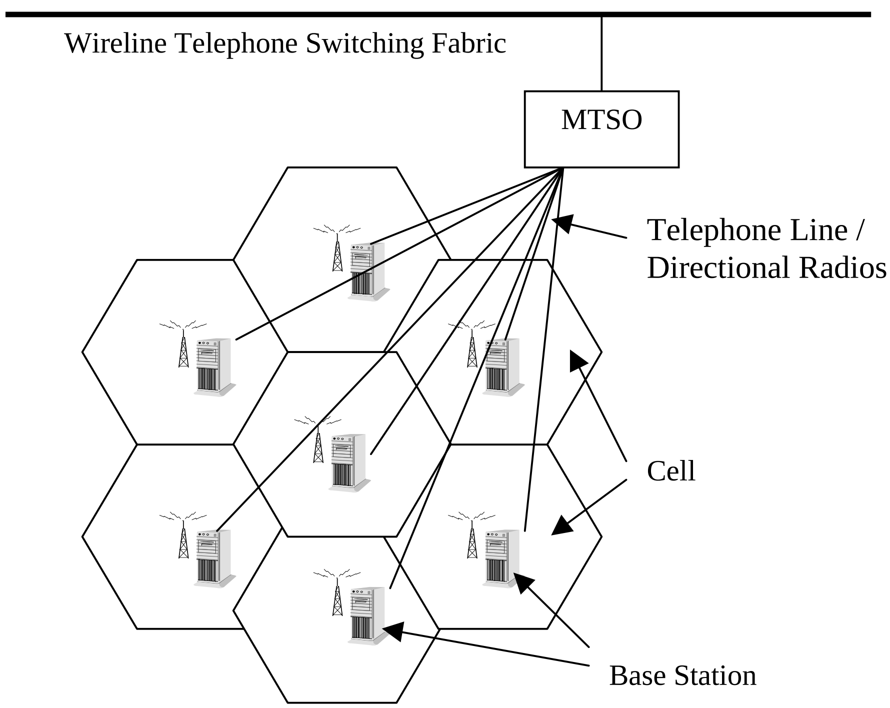

Figure 1.1 Basic Cellular System Architecture

The antennas represent the base stations within cells. The MTSO, or Mobile Telephone Switching Office, integrates the cellular environment into the wireline telephone switching fabric. Base stations and MTSO are typically connected by high bandwidth telephone lines or directional radios [7].

We are using a cellular system as our base environment. A view of the target environment is presented in Figure 1.2. In this figure, a pair of mobile devices (clients) are communicating with fixed hosts (servers). The mobile and fixed hosts are separated by an area of restricted network resource. Mobile devices are connected to the base stations via low bandwidth wireless links. However, on the other end of base stations, links to the servers have high bandwidth but very long distances (over the Internet). In between, base stations are gateways, receiving data from both sides and forwarding them to both sides as well.

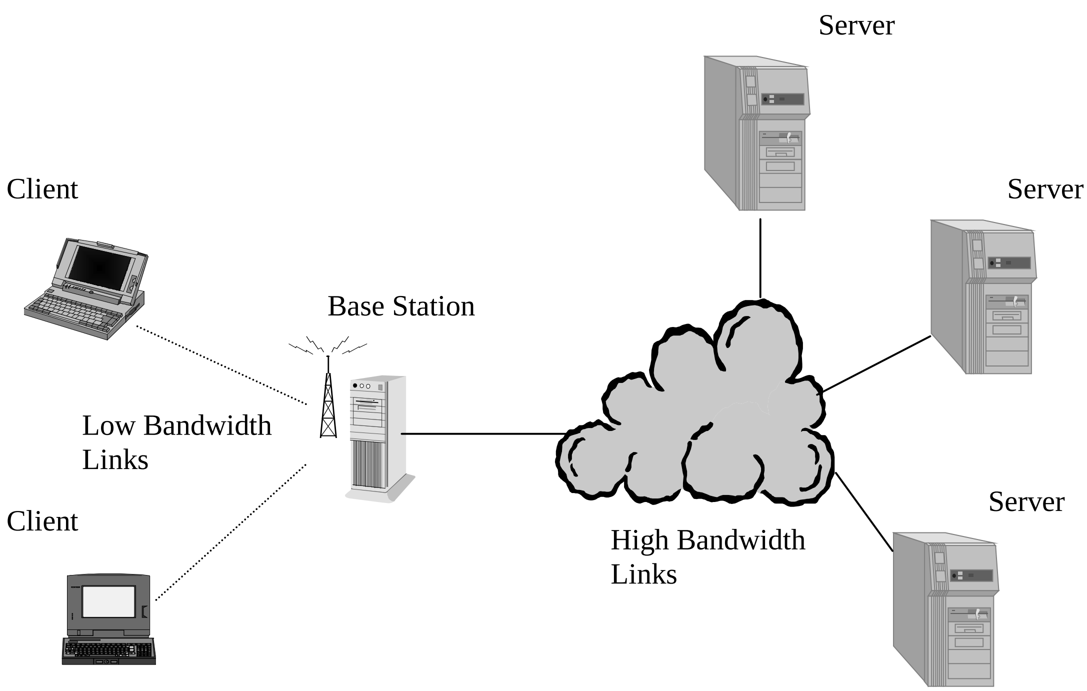

Figure 1.2 Wireless Mobile Computing Environment

Figure 1.3 provides a view of the same environment with the addition of an intermediary. The intermediary has been realized as the Proxy server, which is inserted between base stations and server hosts. We separate the proxy server from the base station because they provide different coverage and functionality. As a second execution environment for an application running on the Client, it may provide some extra services. One or more than one base stations are possible connected to the same proxy server. Also, base stations may connect to different Proxy servers. Therefore, it is possible for a client to choose a proxy server to get better services and/or connectivity without changing the location.

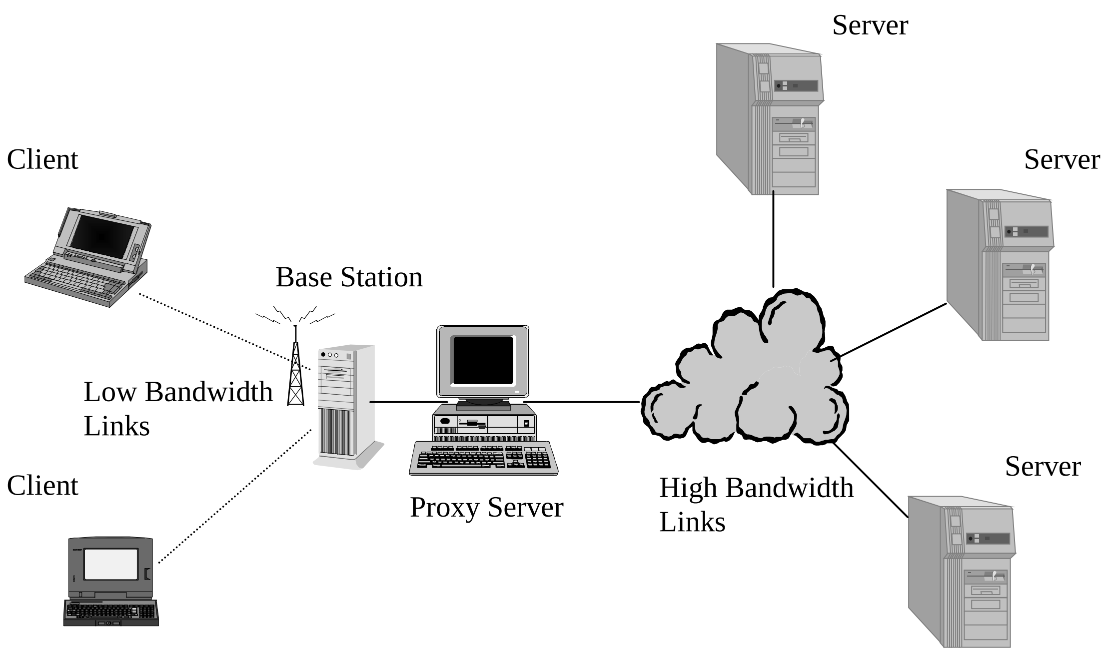

Figure 1.3 Wireless Mobile Computing Environment with Proxy

Proxy servers hide the fact of low bandwidth links between clients and base stations to the server hosts. Between neighboring proxy servers, a high bandwidth link exists. An infrastructure is designed and installed on the Proxy servers in order to support data mobility between servers.

1.2 Thesis Organization

The thesis focuses on the design of the architecture and introduces related knowledge and work, which has been broken down into eight chapters:

- **Chapter 1 Introduction**. In the first chapter, the motivation and the general architecture of this work are introduced.
- **Chapter 2 General Model, Mobile Applications, and Wireless Scenario**. In a wireless mobile computing environment, instead of the traditional Client-Server model, a Client-Proxy-Server model is used to support adaptive mobile applications. This concept is discussed in more depth.
- **Chapter 3 Related Work**. Some of those the related work pursued by other researchers are classified into four categories. They are briefly introduced and compared to our work.
- **Chapter 4 Development Platform and Sample Application**. ObjectSpace's Voyager technology has been chosen as our developing platform, which, together with some other mobile agent toolkits, is introduced and compared in Chapter 4. A sample application (MPEG Player – Decoder) is written in Java and deployed on the Voyager platform.
- **Chapter 5 High Level Infrastructure Design**. In order to support mobile applications and object mobility at run-time, an infrastructure is needed. In this chapter, a proxy server infrastructure is designed and implemented based on Voyager technology.
- **Chapter 6 HandOff Protocol Design**. Under certain conditions, objects running on one proxy server migrate to another server and continue the process, which is the key issue of this work. All these questions such as Why, When, and How does it work, will be addressed in detail.
- **Chapter 7 Experimental Setup and Evaluation**. This chapter concentrates on the experimental setup and an evaluation of the design.

- **Chapter 8 Summary and Future Work**. In Chapter 8, a summary of this work and some issues regarding possible future work are discussed.

Chapter 2 Challenges in Mobile Computing and Client-Proxy-Server Model

In three to five years, 20 to 50 percent of all Internet nodes may be wireless, mobile terminals [22]. Behind this development, advances in wireless networking technology and small data terminals have engendered a new paradigm of computing, called mobile computing, in which users carrying portable devices, such as laptop and palmtop machines, have access to a shared network infrastructure independent of their physical location.

In the following sections, we will briefly list some major challenges in mobile computing, and introduce the new client-proxy-server model, which serves a key role in our infrastructure design.

2.1 Challenges in Mobile Computing

In [48], [21], [23], and [64], the major challenges in mobile computing are described in detail. In this section, we will briefly discuss some of them, including: low bandwidth, high error rates, improper protocols, limited capabilities, and mobility.

Low Bandwidth

Wireless networks deliver lower bandwidth than wired networks. Different techniques support various bandwidth, ranging from 10 Kbps to 150 Mbps [23]. Some software techniques are used to improve effective bandwidth usage. [70] dynamically inserts functional filters at proxies to cut off less important contents of files or compress the files before sending them over the wireless link. [47], [18], and [42] propose schemes to dynamically change the division of labor between mobile devices and proxies at run-time.

High Error Rate

Unlike most stationary computers, mobile computers move across heterogeneous networks. Various networks have different characteristics and QoS (Quality of Service) support. The wireless environment introduces higher error rates than wired networks do. More aggressive error recovering schemes should be used to improve performance.

Improper Protocols

Mobile applications must deal with the legacy of wired network software. Transport level protocols, such as TCP, were designed specifically for the error and delay characteristics of a wired network [38]. Inefficiencies arise when these protocols misinterpret the causes of problems on the wireless link. For example, in the presence of the high error rates of wireless environments, TCP reacts to packet losses as it would in the wired environment: it drops its transmission window size before re-transmitting packets, to initiate congestion control mechanisms [67]. This reduces the throughput when the protocol should in fact retransmit the lost packet immediately. The IP protocol is also not suited for mobile computers. If the mobile computers move without changing their addresses, they will lose routing; if they reconfigure their addresses, they will drop existing connections. Instead of the regular IP protocol, a new scheme, Mobile IP, is employed. For a more detailed introduction to Mobile IP see [62] and [50].

Limited Capabilities

Compared to desktop workstations, mobile devices have limited capabilities, such as lower CPU power, less memory storage, smaller screen, and limited battery power [58]. These restrictions prevent mobile devices to be direct replacements for the more powerful workstations. Applications considered to run on mobile devices have to be carefully designed to take into account all of these limitations. A proxy-based design and object mobility can possibly be used to overcome these restrictions. For example, some heavy computational jobs can be executed on a proxy to reduce the CPU power usage and save the battery power on mobile devices, or proxies can be used to store big files to overcome the shortage of memory on mobile devices.

Mobility

In mobile computing, the ability of mobile hosts to change their locations while connected to the network increases the volatility. The network address of mobile hosts changes dynamically, so that they have to re-configure their network information appropriately. The basic mechanism for determining the current address of a mobile host is to use Mobile-IP schemes [62] and [50].

2.2 Client-Proxy-Server Model

"A client-server application on a mobile device or desktop workstation provides some functionality to the end-user in conjunction with server(s) in the Internet. Examples are the WWW browsers that retrieve documents from servers over the Internet, or clients that connect to FTP servers to upload or download files"[42]. The Client-Server model is widely used and it seems to fit many applications. However, in wireless mobile environments, the typical client-server model is ill-suited. For instance, a mobile device, with limited capabilities, is not a preferable place for a large client program to do some heavy computational jobs. Also, over a low bandwidth wireless link, it could be unacceptably slow for a client to download a big file from the server.

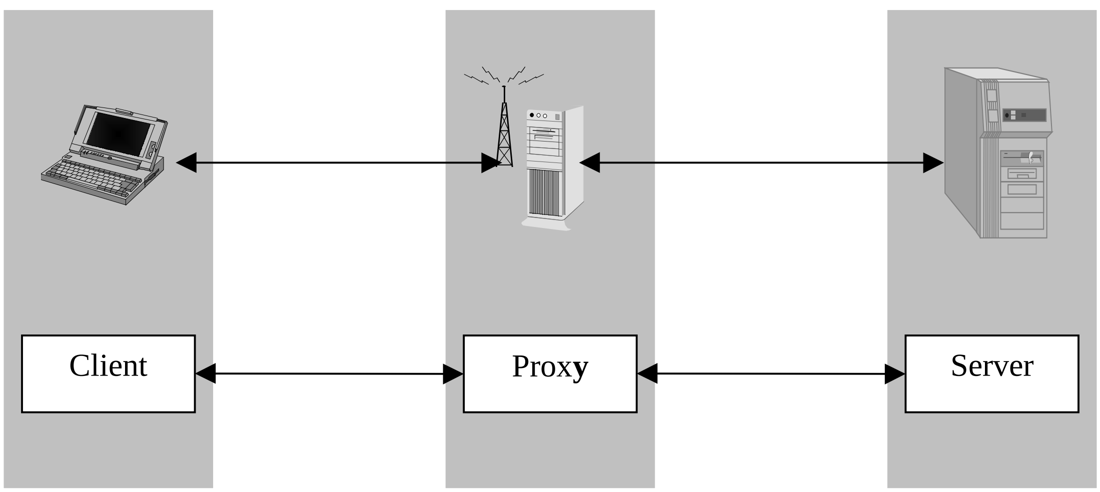

Figure 2.1 Client-Proxy-Server Model

Our design is based on an extension of this traditional client-server model to a client-proxy-server model. Figure 2.1 shows the model and relevant components. The typical client program is separated into two parts running on mobile devices and proxies, respectively. The client provides the user interface and some part of the application logic, which usually is relatively small. The proxy holds most part of the application logic, because it is, usually, a powerful machine. Proxies act like gateways for clients to communicate with servers, and hide the fact of the low bandwidth of link and limitations of mobile devices to the servers. For servers, proxies are clients, while, for clients, they are servers.

Existing proposals [70][12][29] typically install a proxy that filters and/or compresses data for a specific application in order to help eliminating limitations introduced by mobile devices and the wireless link. This filter is either enabled or disabled depending on changes in the current environment. For example, in order to reduce the bandwidth usage, an MPEG player can download the file from the web server, decode streams at the proxy, compress the pix-map, and send the compressed results to the client, which decompresses the result and displays the pictures.

In our approach, proxies are connected via a wired network. Workload can be shifted not only between clients and proxies, but also between different proxies, depending on current run-time conditions. A proxy infrastructure is created to support client/user mobility. In the following chapters, we will talk about this in detail.

Chapter 3 Related Work

The goal of our work is to provide a proxy server infrastructure for adaptive mobile applications in mobile computing environments that often change dramatically. A number of recent efforts have resulted in systems which are related to the one designed by us. We place related work into four categories: infrastructure support for object mobility, environment and network awareness, load-balancing strategies, and technologies for developing mobile applications.

3.1 Infrastructure Support for Object Mobility

In M-Mail [47], application objects are placed locally on the mobile client or remotely on the proxy server. For the application, the migration of the objects from one host to another is simulated at the user-level. The kernel or any low-level information is not transferred. However, M-Mail does not really support object and process migration between hosts. To “move” an object between hosts, a new copy of object is remotely created on the target host, and then the contents of the old object are transferred to the new object in sequence by using RMI (Remote Method Invocation) calls. Finally, the old object is destroyed.

Compared to pure object migration, without explicit object creation, destruction and separated state information transfer, supported in our design, M-Mail may take more time to do these “extra” jobs. In our design, objects and their runtime states are fully packed and transferred to the new location by using Java serialization technology. It is safe and done without user’s involvement. M-Mail’s design only describes the object mobility between client and server. It does not support the object mobility between proxy servers (handoff). Another drawback of this design is that it is not safe if many objects are closely related to each other (for example, two objects communicate with each other frequently). To kill an object, at runtime, may affect the others. It raises the probability of crashing the program and increases the complexity of the application logic.

The Rover [35] toolkit provides re-locatable dynamic objects (RDO) and queued remote procedure calls (QRPC) in order to better support application mobility. A re-locatable dynamic object is an object with a well-defined interface that can be moved between hosts. Queued remote procedure call is a system that allows RPCs to be queued for later transmission and execution, allowing the caller to continue processing. Applications are designed to consist of a set of RDOs that move between client and server depending on the state presented on the mobile host. The Rover system provides an execution environment for code associated with RDOs and handles the transport of RDOs between the client and server (via queued RPC).

The toolkit is designed for the case of object(s) mobility between client and server. Like M-Mail, it does not support proxy handoff between servers. Another disadvantage of such a system is that mobile applications must be designed from scratch, using a new programming model.

[70] describes a mechanism and how it has been applied to problems arising in the mobile environment. A service proxy is inserted into the middle of client-server connections. The paper presents a detailed design and implementation of the system that provides filter control and allows filter objects to adapt to their environment. In [68], however, the underlying operating system has to be modified to support the mechanism. His approach does not talk about how to support multiple applications. Also, the approach does ignore the problem of proxy handoff process, which actually is one of the important problems in mobile computing.

In [31], [32], and [52], a framework for constructing mobile applications has been designed. The key ideas in this framework are service proxies and object graphs. In the approach, an application is split into two segments, one segment runs on the mobile host, and another is on the service proxy, which is an extra component located between client and server. The segments themselves are composed of object graphs, which are modular code segments that can be reconfigured, depending on the current resources available to the mobile host.

This work is similar to ours in the way that the infrastructure also supports object mobility between service proxies (proxy handoff). The focus of this effort is an infrastructure to allow these object graphs to be constructed and reconfigured in a new service proxy, depending on state information obtained from the mobile host when a handoff happens. The mobile host is involved to trigger the handoff process in their design. This is different from what we are doing. In contrast, we design an infrastructure on the proxy server side triggering handoff process and migrating objects without the mobile host's involvement. The advantage in our approach is that proxies can initiate handoffs to balance the load among them, rather than simply trading mobile users.

Another disadvantage of [31] is that the running application suffers longer periods of suspension than in ours approach. All data packets from the old service proxy are suspended from the time the mobile host triggers the handoff to the end of the migration process. In our design, an application keeps running and connected till the infrastructure starts to migrate objects. In addition, like the other approaches described above, this approach does not have any further discussion about how their system supports the case of multi-user access.

3.2 Environment and Network Awareness

[15] discusses how to develop applications that can deal with changes in the network environment. Their model emphasizes the crucial role of the network connection: in many cases, the network is on the critical path, and performance problems in the network are the cause of the degradation of application performance.

In [15], they focus on the concept of network-awareness implying that an application's behavior is primarily controlled by the availability of network resources (bandwidth and latency). Although they believe that the system resources (CPU, memory, and processor queue length) are important, they do not focus on the issue of system-awareness. We not only take into account network-awareness in our design, but also system-awareness. Network bandwidth, latency, and the proxy server's current workload are primary environment parameters used in our design to optimize the services.

[66] builds an architecture to support user-level environment monitoring for environment awareness. Their architecture is built around a flexible mechanism for event detection and delivery, where a change in the state of the environment is modeled as an event. Several environment parameters are considered in their system such as power state (battery), network state (latency and bandwidth), and system loading state (CPU, memory, and disk). In our design, the power state information is not considered because we have more focused on the state information related to the proxy server. Therefore, we primarily take into account network state and system loading state.

In [29], an Image Transcoding proxy is used as an intermediary between generic WWW servers and a variety of client devices. Varying bandwidths of links between client and server and the limitation of the portable devices are two major concerns in their approach. The system designed in [29] is separated into two primary components: the policy module and the transformation modules. The former makes the decision while the latter modify the downstream data. In our design, we have a similar component, **DMaker**, making migration decisions and supporting object migration. However, the system loading states on proxies are not considered in their design. Decisions are based on current link conditions and client device capabilities. In addition, object mobility and proxy migration are issues not addressed in their design. Similar to the approaches in [44], [19], and [37], this work is base on a static client-proxy-server architecture.

3.3 Load-Balancing Strategy

How to develop an efficient load-balancing algorithm is not the major objective of this work. However, during our design of the proxy handoff process, we need to use a load-balancing strategy to locate the target proxy server. Many load-balancing schemes exist but different strategies may be best in differing circumstances.

[30] introduces a dynamic load-balancing algorithm. Most dynamic load-balancing algorithms can be placed in one of two general classes: geometric and topological. Geometric methods divide the computational domains by exploiting the location of the objects in the simulations. Topological methods work with the connectivity of interactions instead of with geometric coordinates. Topological methods usually can lead to better partitions than geometric algorithms. The GON strategy used in our design belongs to this class, and is a so called local load-balancing method. Local load-balancing methods use measures of work within neighborhoods, or closely-connected processors, to improve load-balance within each neighborhood. In our design, neighborhoods are directly connected proxy servers. Unlike global load-balancing methods, each iteration of a local method is fast and inexpensive, because all of the information and communication is performed within small sets of proxy servers.

Several load-balancing schemes are described and compared in [69]. Many of these schemes are not suited in our case. For example, in the global load-balancing strategy, the decision is made using global knowledge, i.e., all the processors in the network take part in the synchronization, and send their performance profiles to the load balancer. This is an expensive process and will not scale for a large network. [69] also gives a detailed description of a centralized local load-balancing strategy which is similar to our GON strategy. In this scheme, the processors are partitioned into different groups of size K. The group’s members can be selected randomly and changed dynamically. There is one centralized load balancer, which asynchronously handles all the different groups. Once it receives the profile information from one group, it sends instructions for redistribution for that group before proceeding to the other groups. This is different from our approach. We group only the proxy server serving a mobile user and its neighboring proxies into a group. Therefore, in our approach, the size of a group is fixed and membership can not be changed. The centralized load balancer executes on the current proxy server and it only is responsible for the local group’s load-balancing.

[58] introduces a load sharing strategy for mobile computers. Some literature distinguishes between load-balancing and load sharing. Load-balancing is often defined as a strategy which attempts to assure that each processor in a system has equal load. Load sharing, on the other hand, is usually referred to as a strategy, which attempts to share loads in a distributed system without attempting to equalize its load. Both strategies have the same goal, which is to make better use of the system resources. The algorithm consists of 2 policies. A transfer policy decides when a job should be transferred. And the location policy decides to which host a job should be transferred. The number of jobs in the queue waiting to be serviced is used to determine the transfer policy. Similarly, we also use the queue length as well as some other network states to determine the migration target.

3.4 Technologies for Developing Mobile Applications

Aglets is a Java library developed by IBM Tokyo Research Laboratories [63]. The Aglets library is simple and has a clean design. Aglets are executed in a similar way to runnable applets. Each aglet must implement the **run()** method. The **run()** method is invoked when an aglet starts execution. Like Voyager, Aglets supports rich message features, and Aglet messages are encapsulated in a Message object.

Aglets allows an agent to move itself to a well-known location (URL). Other aglets or threads may request an aglet to migrate. However, Aglets does not allow an agent to move to an object, even if the object is stationary. In addition, the platform does not support mobility of simple Java objects. Voyager has more general migratory support as it allows any object to migrate, not just agents. This is an important feature and is extremely powerful for developing mobile applications.

Odyssey is the Java Agent toolkit written by Gereral Magic [28], the company that wrote the first commercial mobile agent language, Telescript. The Odyssey API is small and simple. It offers the essential features needed to write mobile agent applications.

In Odyssey, an agent may migrate to another place at the location given explicitly. Some migration status methods are supported to tell the status just before and after the agent leaves and arrives, respectively. The same feature is supported in Voyager.

Communication between agents, in Odyssey, is done through meetings. This is also a drawback of this system because agents can only meet when they are at the same location. If an agent wants to meet with another, it has to request the place to arrange a meeting. Again, like Aglets and Concordia, Odyssey only supports an agent to move between URLs.

Concordia is developed by Mitsubishi Electric ITA [49] and is a good and well designed system. It is easy to use, provides helpful tools and is well documented. Mobility is implemented in Concordia using itineraries. Each agent has an itinerary associated with it. However, Concordia does not support regular object mobility.

Compared to Voyager and Aglets, Concordia's weakness is its lack of support for different modes of delivering messages. It uses events as the method of communication between processes. That means, in Concordia, each agent has to implement a special method to handle different events (message types).

Chapter 4 Development Platform and Sample Application

4.1 OO Programming Language

Object-Oriented Programming is different from other styles of programming in that the design of a program breaks the problem into objects and their interactions. Each of these objects is supported by code that allows it to function as required by the problem. Each of these objects will also probably interact with each other, via “messages”. These messages are simply the member functions or methods associated with the particular object. By contrast, those procedural-style programming languages such as BASIC, C, Pascal and FORTRAN all use functions and data to do their work.

Objects are the key issues in an OO program. There are usually many objects in such a program. Each of them represents different aspects or features of the problem. In the MPEG [45] video player we developed, for example, there are as many as nine major objects. The main object, called **MV_Displaying**, has methods such as **initialize()**, **resizeWin()**, **repeatMovie()**, **stopPlay()**, etc. It acts as a driver and an interface, creating other objects, displaying frames passed from **MV_Decoder** object, and controlling the user interface.

4.1.1 Why Use OO?

There are many reasons why people use OO. Some of them are critical and will be discussed here.

Maintainability: Code is easier to maintain if it is divided into relatively independent components such as classes. If a class is faulty, the only code that is at fault is in the class members. No other class is interfering, assuming those classes all work properly. Classes can be tested individually. In other programming paradigms, problems might arise because of global data conflicts, but encapsulation prevent that sort of nightmare.

Modularity & **Code Reuse**: Classes are standalone, or might rely on some limited number of other classes. This makes it easy to take code you have already written and debugged and reuse it in some other application. The best classes are simple and general, and become that way only through reuse and redesign. Once a class is simple enough to use and general enough to work in many domains, it can greatly simplify the creation of new applications. A new application has only to reuse the well-tested classes and create any new ones it needs.

Encapsulation: Classes group functions together under a common header. But they also group the data with which those functions work under a common header. The class then ensures that no other code outside of the class's domain can affect that data. This creates harmony between different classes when they are placed in the same application since there can be no name conflicts. Additionally, classes will provide a concise description of some component of hopefully many programs. This component is now a building block for new programs, which allows you to program at a much higher level of abstraction.

4.1.2 Java

There are three major widely used OO programming languages: Smalltalk, C++, and Java. There is no reason to review the detail for all three programming languages in this thesis. However, I still want, and I believe it is necessary, to give a little introduction on Java which was chosen as our coding language.

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991 [59]. Java derives much of its character from successful languages such as C and C++ by intent. But it is not simply an enhanced version of C++. It was designed to solve a certain set of problems just like C++ does. As an interpreted language, Java programs are interpreted rather than compiled. It is much easier to run them in a wide variety of environments. The reason is straightforward: only the Java run-time system needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Java has the ability to load classes into a virtual machine at run time. Both of these features, which are not supported by C++, are extremely important for data and code mobility across platforms.

Java has also been widely adopted as a language for writing mobile agent systems. The reason for this is that the Java technology comes with built-in support for many of the features needed to build a mobile agent system [36]. The Java core API includes powerful networking libraries such as sockets. Java supports class serialization, which enables an object to be written to a serialized stream or be read from a serialized stream into an object. Class serialization is necessary for transporting objects.

Java's native thread support makes it possible to create efficient systems. In our experimental case, we need the capability to execute special C/C++ functions or processes at run-time in order to get system information. Java supports this invocation of native code.

As of version 1.1 or higher, Java supports remote method invocation (RMI). RMI allows a Java program to invoke methods of objects that exist on other Java Virtual Machines, possibly on another hosts. RMI can be thought of as object-oriented equivalent of remote procedure calls (RPC) [59].

4.2 Object Mobility and Mobile Agents

Mobility is the ability to move independently from one device to another in a network. Object Mobility and Mobile Agents are slightly different from each other [36]. An object we are talking about, is a regular object in the OO sense and could be anything. In general, only serializable objects can be moved by either passing objects as arguments in a remote method call or creating objects remotely. There are several reasons why object mobility is useful:

- If two objects need to have a high-bandwidth conversation, they can move closer to each other to speed communications and reduce network traffic. Local messaging is often much faster than remote messaging.
- If an object requires features such as persistence or a fast processor, the object can move to a machine that has these features.
- If a machine containing an object is about to be disconnected from the network, the object can move to another machine to continue its execution.
- In the process of handoff or load-balancing between multiple machines, object mobility could be the key role.

An agent is a special object type. Although there is no single definition of an agent, all definitions agree that an agent has autonomy and mobility. An agent can be programmed to move from one location to another and satisfy one or more goals, even if this special object moves and loses contact with its creator.

There are many Mobile Agent Development platforms in use today. Most of them are well designed but have different limitations. The full description and comparison of some platforms, such as IBM's Aglets [63], General Magic's Odyssey [28], and Mitsubishi's Concordia [49], are presented in Appendix A. In Section 4.3, we will give an introduction to Objectspace's Voyager [56] toolkit, which is chosen as our development tool.

4.3 A Toolkit for Developing Mobile Applications

ObjectSpace Voyager Core Technology [56] (Voyager) is a powerful object request broker (ORB) for creating distributed Java applications. Unlike the other mobile agent toolkits Voyager is not built on top of Java RMI. The version of Voyager we used is Voyager 2.0.0.

A server is an environment for objects to execute. Objects can move between servers, servers can run on same machine or on different machines. A voyager server can be created from either the command line or a program. An address, in Voyager, is represented in the URL format **(//IP:Port)**, where IP and Port denote IP address and Port number, respectively.

Voyager uses Java interfaces to cleanly de-couple a (virtual) reference to an object and the object's implementation. Each class that is to be used as a remote object has a Java interface associated with it. Each method that has the capability of being called remotely needs to be included in the interface. The interfaces can be either written manually or be automatically generated with the Voyager **igen** tool [56], with **igen** creating interfaces from .class and .java files.

Voyager supports remote object creation. An object can be created at a remote host by calling the **Factory.create()** method. A reference proxy of the newly created object at the remote place is returned. Other objects can access that remote object via its proxy from anywhere in the network.

Voyager agents are a special case of Voyager objects. Voyager has an abstract base class Agent for writing agents. The main difference between Voyager agents and other Voyager objects is that is they run on their own thread.

Voyager allows any Voyager-enabled object to move locations, not just agents. Voyager objects can be moved by invoking the **moveTo()** method. The **moveTo()** method has been overloaded so that an object can move to either an explicit address, or if another Voyager object is given as the move-to location, to the location of that object. The **moveTo()** method of an Agent takes an optional method name argument. The method named is invoked upon arrival at the new location. If no method name is given, no method is called, so the agent arrives suspended. Since an agent's **moveTo()** method takes a pair of the location and the method to execute at the location, it is quite simple to create an itinerary for an agent. That is to create a list of paired locations and method names. The location specifies where the task is to be done. The method name specifies what method of the agent is to be invoked to do the task. The agent can iterate through the itinerary each time it completes a task at a location, and hence move onto the next task at the next location.

Voyager is the only platform to support messaging between agents located at different servers. Messages can be sent to remote objects (including agents) either by method call, (in a similar way to RMI), or by constructing a message object and calling the method **invoke**() to send it. A remote method call is synchronous: the calling thread blocks until the method returns. How a message sent by an **invoke**() is sent depends on the subclass of message. There are three message subclasses:

- **Sync**: for sending messages synchronously.
- **OneWay**: for sending a message asynchronously without expecting a reply.
- **Future**: for sending a message so that the calling thread continues executing and is returned an object that will contain the reply when receiver answers. The sender can poll this object to check for the reply arriving.

Voyager supports multicasting. A multicast message is sent to a space. When a message is multicast, the message is sent to all the objects within the same space as the sender object. Spaces are built up out of subspaces. A subspace is represented by the Voyager Subspace object. A subspace is simply a group of objects. A subspace can be connected together with other subspaces. The connected subspaces form a space. Multicast messages can be sent to either the entire space, or a filter can be used. Multicast messages are constructed and sent in a similar way to other messages. Multicast messages are sent asynchronously and without reply.

Voyager supports message forwarding. When an object migrates from one server to another, a forwarder is left at the old location. If a message is sent to the old address, the forwarder is used to send the message to the object at the new location. When the reply is returned, it carries with it a tag specifying the new address, so that the reference can update its information about the location of the object. All of this is transparent to the programmer.

Voyager supports object persistence transparently on an object by object basis. If an object is told to persist, then Voyager will ensure that a copy of the object is stored to disk. If the object is moved to a new location, then its stored representation is moved from the first location and stored again at the destination location. With the support for persistence, a node can be shut down and later restarted and all objects will be activated and in-memory.

Voyager has more features than we can describe. It provides diagnostics and error messages, security, and integrates with RMI, CORBA, and DCOM transactional systems which makes server-side development practical for mission-critical systems [35]. We built our mobile application and infrastructure based on the Voyager technology because of its rich features and better performance comparing to other mobile application development platforms. See comparisons in Appendix A.

4.4 MPEG Background and A Sample Application

In order to test the infrastructure, we need to implement a mobile application in client-server style. This application could be a MPEG player, Web browser, or something else. The application should consist of some objects and be written in Java. Some Voyager features, such as mobility and messages, will be added to the application. We decided to implement a MPEG player as our testing tool. To understand the MPEG standard and how it works is very important and necessary. We will give an overview of the MPEG standards in Subsection 4.4.1, and, in Subsection 4.4.2, we will describe the sample application in detail.

4.4.1 MPEG Video Overview

MPEG stands for the "Moving Pictures Experts Groups". To the real word, MPEG is a generic means of compactly representing digital video and audio signals for consumer distribution [45].

The basic idea behind MPEG video compression is to remove spatial redundancy within a video frame and temporal redundancy between video frames [51]. MPEG video syntax provides an efficient way to represent image sequences in the form of more compact coded data. Usually, MPEG contains both encoding and decoding processes. The MPEG encoding process is to transform a stream of discrete samples into a bitstream of tokens which takes less space. For example, a few tokens amounting to only 100 bits can represent an entire block of 64 samples which normally consume (64*8), or, 512 bits. However, the resolution of the image is kept almost the same. Then the essence of MPEG decoding process is to reverse the compact tokens back into something resembling the original stream of samples. In this reconstruction process, the coded bits are mapped from the compact representation into the original, "raw" format of the image sequence. For example, a flag in the coded bitstream signals whether the following bits are to be decoded with a DCT (Discrete Cosine Transform) algorithm or with a prediction algorithm.

By definition, MPEG samples have no more and no less than 8-bits uniform sample precision (256 quantization levels). For unsigned luminance data, black corresponds to level 0, white is level 255. With three color components (Red, Green, and Blue) per pixel, the total combination is roughly 256*256*256 or 16.8 million colors (i.e. 24-bits).

The compression ratio for MPEG, a lossy compression scheme, is always a trade-off issue. Higher compression ratio result in more lost resolution. If you want better quality of the image, lower the compression ratio. The MPEG compression scheme can easily reach a 100:1 compression ratio.

The video compression standards currently are MPEG-1, MPEG-2, MEPG-4, H.261, and H.263 [51]. MPEG-1 refers to the delivery of video for a CD-ROM quality presentation. MPEG-2 refers to broadcast quality compressed video. MPEG-4 allows for instance object manipulation for digital video even at low bitrates. H.261 and H.263 are video conferencing oriented. To explore and understand all MPEG standards is quite complex. More information about MPEG can be found at: http://www.mpeg.org/. We decided to use MPEG-1 standard for our implementation because we have found more MPEG-1 streams on the Web and it is also sufficient for our testing purpose.

4.4.2 Implementing a Distributed Mobile MPEG Video Player

In our experimental case, we only need a player which is able to read in a MPEG encoded bitstream from a Web/File server over the Internet and decode it back to the normal digital format (pixel-map) which is acceptable by computer display devices. A client-server MPEG video decoder (without encoder) is designed and implemented in Java.

In order to execute the decoder in a distributed environment, it was redesigned and implemented in Voyager. The player (decoder) consists of nine objects such as **MV_Displaying**, **MV_Scanner**, **MV_Decoder**, **MV_Backword**, **MV_Forward**, **MV_Idct**, **MV_Huffman**, **MV_Tools**, and **MV_FileIn** object. Among these objects, the **MV_Displaying** object is a user interface object running on the client host, while the **MV_FileIn** object is an object reading the MPEG bitstream from a file server (Web/File server), the rest are computationally-intensive objects running on the server side (Proxy server). Between objects **MV_Tools** and **MV_FileIn**, a socket connection is established to transmit data streams from a remote server to the local proxy server. This Java and Voyager based MPEG decoder works on all MPEG-1 files. Sound tracks are simply skipped while scanning the data stream. Figure 4.1 shows the program structure and the execution environment,

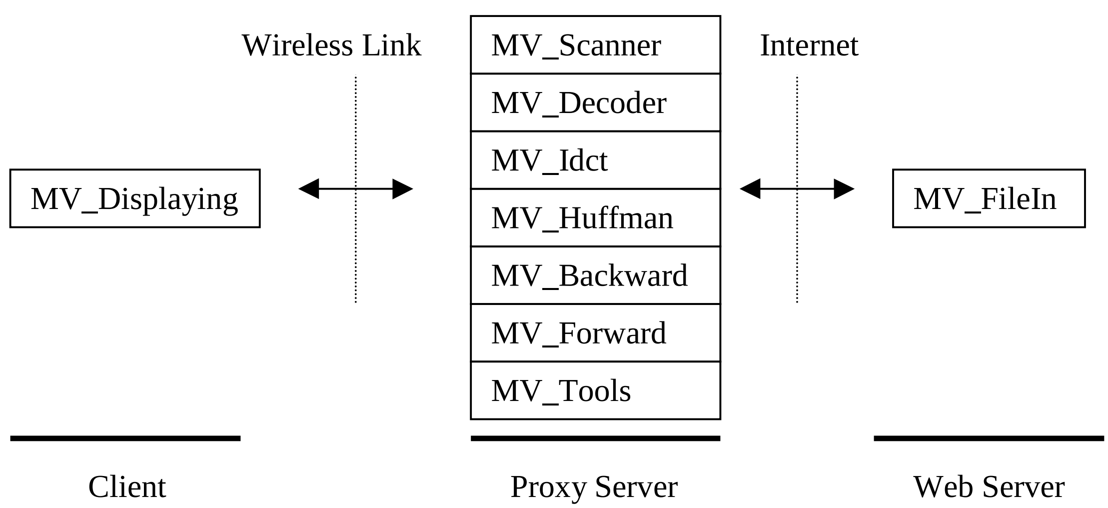

Figure 4.1: MPEG Player Structure

MV_Displaying: An interface object displays reconstructed frames (pixel maps). The object is not movable and runs on the mobile device (Client) as a thread

(DisplayingThread) communicating with heavy-weight objects on the proxy server via the wireless links. The interface window is depicted in Figure 4.2.

Figure 4.2: Displayer User Interface Window

The GUI consists of two components. The first window (1 in Figure 4.2) displays picture frames. The Interface window is resizable, depending on the size of the movie. The second component (2 in Figure 4.2) consists of a number of interactive buttons: Stop (temporary stopping), Cont. (continue playing), Replay (replay the movie one more time after it is finished), Repeat (continuously playing the movie while it is finished), and Quit (stopping the process and quit).

MV_Scanner: The Scanner scans the MPEG-packet-layer (layer I) and works as a thread (ScanningThread) concurrently to the DisplayingThread which displays frames. It determines the kind of information in a packet and eventually passes the information to the Decoder object. The object is movable and runs on the proxy server (Proxy).

MV_Decoder: The Decoder object scans the MPEG-video stream, extracts the information (the DCT values), activates the IDCT, applies the motion vectors and passes the pixel values (pixel maps) to the DisplayingThread, which is on the Client side. The object is movable and runs on the proxy server (Proxy).

MV_Idct: IDCT stands for Inverse Discrete Cosine Transform, which is the reverse process of the DCT algorithm used in the encoding process. The object is movable and runs on the proxy server (Proxy).

MV_Huffmann: The object implements the HVLC (Huffmann Variable Length Coding) algorithm, which is necessary for both encoding and decoding. The Huffmann object is movable and runs on the proxy server (Proxy).

MV_Forward/**MV_Backward**: Both objects are used to store and compute the motion information. The Forward object is for forward motion prediction while the Backward object is for backward motion prediction. Both objects are movable and run on the proxy server (Proxy).

MV_Tools: This object is an intermediary object sitting between objects on the proxy server and the **MV_FileIn** object on a remote Web/File server. It is running on the proxy server and contains all methods which are necessary for manipulating the data stream. A socket connection is established between **MV_Tools** and **MV_FileIn**. Via connected socket pipe, the data stream is sent from **MV_FileIn** to **MV_Tools** so that all objects on the proxy server can access the data stream locally.

MV_FileIn: FileIn object is the object that reads in the MPEG compressed bitstreams from a Web/File server and supplies some bit-oriented special I/O methods to the objects running on the proxy server. The object itself is not movable and runs on a remote Web/File server (Server).

The program is implemented in Voyager version 2.0.0. Except for the **MV_Displaying** running on the Client machine, all objects are remotely constructed at the proxy server or the file server when starting the program. This process is done on the Voyager platform that is introduced in previous sections. Voyager supports object mobility and remote messaging. By running this application, there are two possible moving directions for those objects running on the proxy server. One is to migrate between client machine and proxy machine, while the other is to move objects on one proxy server to another. The first possible migration, which is related to the load-balancing issues between client and server, is a part of the overall research project but will not be covered in this work. The second possibility is the problem of proxy handoff process, which will be discussed in detail in the next chapter.

4.5 GUI, Monitoring and Controlling Object Migrations

In today's Windows working environment, it is always nice and important to have a Graphic Interface so that a user can interact with the program graphically. In our case, a Graphic User Interface program is implemented in order to let the user easily monitor servers and objects involved in certain running projects. A migration process (handoff) of a group of objects or a single object movement between different servers can be simply done by a user's mouse drag-and-drop behavior. The interface program is platform independent and can run on any machines which have the JDK1.1 and Voyager packages installed.

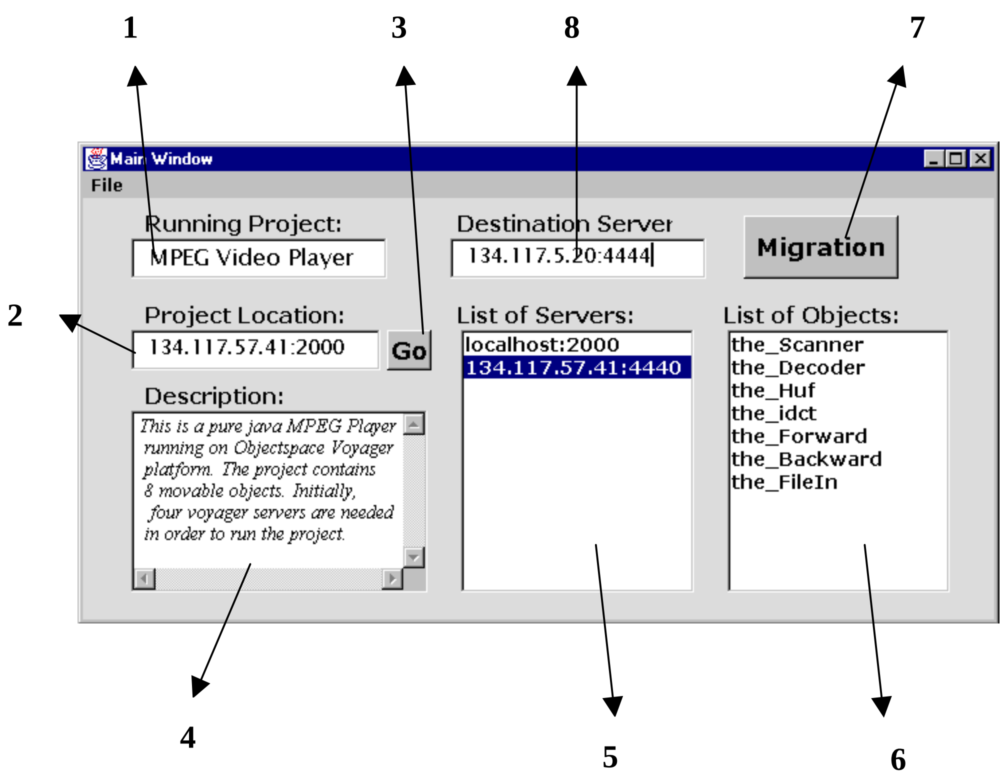

Figure 4.3: Main window

The program is designed to enable to monitor a running project from either a local or a remote machine. Therefore, the pre-condition for running the GUI is that the project that the program is going to monitor is already running. The user may start the program with or without an argument (location information of target running project). After starting the program with an argument, the main window appears on a screen as shown in Figure 4.3. In the absence of an argument, an empty window is shown. The user may enter a target project at anytime.

1. **The running project name field**: the name of the monitored project. The user can also enter a name into this field to start monitoring another running project.

2. **The running project location field**: shows the location information (IP:Port) of the running project. In this work, "location" means a voyager server that started application. The user may enter a target running project's location information into this field, then click the "Go" button on the right to monitor the target project.

3. **The Go button** is used to retrieve the information of a running project at the location shown in the project location field.

4. **The project description field**: a brief description of the project. The user can get basic information about the project such as what is the project, how many objects are involved, etc. from this field.

5. **Server Listing field**: lists all servers (Voyager servers) that are currently in use. The message consists of the server's IP address and the port number, separated by a colon. Sometime, a word "localhost" is used instead of the local machine's IP address. The user can select a server from the list by clicking on the address.

6. **Object Listing field**: contains a full list of objects running on a certain server. After a server is selected from the Server Listing field, all objects running on this server are displayed. The list is updated automatically while selecting another server.

7. **Destination server field**: gives the user a chance to enter a full address of a destination server, to where the user wants to move an object(s). The input address consists of IP address and port number, separated by colon. The word “localhost” is also good if the user only wants to move locally. After entering the correct address, the user can either hit the Return key to move groups of objects together or click the Migration button on the right to move an object(s) manually. The correctly entered address is added to the Server Listing if it is not there.

8. **Migration button** is used to open a migration controller window that will be introduced later. If the button is clicked while the Destination Server field is empty, a server window for each server in the Server Listing field is opened. This is also the case when the entered destination address is already shown in the Server Listing field. Otherwise, an additional server window (destination server) is opened as well. After the migration button is clicked, an Objects Migration Controller window (Figure 4.4) appears on the screen.

Following three items (9, 10, and 11) are defined and belong to an Objects Migration Controller window (Figure 4.4).

9. **Server Address**: contains server’s address information, consisting of full IP address and port number, separated by a colon.

10. **Server Window panels**: represent different servers (Voyager servers). Each of these server panels may be empty or contain one or more than one icons (objects). The user

can drag and drop icons between these server panels, which will cause the corresponding objects to move from source to destination screen.

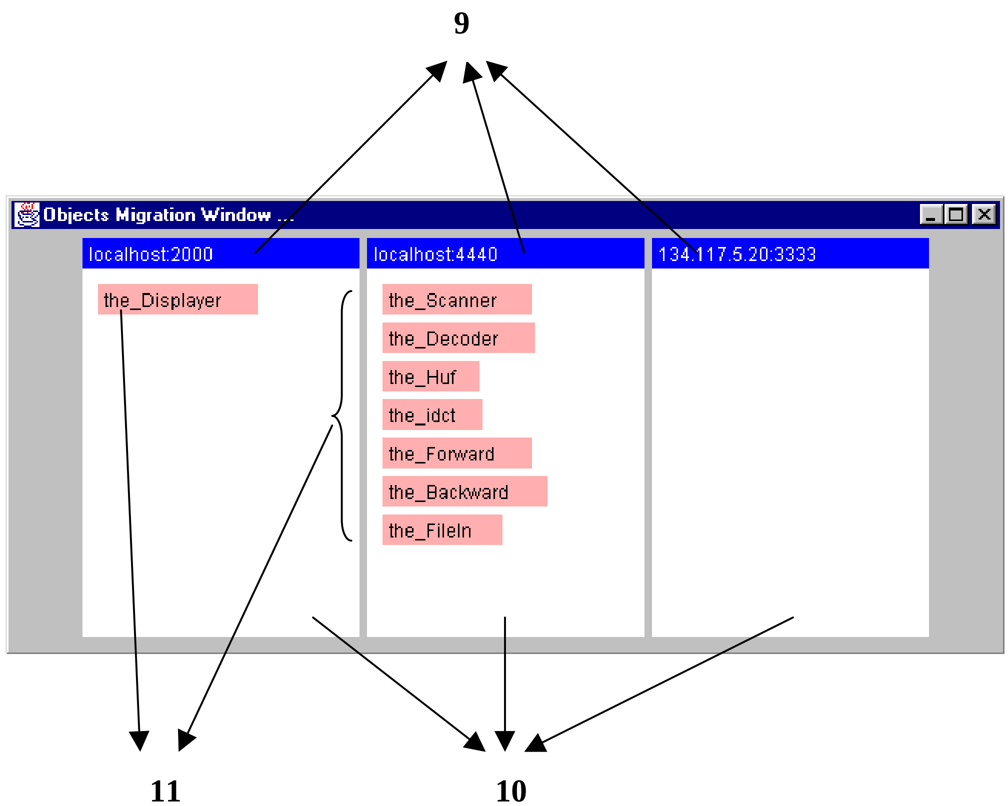

Figure 4.4 Objects Migration Controller window

11. **Object Icons**: symbols of objects running on different servers. They are colored rectangles, containing names of objects and can be dragged in one panel (server), and then dropped in another panel (server). The selected object is removed from the source panel and added to the destination panel. Whenever this happens, a moving message is sent to the running project, and the selected object will migrate to the destination server.

Chapter 5 Infrastructure Design

5.1 Introduction

In the previous chapter, we introduced mobile application platforms and described a sample mobile application – MPEG Player. To achieve truly adaptive application, we need to design and implement an infrastructure, which monitors the execution environment and supports object mobility. In Figure 5.1, an overview of the designed infrastructure and its components is given.

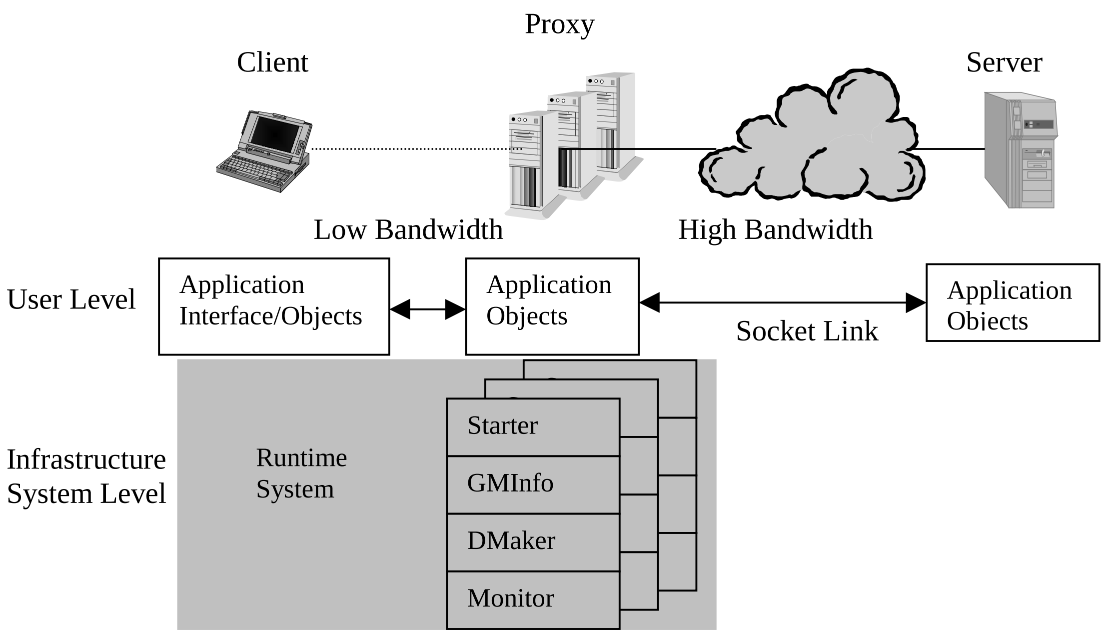

Figure 5.1 Infrastructure

According to Figure 5.1, the mobile device executes the user interface and (maybe) parts of the client application logic. Other parts of the client logic are executed on the proxy server, which is a powerful machine in the fixed network. The division of labor changes over time depending on the current environment, which relates to the load-balancing issues between client machine and proxy server, which is not part of this work.

In our design, the infrastructure is a part of the runtime system and supports mobile user applications running efficiently in a wireless mobile environment. The infrastructure contains four components, each of which is responsible for certain jobs. All components communicate with each other in order to coordinate their activities. Even additionally, some components on different proxy servers are in connection. The infrastructure is running in a default voyager server at each proxy server and waiting for any incoming user requests. Upon receiving a request from the mobile user, unique server space is generated at the proxy server for this application. The system monitor starts to collect local and remote (neighbors) runtime information. Based on the gathered information, a decision to either trigger the migration process or to continue execution locally is made. Objects at the current proxy server will be shifted to a new proxy server if a migration decision has been made.

The designed infrastructure supports multi-user and proxy migration (HandOff) process at runtime. Different applications share common resources, but run in separate logic spaces. After migration, a new logic space for a moved application is created at the new location. The infrastructure itself is implemented by using the Voyager 2.0.0 mobile computing development platform, which was introduced in the last chapter. In the following sections, we will discuss all infrastructure components and how they cooperate with each other to support multi-user access and proxy migration at runtime.

5.2 Infrastructure Components

5.2.1 Starter Component

The infrastructure includes four components: **Starter**, **GMInfo**, **DMaker**, and **Monitor**. In our design, the infrastructure is running on the proxy server and waits for user requests. The **Starter** component is like an access point to the infrastructure. It accepts request messages from the mobile users, and then tries to open a new voyager server on an unused port at the proxy server. If the randomly picked port number is found to be in use already, another port number is picked. This port selection process is continued until an unused port is found. In the return message, this new opened port number is carried back to the user application. After receiving the message from the **Starter** component, some objects of the application are shifted to the new port at the proxy server. The application is separated into two logic parts. One is on the mobile device, and the other is on the new port at the proxy server.

User: Request to Starter at Proxy: 1000

Starter: Generate an Unused Port: xxxx

Return Message to User

User: Connect Generated Server Space: xxxx on Proxy Server

Starter: Signaling Other Components in Local and even Remote

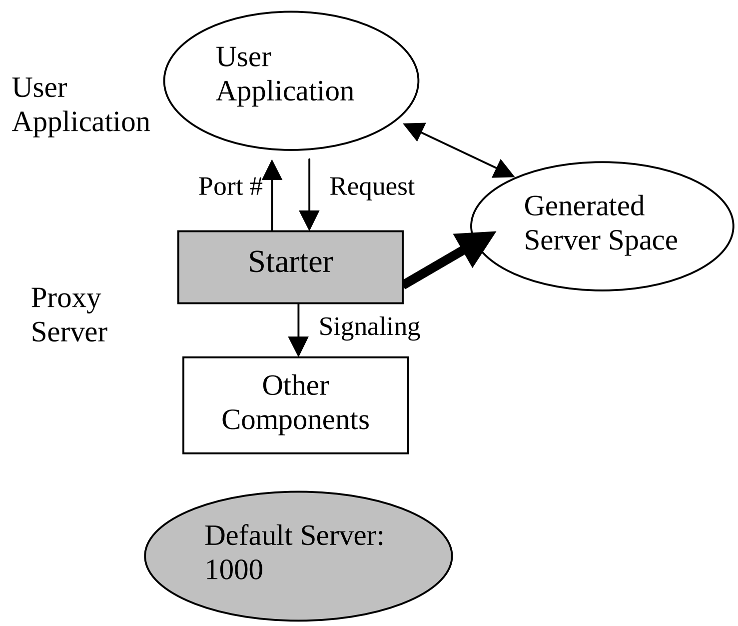

Figure 5.2 **Starter** Component and Message Passing

Figure 5.2 gives an overview of the **Starter** component and message passing between user application and infrastructure. The **Starter** component has its name bound to the default port (1000 in our case) at the proxy server, which has a fixed IP address. We assume that the IP addresses for the proxy servers and the default port number are well known to all mobile users. Mobile applications register themselves to the infrastructure by simply sending request messages to a proxy server at this fixed IP address and port. Voyager supports objects binding themselves to a certain address (IP:port), so that other objects can access them from anywhere. In addition, the **Starter** component is also responsible to signal the local **DMaker** component to start to work.

5.2.2 GMInfo (Group Member Information)

At runtime, the current proxy server (executing parts of client logic) and its neighboring proxy servers are gathered as a group. The current proxy server, executing the client logic, is the center host of the group. The group characteristics (center and size) are changed only at the time that the proxy migration process (HandOff) happens. During the HandOff process, we believe that the possible target proxy server must be a group member except the center. This strategy is called GON (Group of Neighbors), and it will be discussed more in Chapter 6. All members (proxy servers) in the group monitor the local system information and cooperate with other members in order to find the candidate with the best load conditions, i.e., it can supply best services to the user compared to the others.

Clearly, the communication between neighboring proxy servers is important. The **GMInfo** component at each proxy server holds all group members' IP addresses and potentially additional information such as a machine's alias name and type. Other components can obtain these neighbors' address information from this component anytime. Figure 5.3 shows the message passing to or from **GMInfo** and its lookup table.

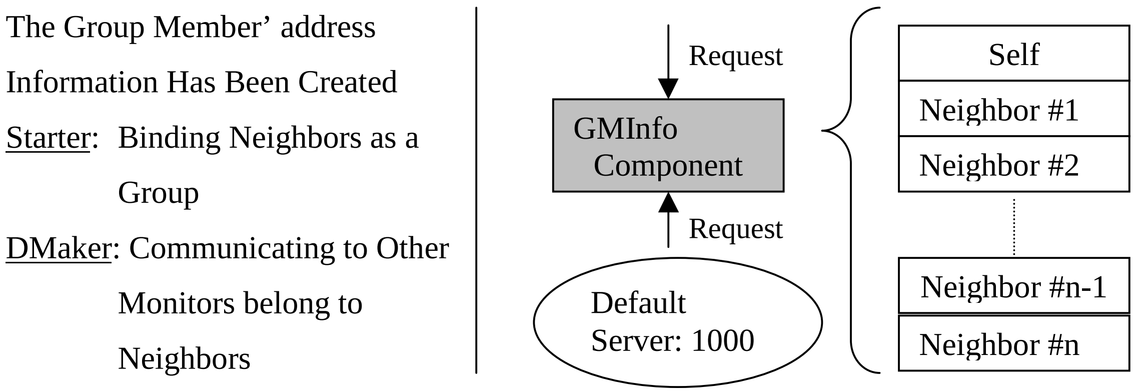

Figure 5.3 **GMInfo** Component and Messages Passing

The **GMInfo** component is designed to be independent from the others. It has a lookup table that contains name and IP address information of all neighbors. The **Starter** and **DMaker** components use the information to access remote proxy servers (neighbors). Since each proxy server has different neighbors, the lookup tables at different servers have different contents. Even the neighborhood information for a certain proxy server can be updated at runtime. For example, a machine entry can be removed manually from all lookup tables at neighboring proxy servers if it is down, or a machine entry can be added manually to all tables if it is newly connected to the network.

The **GMInfo** and **Starter** components are simple and only responsible for limited jobs. We will see that the other two components, **DMaker** and **Monitor**, are relatively heavy and supply more services to users. Of course, they contribute more overhead to the system. **DMaker** and **Monitor** components are critical for the infrastructure design and discussed in the following sections.

5.2.3 DMaker (Decision Maker)

For an application running both on a mobile device and a proxy server, only the **DMaker** component at this proxy server is active. All neighboring proxy servers' **DMaker** components are not in use because of the fact that only this center proxy server holds parts of the client logic and needs to make migration decisions. After the migration process has been triggered and successfully finished, the **DMaker** component at the target proxy server is activated and the one at the old proxy server is dismissed. This design can also preserve local machine resources.

Compared to the two components already introduced before, the **DMaker** component is responsible for several jobs in five steps executed repeatedly:

1. It gets system information from both local and remote (neighbors) machines. A new information table containing address information and each group member's runtime system information is created.
2. An algorithm is used to process these values in the table.
3. A migration decision is made based on step two.
4. If a migration decision has been made, **DMaker** informs the target proxy server's **Starter** component to open a server space.
5. After receiving a return message with a new opened server port number from the **Starter** component at the target proxy server, **DMaker** informs all objects at the current proxy server to migrate to the new place.

The **DMaker** component cooperates with other local and remote components. For example, the **Monitor** component keeps writing information into a local file periodically, and similarly, the **DMaker** component repeatedly reads those files locally and remotely. The **GMInfo** component supplies neighbor's address information including IP address and port number to **DMaker**.

DMaker: Read Address Information
from GMInfo Component
DMaker: Read Parameters Supplied
by Monitor Components
from Local and Remote
Files
DMaker: Process Values and Make
a Decision
DMaker: If Decide to Move {
Inform Starter Component
at Target Proxy
Call MoveTo()
} Else Repeat the Process

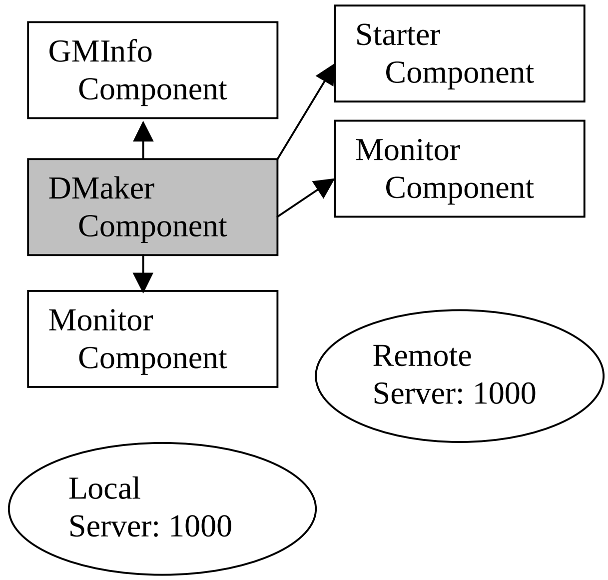

Figure 5.4 **DMaker** Component and Messages Passing

Figure 5.4 shows the message passing to or from **DMaker**. The **DMaker** component has more jobs to do and is dependent on other components. The final migration decision is made by this component.

5.2.4 Environment Monitor

A central component of the infrastructure is the environment monitor. Like the other components, a **Monitor** component can be found at every single proxy server of the network. A daemon process executes in a separate thread monitoring relevant environment parameters, and writing parameter values into a local file periodically. The **DMaker** component periodically reads in these values.

Considering our intended application, we decide to collect only few environment parameters such as memory usage, idle processor time, process queue length, and maybe the network latency and bandwidth. From the first three parameters, the current machine (proxy server) load can be estimated roughly. The latency and bandwidth parameters could tell us the current link situation (roughly) between client and each proxy server. Finally, based on the information, **DMaker** at the current proxy server could trigger the HandOff process. All these environment parameters and their relationship to our system will be introduced in Chapter 6. Figure 5.5 shows the message passing to or from **Monitor** component and its status at runtime.

Monitor: Execute Daemon Process Periodically

Monitor: Writing into a File Periodically

DMaker: Reading from a File Periodically

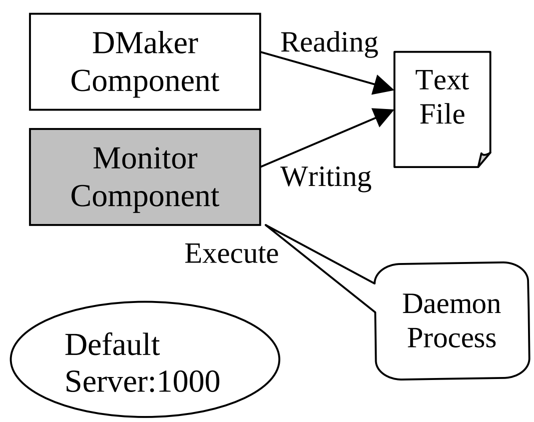

Figure 5.5 **Monitor** Component and Messages Passing

We are using both Windows NT and Linux (Unix) platforms as proxy servers. We also mentioned that the infrastructure is implemented by using Java and Voyager libraries. The Java language does not support some powerful system calls for language portability reasons. Therefore, in our infrastructure, the daemon process is written in C and uses the NT's system performance APIs to get required environment parameters at runtime. If the proxy server is a Linux/Unix machine, we do not need a separated daemon process at all. On Linux/Unix machines, those files containing environment parameters already exist and are updated periodically. Some of these files can be found under the /proc directory.

Clearly, running a daemon process (written in C/C++) from a Java program has overheads. Usually, there are two ways to execute a piece of C/C++ code from a Java program: either using JNI (Java Native Interface) technology or making C/C++ code as an executable file, then using Java External Process call to run the process (executable file) [59]. Both methods add overheads, however, the second method is quite straightforward. We decided to use the second method rather than the JNI approach. Of course, JNI has its benefit such that messages can be passed between Java and C/C++ programs at runtime. The comparison of the two methods can be an item of future work.

5.3 Multi-user and Proxy Migration Support

5.3.1 Multi-User Support

In reality, more than one mobile user may share the same proxy server at one time. For example, in an office building, many users may access the Web, checking news or stock information from their laptops at lunch time. These applications are possibly running on the same proxy server. Ideally, the number of mobile users connecting to a proxy server should not be limited. However, this is not true because of limited CPU power and memory on proxy server machines. In this section, we are discussing how the infrastructure supports multiple mobile users. Figure 5.6 shows the scenario of two mobile applications running on the same proxy server A at one time.

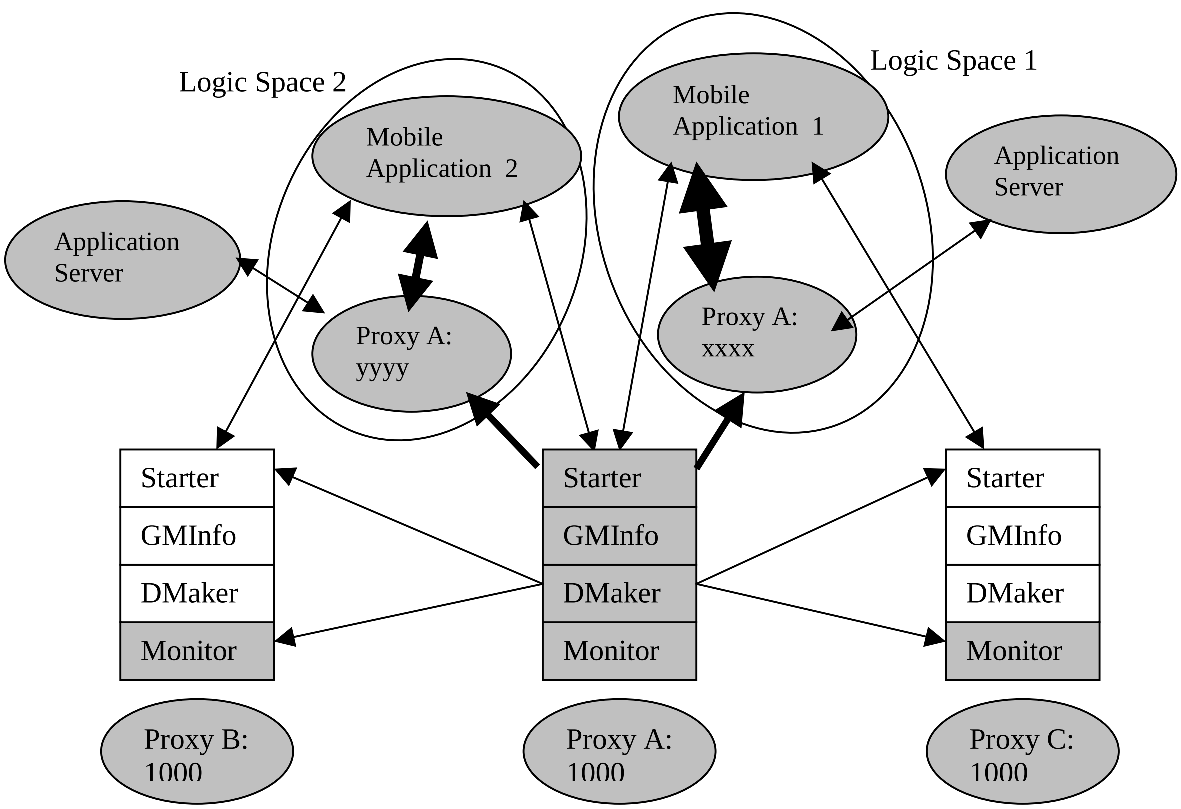

Figure 5.6 Multi-User Support

The designed infrastructure supports multi-user access. Several mobile applications can run on the same proxy server sharing resources. Each application has its own working space (port) on the proxy server. Before starting the program, each application registers itself with the infrastructure at the proxy server. In return, the infrastructure issues unique port space for each registered application. From the figure above, mobile application 1 has been allocated a working port xxxx, and mobile application 2 has been allocated a different working port yyyy. This work is done by message passing between the client applications and the **Starter** component at the proxy server (see Section 5.2.1 for the details).

After being informed that the unique port is ready at the proxy server, the mobile application will start to load its objects into the working port at the proxy server. A unique logic space is created for this mobile application. Therefore, each application has its own logic space across the mobile device and the proxy server, so that there are no messaging conflicts between different applications. Multiple applications are running on the same machine but in different logic spaces and remain there till the HandOff process happens. However, applications share the computing resources at the proxy server. In particular, the more applications run on the same machine, the slower the processing speed.

Proxy servers are generally machines with more CPU power and memory compared to the portable devices. However, they still could be congested when being accessed by multiple users. In this case, some workload on this proxy server should shift to a neighboring proxy server with less workload (short process queue list) or better links (high bandwidth).

5.3.2 Proxy Migration Support

Many reasons may trigger the proxy migration process. These reasons could be the current wireless link condition between mobile device and the proxy servers, and the current workload of proxy servers. In Chapter 6, we will discuss these triggers in detail.

The **Starter** component opens a Voyager server on port:XXXX (a unique port) at proxy server A for executing (parts of) the application after receiving a request from the client. The **Monitor** object wakes up and retrieves system information periodically. Meanwhile, **Monitor** objects on neighboring proxy servers are in charge to collect this local run-time information and send it to the proxy server. It is mentioned before that all proxy servers are linked together, but only the current running proxy server and its directly neighboring servers need to run the **Monitor** object.

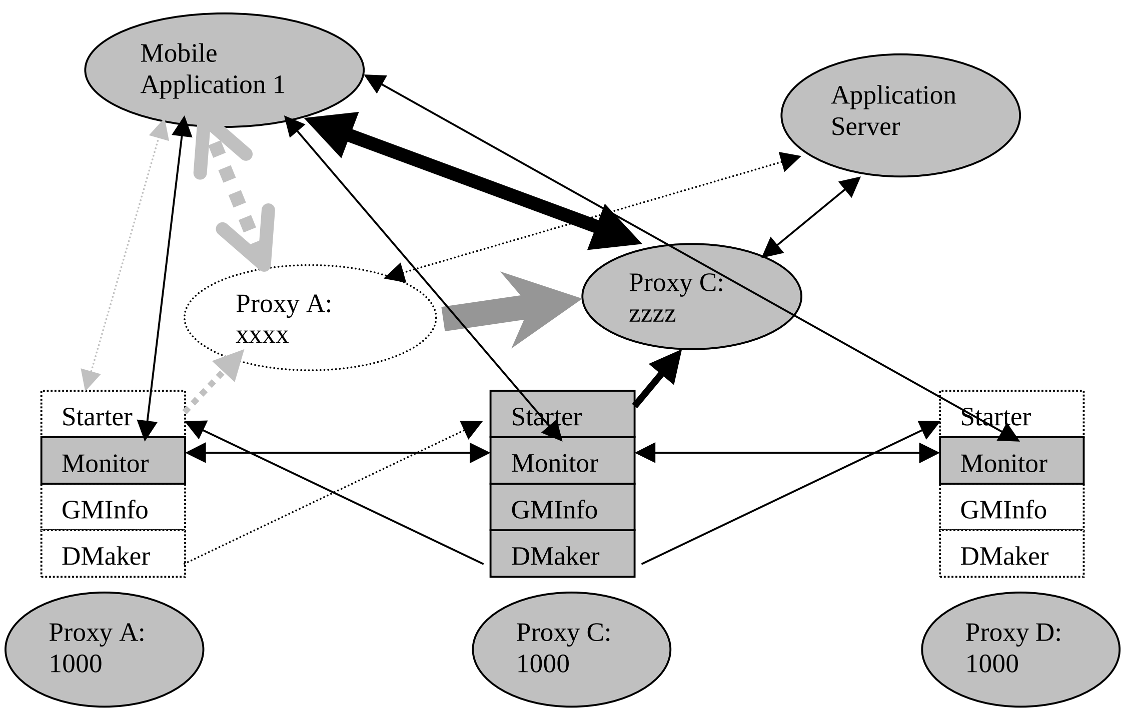

Figure 5.7 Proxy Migration Support

After analyzing the gathered information, the DMaker component makes the migration decision, if it is necessary, and informs object(s) at proxy server A to move to a new location at proxy server C. See Figure 5.7, which shows the situation after proxy migration. Notice, in the figure, that the center proxy server is Proxy C after migration.

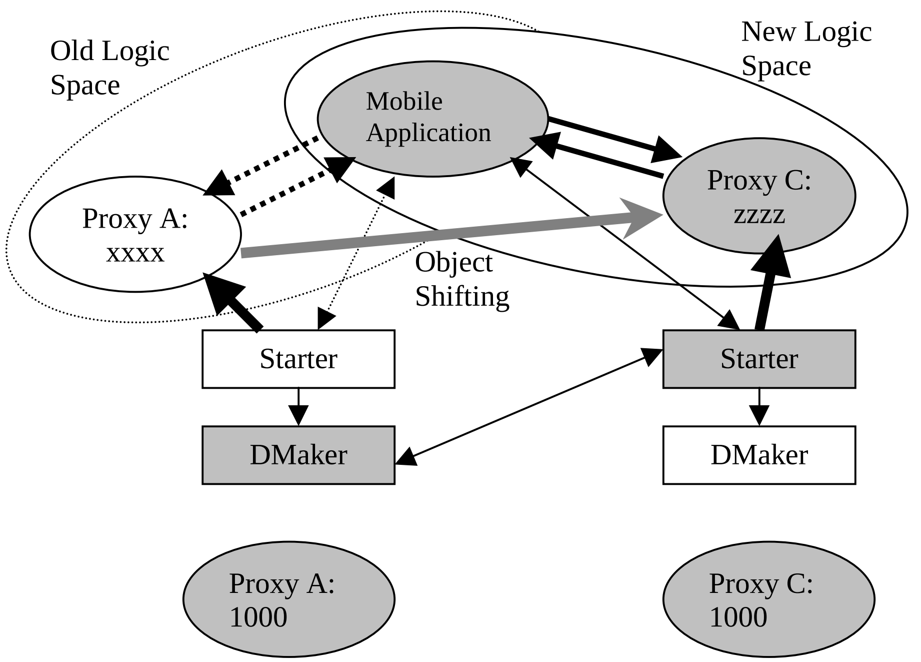

Figure 5.8 Logic Space Creation

After migration, all objects are in a new place. The new logic space for an application is created, and the old one is dismissed automatically (Figure 5.8). This migration process is transparent to the mobile user. And the application keeps running on the client after suffering a short handoff delay. The time to migrate objects depends on the decision made by **DMaker** component. It takes some time to move an object to another proxy server. Clearly, if the benefit from migration is small, compared to the costs of shifting objects, then not migrating would even be better. Since all proxy servers are connected in a wired network, we assume that the cost of shifting an object is small and stable. How to estimate and use the cost of the migration can be an item of future work.

5.4 Summary

The infrastructure is implemented on top of the Voyager 2.0.0 mobile computing platform. At each proxy server, a default server port (1000) is used and well known to all mobile users. There are four major components in our infrastructure design. The **Starter** component is the one to accept users' requests and generate unique port spaces for each application. The **GMInfo** (Group Member Information) component contains addressing information for all group members. Here the word "group" means that the current proxy server and its immediately connected proxy servers are grouped together in order to reduce unnecessary message passing in a large network (see next chapter for the detail). The **DMaker** (Decision Maker) component is responsible for many jobs such as reading values from a periodically updated local file, fetching information from the other group members remotely, processing gathered data, and making a migration decision. Finally, the **Monitor** component is running on each proxy server all the time. It keeps collecting environment parameters by running a daemon process and writing those gathered parameters into a local file periodically.

The infrastructure is designed to support multi-user access. From the mobile device, a mobile application registers itself with the infrastructure by sending a request message to the **Starter** component at proxy server. In return, a unique port is picked and sent back to the application at mobile device. Upon receiving the message, the application starts to load its objects to that unique port at the proxy server. A unique logic work space is created for each application. Different applications are running on different ports at the proxy server, but they do share local resources.

The infrastructure is also designed to support the proxy migration process at runtime. When the **DMaker** component decides to move, it informs the **Starter** component at the target proxy server to open a new port. It sends a migration command to all objects at the current port to move to a new working port at the target proxy server. A new logic space for the application is created and the old one is abandoned. Voyager supports object mobility so that this is done transparently.

About the proxy migration, one more thing is worth to be discussed here. Some people will argue that, before migration happens, the destination proxy server should be asked that if it is want accept the extra work lord. We agree that it could be an option because the destination proxy could be congested right after the decision has been made. However, we believe that it is just a rare case. The migration process happens quite fast. Also, the infrastructure makes the migration decision based on the run-time environment situation on both current and destination proxies over a period of time (30 seconds). We think that it is not really necessary to get the permission from the destination before moving. But, we may need future studies to see whether our reasons hold when deploying this architecture in a real system.

Chapter 6 HandOff Protocol Design

6.1 Introduction

We described the infrastructure design and implementation in last chapter. In this chapter, we will talk about how to achieve the HandOff process based on the infrastructure without terminating an executing application. A detail description of the HandOff protocol is presented in this chapter.

A mobile user's point of attachment (Proxy server) to the wired network changes over time (Figure 6.1) because of several reasons [64]. First, mobile users run a mobile application on the portable device and move from one place to another place. Their mobile device can no longer connect directly to the current proxy server, so that a new proxy server has to be used to replace the current proxy server. Second, the bandwidth of the wireless link between a mobile device and the current proxy server may be getting worse (low) due to some physical reasons such as weather and walls in the building. If there exists a better link (to a new proxy server, which has direct connection with the mobile device), the proxy server should be changed in order to have faster transmission over a link with higher bandwidth. And third, the current proxy server might be overloaded in the case of running heavy computation jobs or multi-user access at the same time. Some workload (objects) on this proxy, in this case, should move to a new proxy server with more free resources. In this work, we mostly concentrate on the first and third case.

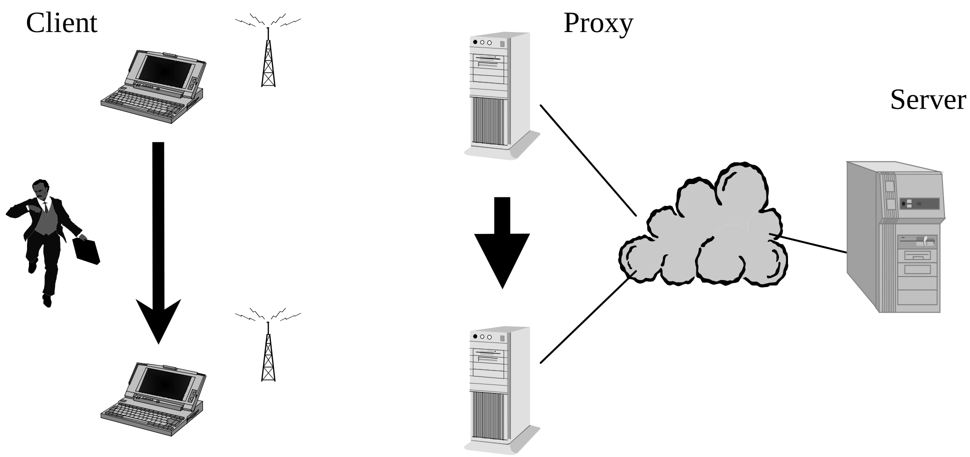

Figure 6.1 Proxy Migration

We already mentioned that all proxy servers are connected with high bandwidth links and there is a server infrastructure based on Voyager technology, which is described in the previous chapter, installed on these server machines. Upon this installed infrastructure, objects on one proxy server can seamlessly migrate to another proxy server at runtime without terminating the execution. This process is called the HandOff, which is transparent to mobile users.

In a mobile computing environment, the HandOff process may happen frequently. As indicated in [64], in order to successfully trigger and efficiently complete the HandOff process at run-time, there are three major issues of mobility that need to be addressed,

- When is the best time to migrate objects?

- How to locate the destination of the migration?
- How does the proposed solution scale to a larger network?

The purpose of this work is to answer first two questions. To determine when to trigger the migration process is important. The infrastructure described in the last chapter has the capability of monitoring the system environment at run-time. The collected information (environment parameters) will be analyzed, after which the migration decision will be made if it is necessary. Based on this run-time information, the infrastructure will also find out the best migration target. In our system, we introduce the GON (Group of Neighbors) strategy, which will help to reduce the network traffic caused by polling messages significantly.

In Section 6.2, the required environment parameters are introduced. In Section 6.3, a proposed HandOff protocol and an algorithm (pseudo code) are presented.

6.2 Critical Issues in Design

6.2.1 Useful Environment Parameters

Before making the decision whether it is necessary to migrate (parts of) an application, the infrastructure needs to collect some system information such as server machines' available CPU power and memory (load), the length of the ready queue, the bandwidth of the wireless link between portable devices and proxy servers, and the time needed for a round-trip message (see Figure 6.2). The infrastructure provides a **Monitor** component located on each server machine. The primary job of this component is to monitor the runtime environment and collect this system information. All collected information are processed in another component called **DMaker**, which is also responsible for making the migration decision if necessary.

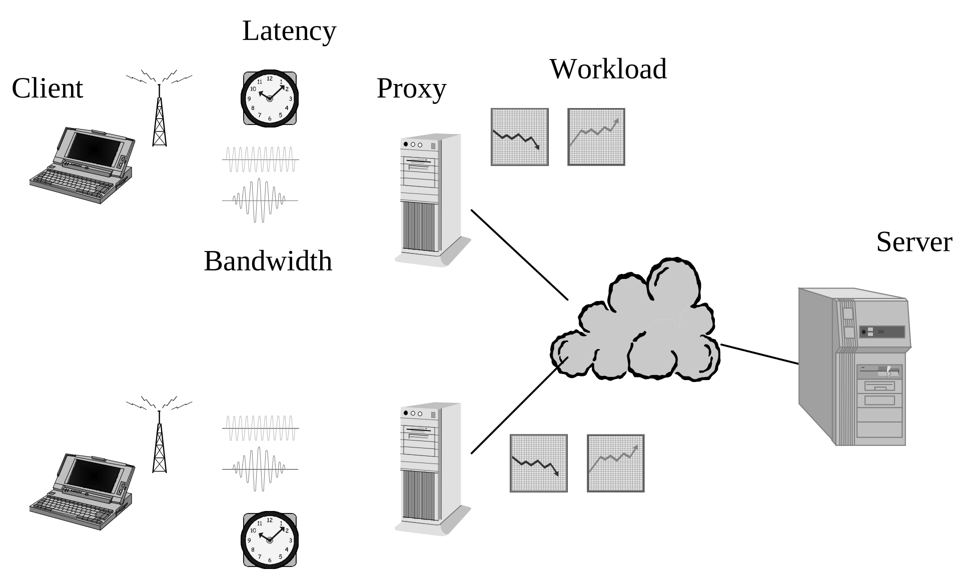

Figure 6.2 System Parameters

In the designed infrastructure, all proxy servers are connected in a fixed network. Unlike a base station, proxy servers do not relate to a certain geographical area. A proxy server may be responsible for one or more than one base stations at one time. Different proxy servers may connect to multiple base stations. On the other hand, each base station may connect with more than one proxy server, see Figure 6.3. Base stations A and B may connect to both proxy servers A and B, respectively.

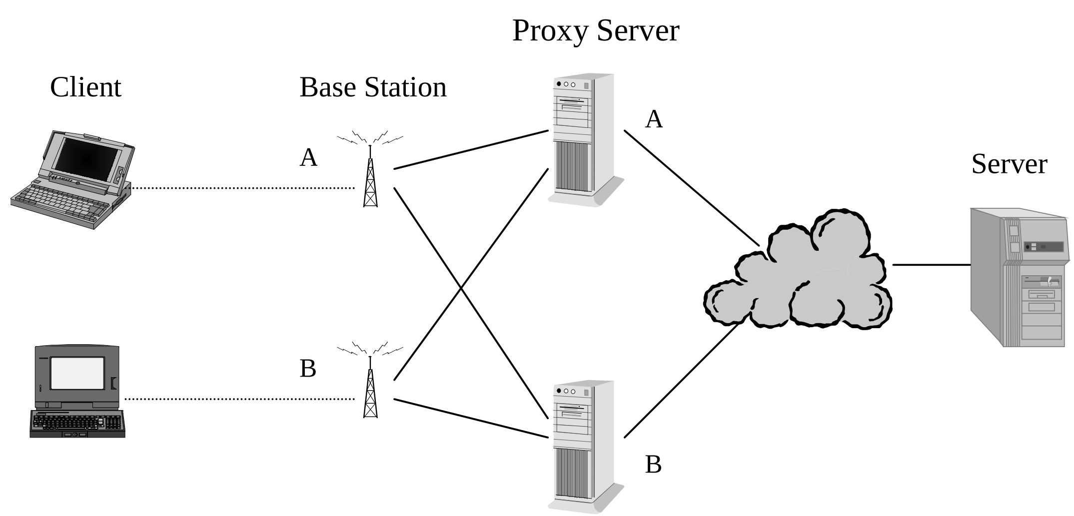

Figure 6.3 Multi-Connections for a Proxy Server

The first important system information we are going to investigate is the current machine load. Several environment parameters, such as available memory, and processor run queue length, might be used to describe the current machine' load.

- Committed Bytes in Use: Committed Bytes in Use is the ratio of the Committed Bytes to the Commit Limit. This represents the amount of virtual memory in use.

- Processor Run Queue Length: Processor Run Queue Length is the average length of the run queue in units of processes within a certain time interval. We use the equation proposed in [3] to approximate for the average run queue length on Unix/Linux platforms, explained in Appendix B.

Instead of user idle time, the Run Queue Length is used as the primary system parameter in our design. Idle time reflects the amount of time the processor was idle. This data is often more accurate in low-load situations than the load average. However, once the CPU idle time is close to zero, it does not work well [3].

In our proposed model, the proxy server is located between the base station and the application server. We assume that all proxy servers are powerful machines. Because of the fact that more than one mobile client can connect to the same proxy server, this proxy server may also be overloaded. In consequence, many processes compete for limited resources on a proxy server and significantly slow down the processing time for all involved applications. In this case, we believe that the better choice is to move the application objects to another proxy server with free CPU and memory or a short processor queue, see Figure 6.4. These system parameters can be gathered at runtime by using local system calls on Unix/Linux platforms and a separate daemon process on Window 95/NT machines.

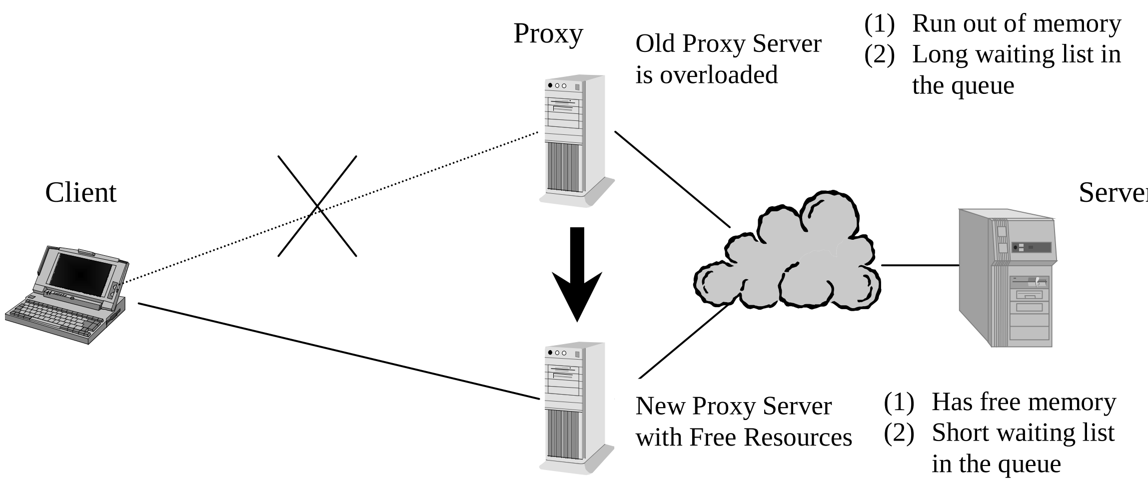

Figure 6.4 HandOff Triggered Because of Heavy Machine Load

The second system parameter is the time needed for a round-trip message between client and proxy server. In a wireless network, such as a WaveLAN, each WavePOINT base station covers a certain geographical area. The covered area is typically called a “cell”.

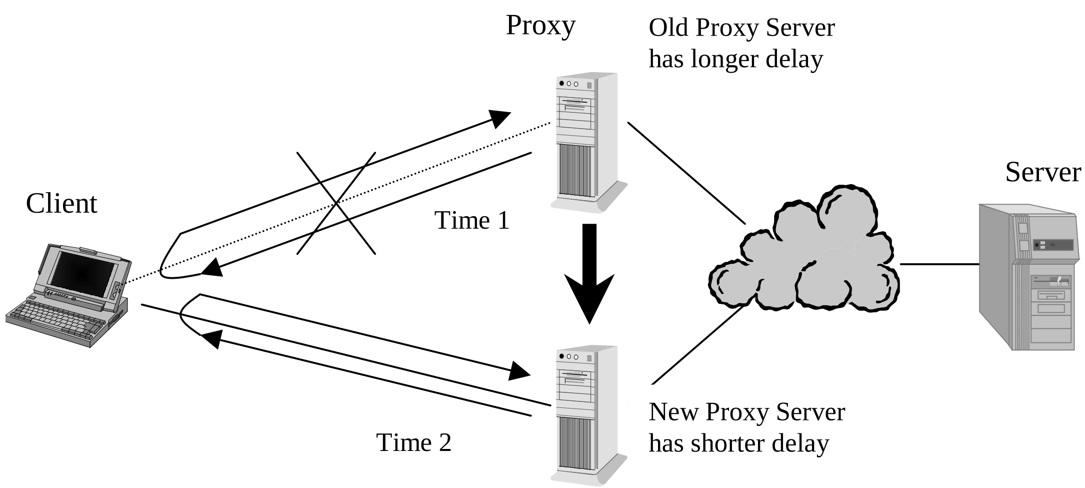

Figure 6.5 HandOff Triggered Because of Longer Message Delay

At the time when a mobile user is moving from one cell to another, the time (time1) needed for a round-trip message between client and the current proxy server increases, while, in contrast, the time (time2) needed for the same message between the client and one or maybe more than one neighboring proxy server(s) decreases. If time2 is less than time1, the mobile user is in a cell that is covered by another base station. Although it still can connect to the old proxy server, all objects running on the old proxy server should be moved to the new server in order to improve the response time. This parameter can be collected by periodically sending a synchronous round-trip message from a proxy server to a client device. The size of the idle message is kept in small and constant. This process could be done in a separate light-weight thread.

The bandwidth of the wireless link between portable devices and proxy servers is not stable. Many factors, such as weather, shadows, and noises, may reduce the bandwidth of the link. Lower bandwidth causes longer transmission times. For an application that requires intensive use of links, such as downloading files and playing movies/music, low bandwidth may cause the program to perform extremely poor. This is not acceptable in real life. We agree that if the bandwidth available on the current link is less than on another link, the HandOff process should be triggered to happen immediately. See Figure 6.6, which depicts proxy migration because of unbalanced bandwidth of two wireless links.

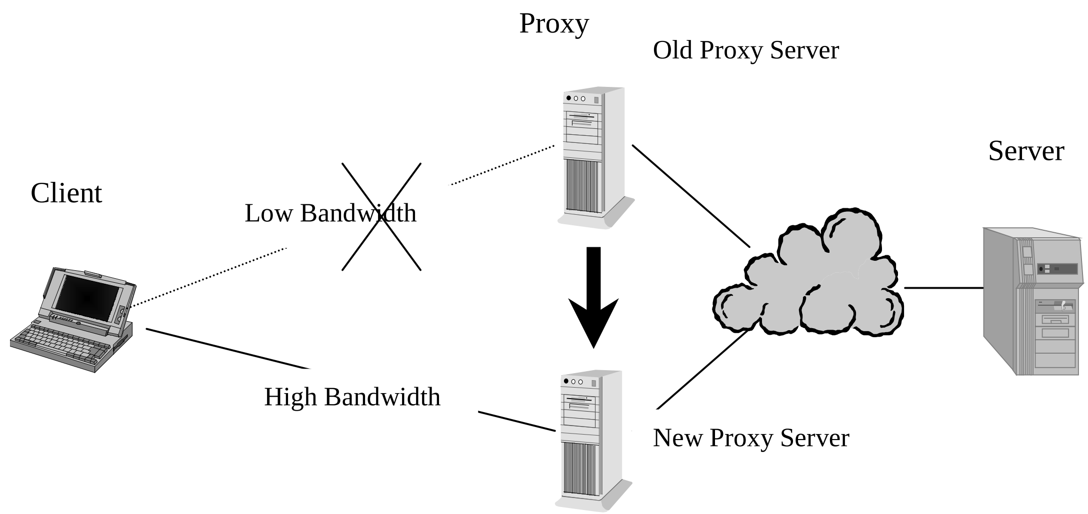

Figure 6.6 HandOff Triggered Because of a Poor Link

The problem is how to collect the bandwidth information at runtime. One way could be to generate a message with large size and pass it over the link. From the elapsed time and the size of the message, the current bandwidth could be estimated roughly. However, this obviously is not the solution we want. The message itself is large and repeatedly sending large messages causes unnecessary network traffic. Another method is to use a piece of software which reports the amount of bytes, in a certain period of time, passing through the link. This piece of software acts more like a snooping device, which monitors the link as required. Unfortunately, we do not have this kind software available at the moment, so we will not concentrate on this aspect in this thesis.

6.2.2 GON Strategy

The GON (Group of Neighbors) strategy is the basis of our design. In a large wired computer network, all hosts are connected together via high bandwidth mediums. Any two hosts in the network can talk to each other directly, if they are immediately connected, or via some other hosts in between if they are not connected directly. For any network members, two kinds of connections, direct connection and indirect connection, are used to connect others in the network. Any single host in the network has at least one direct connection and many indirect connections. In our design, we call those directly connected host(s) local neighbor(s), and indirectly connected host(s) are remote neighbor(s).

The GON strategy is to group a host and all its local neighbor(s) together as a group. Any connected network member has its own group. And any network member host could be the one running parts of the application. We define this host as a center of the group. The group contains the center host itself plus its local neighbors. Of course, group sizes normally are different because different hosts have different local neighbors. See Figure 6.7 for the details.

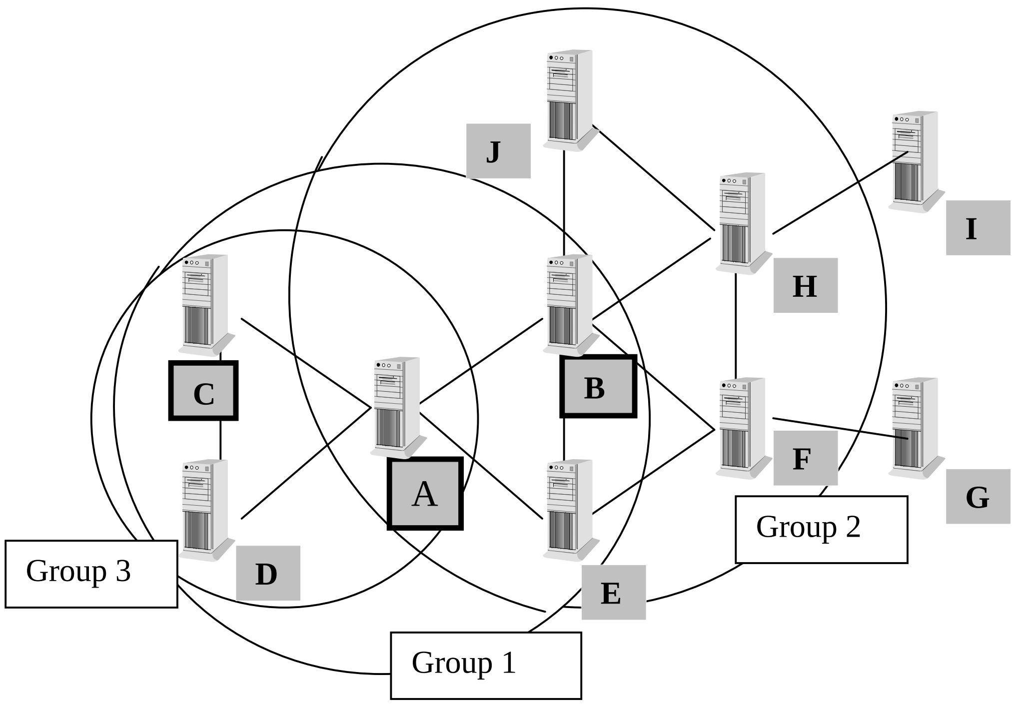

Figure 6.7 Grouping of Neighboring Proxy Servers

In the figure, there are a total of ten hosts connected in a network. Each of them is labeled with a letter. Three groups derived for hosts A, B, and C, respectively, are presented. Group 1 has five members, A, B, C, D, and E, where host A is the center of the group. Group 2 has center host B and its local neighbors, A, E, F, H, and J. Group 3 is derived from host C and has other two members, A and D. Clearly, any single host can belong to different groups.

In our design, these labeled hosts are proxy servers, but in reality, the network could be much larger. As an application running on a proxy server, the infrastructure is responsible to monitor the runtime environment at local and remote servers. This is done by sending messages among hosts. Moreover, these messages are sent periodically because environment information is updated. Apparently, to keep sending these messages all over the network is not efficient. The situation is even getting worse if many applications are running on the network. We use the GON strategy to avoid unnecessary message passing all over the network. Only members in the current group send or receive messages. Any proxy servers outside the group are not involved. For example, if an application is running on the proxy server A (see Figure 6.7), only group members A, B, C, D, and E are in contact.

A lookup table is used at each proxy server in order to keep the record of local neighbors' address information. Therefore, every proxy server in the network knows its local neighbors only. Network Operator is responsible to decide how to populate those tables. The GON strategy reduces unnecessary network traffic dramatically, and it allows to fit the designed infrastructure into a large scale network.

6.3 Proposed HandOff Algorithm

We described the environment and the GON strategy in the last section. In this section, we are talking about the HandOff protocol. Figure 6.8 shows the actual protocol for migrating object(s) from a proxy server to another proxy server,

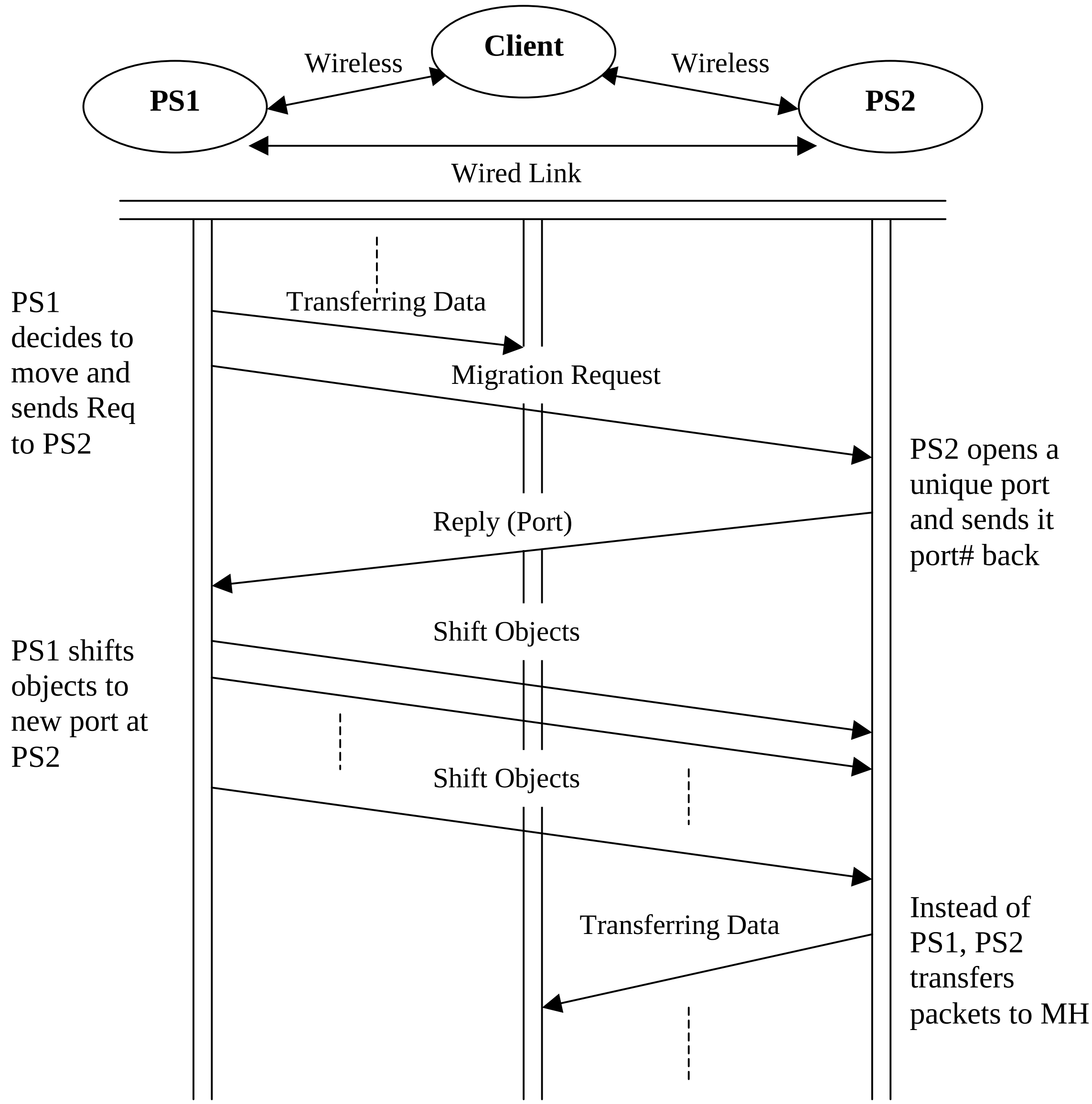

Figure 6.8 Message Passing Diagram for HandOff Protocol

The following shows the sequence of the protocol,

1. **DMaker** at proxy server 1 (PS1) makes the migration decision. (See Section 5.2.3 and pseudo code)
2. PS1 sends a handoff request message to the Proxy server 2 (PS2). If PS2 can accept the request, PS2 opens a port and returns a reply message including the port number to PS1.
3. PS1 starts to shift object(s) to the port at PS2. If migration failed, the object will be resumed at original place, PS1. An error message will be returned.
4. After being successfully migrated, object(s) at PS2 resume execution. Data packets are sent to the client/MH (Mobile Host) from PS2 but PS1. Routing information is updated.

In this protocol, the MH is not involved during the migration process. The infrastructure supports everything: finding a target proxy server, opening a new server at the new location, and shifting object(s) to the new place.

The approach also has another advantage. Traditional protocols stop transmitting data packets to MH before sending out the handoff request message. However, our approach keeps transmitting data packets to MH till PS1 starts to migrate object(s). Users only suffer a short suspension for object(s) migration.

The pseudo code of the algorithm is listed in Figure 6.9.

```
#define load_delta = 1.0;
#define lat_delta = 500;
#define repeat_times = 6;
```

```
// all neighbors are in memory trouble, no moving...
if (all neighbor' proxy_memory <= 0.2)
     return null;

// if the current proxy doesn't have any memory left
if (my proxy_memory <= 0.5)
     move to anyone has better memory condition

// get total loading and latency for all members
total_loading = 0;
total_latency = 0;
idx = 0;

while(more member){
     total_loading += proxy_loading(idx);
     total_latency += proxy_latency(idx);
     idx++;
}

// get average loading and latency between all members
average_loading = total_loading / idx;
average_latency = total_latency / idx;

// make a range, so, if proxy's loading is over the
// upper_bound, the proxy is busy
// in contrast, if proxy's loading is less than the
// lower_bound, the proxy is less loaded
loading_upper_bound = average_loading + loading_range;
loading_lower_bound = average_loading – loading_range;

// make a range, so, if proxy's latency is over the
// upper_bound, the proxy has high latency
// in contrast, if proxy's latency is less than the
// lower_bound, the proxy has the better codition
latency_upper_bound = average_latency + latency_range;
latency_lower_bound = average_latency – latency_range;

// compare my loading to the upper_bound
if (my proxy_loading > loading_upper_bound){
     idx = 0;
     smallest_loading = proxy_loading(idx);
     // get the proxy, which has the smallest loading, in
     // the group
```

```
     while(more member){
          idx++;
          if (samllest_loading > proxy_loading(idx))
               smallest_loading = proxy_loading(idx);
     }

     // record the destination proxy
     target_proxy = the proxy with smallest_loading;

     // increase the load_count
     load_count++;
}
else load_count = 0;

// compare my latency to the upper_bound
if (my proxy_latency > latency_upper_bound){
     idx = 0;
     shortest_latency = proxy_latency(idx);
     // get the proxy, which has the shortest latency, in
     // the group
     while(more member){
          idx++;
          if (shortest_latency > proxy_latency(idx))
               shortest_latency = proxy_latency(idx);
     }

     // record the destination proxy
     target_proxy = the proxy with shortest_latency;

     // increase the latency_count
     latency_count++;
}
else latency_count = 0;

// repeat_times == 6, which means the current proxy has
// been in busy for last 6*5= 30 seconds
if (load_count == repeat_times){
     // if the destination proxy is not in latency trouble,
     //go move
     if (target_proxy.proxy_latency < latency_upper_bound)
          move to target_proxy;
     else return null;  // otherwise, no moving
}
```

```
// repeat_times == 6, which means the current proxy has
// been in latency trouble for last 30 seconds
if (latency_count == repeat_times){
     // if the destination proxy is not in loading trouble,
     // go move
     if (target_proxy.proxy_loading < loading_upper_bound){
          move to target_proxy;
     }
     else return null;  // otherwise, no moving
}
```

Figure 6.9 Migration Algorithm in Pseudo Code

The algorithm given above is simplified. The problem of infinite migration may happen based on this algorithm. For example, if more than one proxy all try to migrate some of their work load to a same destination proxy, the work load on this proxy is increased rapidly. Then, possibly, the migration process will happen again and again among those proxies. In the future work, some ideas could be used to overcome the weakness.

- Using a migration count valuable to control the migration times.
- Two delta values (**load_delta** and **lat_delta**) should be dynamic according to the value of the migration count valuable and application size.
- After migration, objects could be forced to execute at new location for a short period of time before migration.

Chapter 7 Experiment Setup and Evaluation

7.1 Objective

As indicated in previous chapters, the HandOff process happens in several cases, for example when the proxy server is overloaded, has high latency, has poor bandwidth, or the user changes location. In our experiments, however, we only focus on the first two cases: proxy server overloaded and high latency. In Chapter 5 and Chapter 6, we already described the infrastructure, which supports collecting run-time system information, making the migration decision, and moving the objects between servers. The decision-making algorithm and the handoff protocol have been developed in our system in order to achieve the goal that, after migration, the workload on each proxy server is either close to the average or the proxy with high latency is avoided. We collect the Average Number of Processes and the latency values at runtime. The time needed to successfully migrate objects from one server to another is measured. In general, the performance of all running mobile applications is improved due to two reasons: load spread among multiple Proxy servers, and avoid high latency proxies.

In this chapter, we are describing the experiments we have done and discuss the test results. In Section 7.2, we describe the experimental equipment we are using for the test and the general experimental testbed setup. In Section 7.3, the experiment of each case is discussed and evaluated.

7.2 Testbed Setup

We used eight machines (Linux/NT) for testing purpose. Among those eight machines, two are client hosts, four are proxy servers, and two are application servers. The general experimental testbed setup is shown in the following figure.

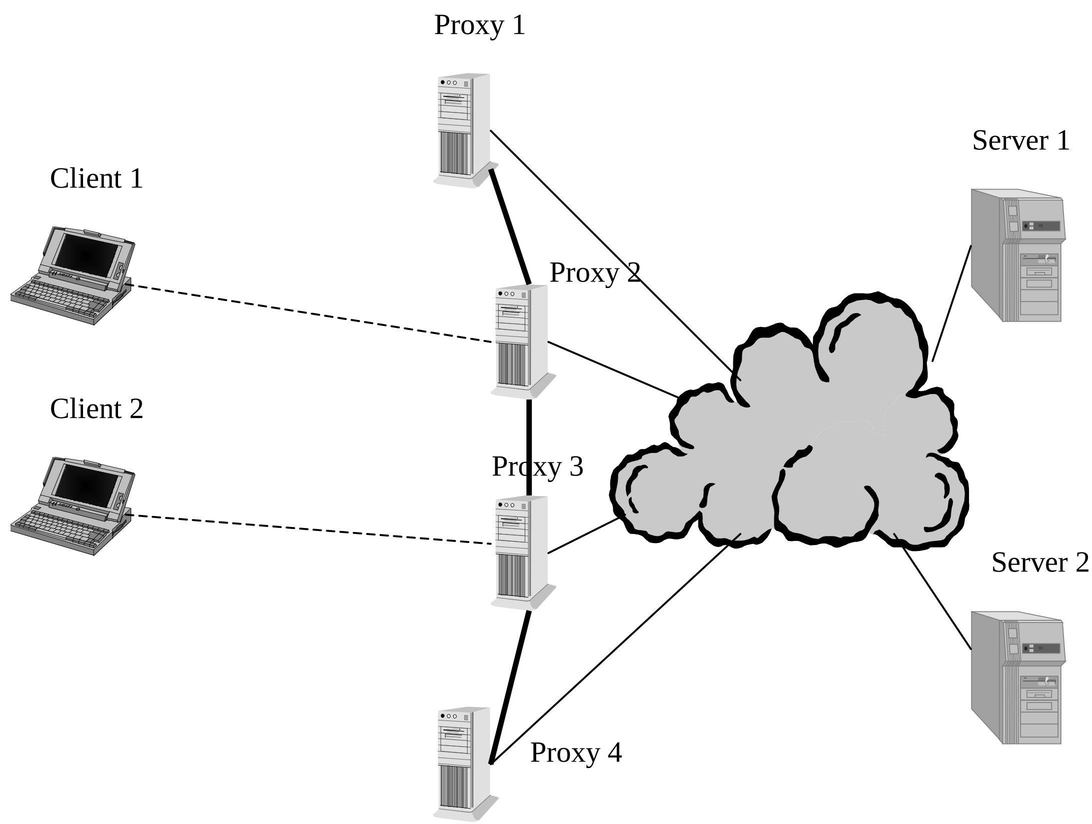

Figure 7.1 General Experimental Testbed

Figure 7.1 depicts the connections between hosts. Four proxy servers are powerful Linux machines with 128 MB memory connected together as a fixed network (switched 100 Mbps Ethernet). Server 1 and Server 2 could be any type of servers such as ftp and http servers. Two mobile devices are linked to the proxy server network over the ‘wireless’ link, which is emulated by using a device driver. During our experiments, we simulate two types of ‘wireless’ link between client devices and proxies. One is the normal link with bandwidth around 10 Mbps and low latency. The other has the same bandwidth but introduces high latency.

We conducted experiments in five different situations:

- **Single application**
  - **Proxy server is overloaded** (Case One)
    The number of jobs in the queue waiting for CPU time is larger than the average number in the group (see description of GON in Section 6.2.3).

  - **Proxy server has longer delay** (Case Two)
    A mobile user moves away from the current proxy server, causing longer delay (see Figure 6.5 for details).

- **Multiple applications**
  - **Proxy server is overloaded** (Case Three)
    The number of jobs in the queue waiting for CPU time is larger than the average number in the group. One of the applications is shifted to a neighbor.

  - **Proxy server has longer delay** (Case Four)
    Mobile users move away from the current proxy server, causing longer delay. Applications are shifted to neighbors.

- **Cascading migration** (Case Five)

  Workload is shifted to a neighbor from the current proxy because of its longer delay. This results in an overload situation at the neighbor. One of its applications is then migrated to a neighbor to balance the load.

7.3 Evaluation

7.3.1 Case One: Single application – Proxy server is overloaded

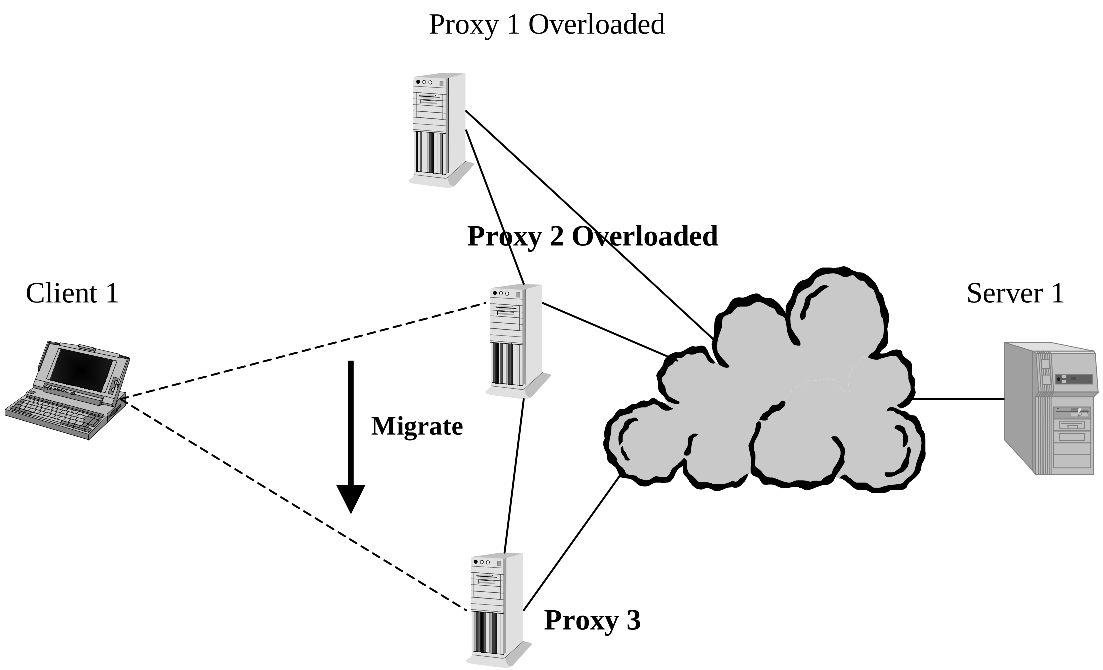

Figure 7.2 Single Application – Overload at Proxy

We are running a single application, MPEG player. The application graphic user interface is running on Client 1. Proxy 2 is the current proxy host, on which seven objects are running. On Server 1, only one object is active. The detailed description of the MEPG player application is given in Section 4.4.2.

On Proxy 2, in order to keep the host in a "busy" state, we also run some other processes, such as a Java version of the MPEG player, as well. The reason we are doing this is because a single Java version MPEG player creates 3-4 threads competing for the CPU time and it is a plain process without interaction with the infrastructure. Proxy 1 and Proxy 3 are two local neighbors of the current proxy – Proxy 2. We add some workload to Proxy 1, on purpose, in order to let infrastructure choose the Proxy 3, which has an empty queue, as the target proxy server of the migration.

	Before		**Migrating**		**After**	
	Proc.	**Lat.**	**Proc.**	**Lat.**	**Proc.**	**Lat.**
Proxy 1	2.35	1343	2.32	1301	2.30	1339
Proxy 2	2.23	1114	**3.45**	**1241**	2.31	1276
Proxy 3	0.13	978	0.11	1020	**1.22**	**1121**

Table 7.1: Single Application – Overload at Proxy

Proc. : Average Number of Processes Sharing the CPU

Lat. : Latency Values between Proxies and Mobile host in Milliseconds

Analysis:

Before Column: Before starting the application, Proxy 1, 2, and 3 have very close latency values: 1343, 1114, and 978 ms, respectively. Proxy 1 and 2 execute some processes sharing the CPU time, while Proxy 3 is almost idle.

Migrating Column: Some objects of the application are running on Proxy 2. The average process number increases up to 3.45, which triggers the migration moving objects to Proxy 3 because of its low load (0.11).

After: After migration, the Average Process Number on Proxy 2 drops down to 2.31, in contrast, Proxy 3 has a higher load. We see that the workloads on three Proxies are more balanced. The mobile application is running in a better environment with more CPU time available to it.

7.3.2 Case Two: Single application – Proxy server has high latency

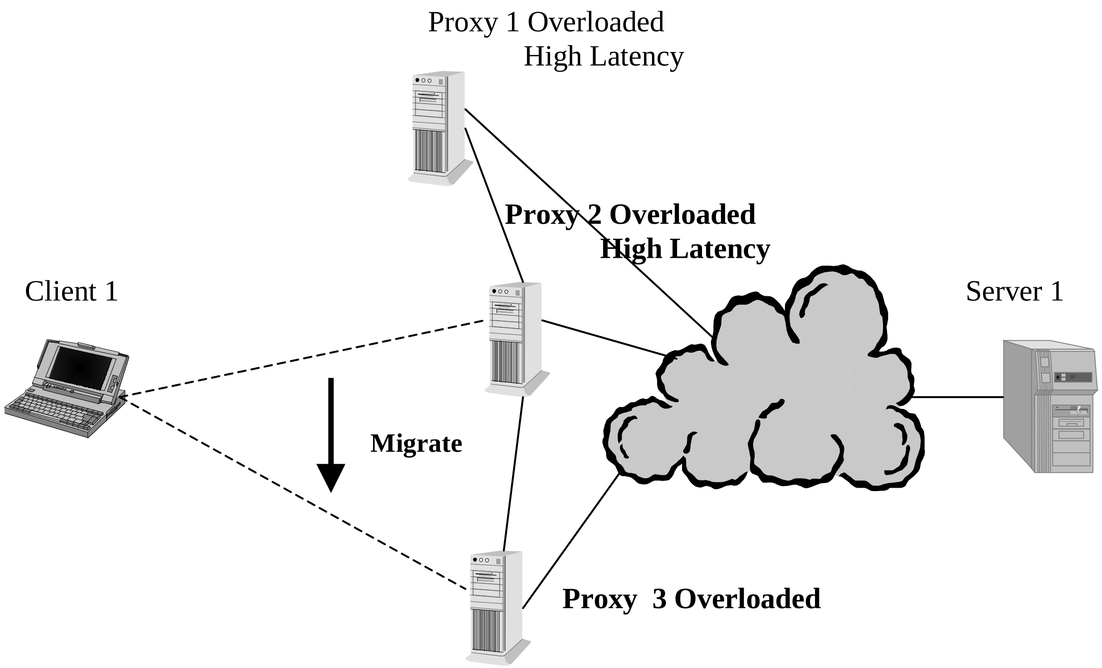

Figure 7.3 Single Application – Proxy with High Latency

Like before, there is only one MPEG player application running on the system. Proxy 2 is the current proxy server holding most of the objects of the application. Proxy 1 and Proxy 3 are two neighbors of the Proxy 2. In this case, we are not going to put extra workload on the Proxy 2. Instead, we increase the latency between Proxy 2 and Client 1. Compared to the link with increased latency between Client 1 and Proxy 2, the link between Client 1 and Proxy 3 is unchanged. In addition, we put extra workload on Proxy 1 and Proxy 3 to keep the three proxy servers similarly loaded. Since the latency of the link between Proxy 1 and Client 1 is increased, Proxy 3 is the preferred target for Proxy 2 to shift its workload.

	Before		Migrating		After	
	Proc.	**Lat.**	**Proc.**	**Lat.**	**Proc.**	**Lat.**
Proxy 1	2.16	1205	**2.28**	**1316**	2.28	1316
Proxy 2	1.50	1322	**2.35**	**12000**	**1.40**	**12000**
Proxy 3	2.32	750	**2.26**	**809**	**3.12**	**785**

Table 7.2: Single Application – Proxy with High Latency

Proc. : Average Number of Processes Sharing the CPU

Lat. : Latency Values between Proxies and Mobile host in Milliseconds

Analysis:

Before Column: From Table 7.2, workload is not a reason to trigger the migration. Before running the application, all three proxies are in balance and have similar latency values.

Migrating Column: Objects are running on the Proxy 2 so that the Average Number of Processes is increased to 2.35. The three proxies have almost similar workload (2.28, 2.35, and 2.26). Therefore, there is no migration triggered because of unbalanced workload. However, at this point, Proxy 2 introduces very high latency (12000 ms) while Proxy 3 shows relatively low latency (809 ms). Migration is triggered and objects are migrated from Proxy 2 to Proxy 3.

After Column: After migration, Proxy 3 has more workload (3.12), which is high, but it offers much lower latency. Apparently, Proxy 3 offers a better execution environment for the mobile application than Proxy 2.

7.3.3 Case Three: Multiple applications – Proxy server is overloaded

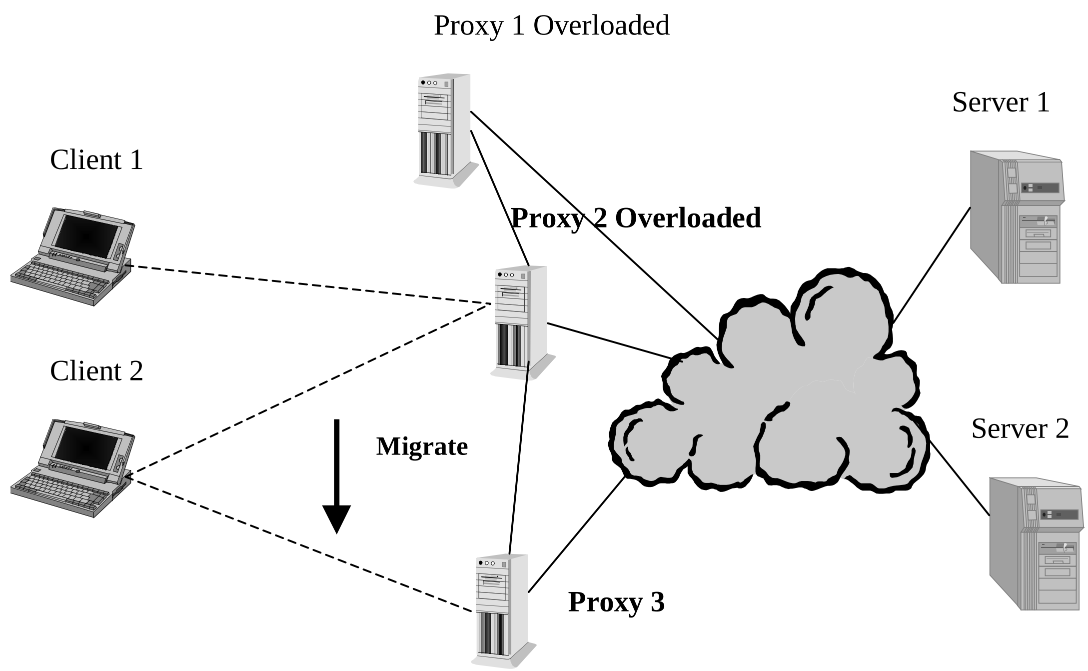

Figure 7.4 Multiple Applications – Overload at Proxy

In cases three and four, we are going to test the multiple applications scenarios. Two MPEG player applications are running on Client 1 and Client 2, respectively. Proxy 2 is the current proxy server for both clients, and is also the center of the current group (including Proxies 1, 2, and 3). It has two sets of objects running in different logic spaces. Similar to case one, some extra workload is added to Proxy 1 and Proxy 2. The infrastructure detects that the Proxy 3 has an empty processor queue. Objects of one of two applications on the Proxy 2 are migrated to the Proxy 3. In Figure 7.4, Client 1 and Client 2 connect to Proxy 2 and Proxy 3, respectively, after migration.

	Before		Migrating		After	
	Proc.	**Lat.**	**Proc.**	**Lat.**	**Proc.**	**Lat.**
Proxy 1	3.32	2645	3.39	2799	3.34	2802
		2448		2552		2300
Proxy 2	1.18	2712	**3.62**	**2600**	2.27	2644
		2390		**2411**		2425
Proxy 3	0.18	2476	0.17	2533	**1.21**	**2617**
		2621		2798		**2588**

Table 7.3: Multiple Applications – Overload at Proxy

Proc. : Average Number of Processes Sharing the CPU

Lat. : Latency Values between Proxies and Mobile host in Milliseconds

Note : The value in the upper row of the latency column represents the latency from this proxy to Client 1, and the one in the lower row represents the latency from this proxy to Client 2.

Analysis:

Before Column: Before connecting any applications to Proxy 2, Proxy 1 and 2 have some processes – background load (3.32 and 1.18) sharing their CPUs while Proxy 3 is very little occupied (0.18). Latency values between proxies and the two mobile hosts are in balance.

Migrating Column: From Client 1 and Client 2, respectively, we run two MPEG player applications and share the Proxy 2. The workload on Proxy 2 increases rapidly. At the point of migration, the highest value of workload (3.62) on Proxy 2 causes the migration. The destination is Proxy 3 which is really idle (0.17). The proxies are unbalanced.

After Column: After migration, workload on Proxy 2 is decreased (2.27), while the one on Proxy 3 increases to (1.21).

7.3.4 Case Four: Multiple applications – Proxy server has high latency

Multiple applications are running on Proxy 2. Group members, including Proxies 1, 2, and 3, are all overloaded with similarly long CPU queue lengths. Since Proxies 1 and 2 introduce high latency, workload on Proxy 2 is shifted to Proxy 3 even though it is already overloaded.

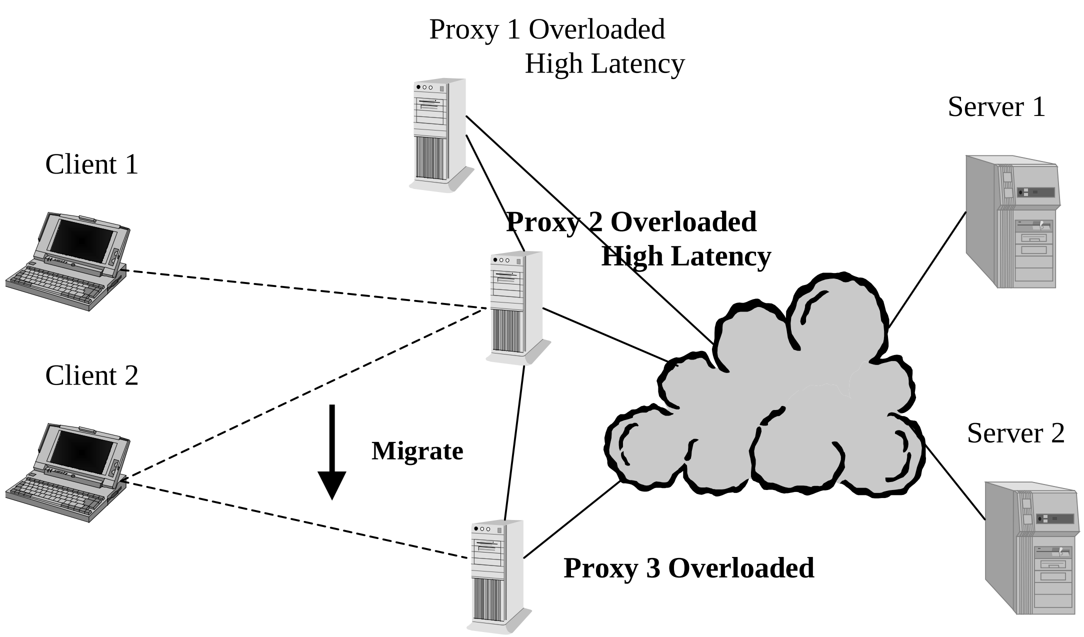

Figure 7.5 Multiple Applications – Proxy with High Latency

	Before		Migrating		After	
	Proc.	**Lat.**	**Proc.**	**Lat.**	**Proc.**	**Lat.**
Proxy 1	3.29	2431	3.30	10000	3.22	10000
		2500		10000		10000
Proxy 2	1.21	2600	**3.62**	**12000**	2.31	12000
		2033		**12000**		12000
Proxy 3	3.20	1687	3.17	1477	**4.12**	**1520**
		2221		1908		**2123**

Table 7.4: Multiple Applications – Proxy with High Latency

Proc. : Average Number of Processes Sharing the CPU

Lat. : Latency Values between Proxies and Mobile host in Milliseconds

Note : The value in the upper row of the latency. column represents the latency from this proxy to Client 1, and the one in the lower row represents the latency from this proxy to Client 2.

Analysis:

Before Column: We pre-set workloads on each proxies shown in the column (3.29, 1.21, and 3.20). Latency values are in balance (all around 2000 ms) at the time.

Migrating Column: We start two applications on Proxy 2, so that its workload increases up to 3.62. Among the three proxies, workloads are in balance (3.30, 3.62, and 3.17). We increase the latency values on both Proxy 1 and 2 (10000 ms and 12000 ms). Afterwards, apparently, migration is triggered and some objects on Proxy 2 are shifted to Proxy 3 because of its low latency (1477 ms and 1908 ms).

After Column: After migration, the workload decreases on Proxy 2 (2.31) and increases on Proxy 3 (4.12).

7.3.5 Case Five: Cascading Migration

Here, Clients 1 connects to Proxy 2, which is the center of the group (including Proxy 1, 2, and 3). All three proxies have similarly long CPU queue lengths. In addition, Proxies 1 and 2 also introduce longer latencies. Proxy 4 is idle.

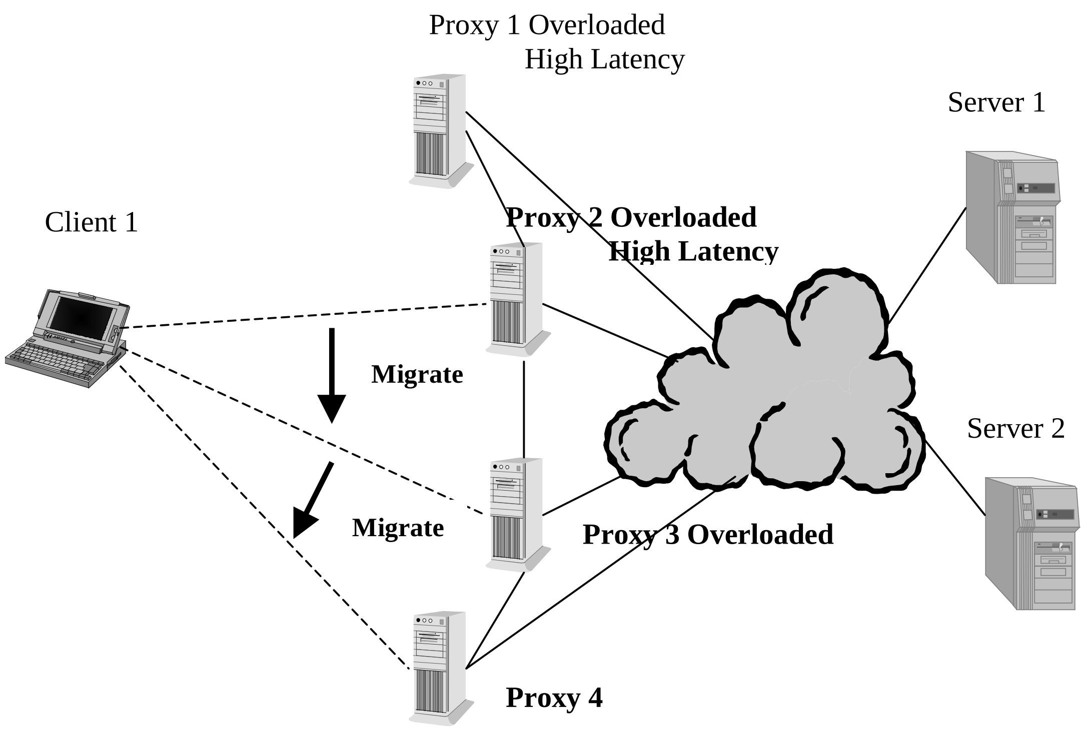

Figure 7.6 Cascading Migration

Like in Case two, workload on Proxy 2 is shifted to Proxy 3 because of its lower latency. After migration, Proxy 3 becomes the center of another group (including Proxies 2, 3, and 4), and it becomes overloaded because of the additional workload migrated from Proxy 2. Since Proxy 4 is lightly loaded and has low latency, some workload at Proxy 3 is shifted to Proxy 4.

	Before		**Migrating(1)**		**Migrating(2)**		**After**	
	Proc.	**Lat.**	**Proc.**	**Lat.**	**Proc.**	**Lat.**	**Proc.**	**Lat.**
Proxy 1	3.15	1532	3.34	1437	3.37	1532	3.37	1532
Proxy 2	1.42	1335	**2.29**	**12000**	1.75	12000	1.75	12000
Proxy 3	2.32	679	2.22	621	**2.89**	**732**	1.92	732
Proxy 4	0.12	0	0.12	457	0.11	599	**1.21**	**599**

Table 7.5: Cascading Migration

Proc. : Average Number of Processes Sharing the CPU

Lat. : Latency Values between Proxies and Mobile host in Milliseconds

Analysis:

Before Column: Four Proxies are involved in this case. Before running the application, we set up the workload and latency on each proxy shown in the Before Column.

Migration (1) Column: We start running the application on Proxy 2. Proxies 1, 2, and 3 are in the group, and Proxy 2 is the center of the group. These three proxies are approximately evenly loaded (3.34, 2.29, and 2.22). We change the latency value on Proxy 2 to 12000 ms, which is very high compared to the others (1437 ms and 621 ms).

This unbalanced latency value triggers the migration, and objects on Proxy 2 are moved to Proxy 3, which has the lowest latency value (621 ms) in the group.

Migration (2) Column: After the first migration, a new local group is created and Proxy 3 becomes the center of the group. We see that within this group, the three proxies are not load- balanced (1.75, 2.89, and 0.11). Proxy 3 has the highest workload. The infrastructure detects that Proxy 2 still introduces high latency, however, Proxy 4 is an ideal migration candidate (the lowest value of workload, 0.11, and small latency value, 599 ms, in the group). The migration decision is made and objects are moved again to Proxy 4.

After Column: Now, objects are running on Proxy 4, which is also the center of the new group. Two group members, Proxy 3 and Proxy 4, are load-balanced (1.92 and 1.21) and also have no big latency differences (732 ms and 599 ms).

7.3.6 Migration Time Measurement

The time needed to migrate objects from one server to another server is measured in our experiments. We want to point out that we do not measure the run-time size of an object or an application. Our measurements are based only on the same object(s) and application (MPEG player). For a single object of MPEG player, migration costs around 2160 ms in average. The application contains nine objects in total at runtime, but only seven of them are involved in a migration. The average time needed to finish the migration is about 17700 ms.

The results show that the costs of migration are high. We believe that the migration process itself should be optimized. Further, migration costs should be considered combining with some other factors, such as run-time size of an object or an application. If expected improvement is small, it is probably better to not trigger the migration.

Chapter 8 Summary and Future Work

8.1 Summary

A proxy-based infrastructure is designed and implemented to support adaptive mobile applications. In general, a proxy takes the form of an intermediary inserted between two communication endpoints such as a client and server. The purpose of using a proxy is to improve the performance and reliability of mobile applications running in wireless mobile environments. In order to guarantee that mobile users have the best services, application objects running on one proxy server can migrate to another at run-time. The proxy-based infrastructure collects run-time information without the users' knowledge. It supports multiple applications sharing common resources but belonging to different subspaces. A decision-making algorithm is developed to locate a destination proxy of the migration based on collected run-time information, and to trigger the handoff process. The handoff protocol is presented in detail in this work.

Throughout the course of this thesis, we presented a detailed design and implementation of the infrastructure along with the handoff protocol and decision-making algorithm. A sample application, MPEG player, which runs across a mobile client and a proxy server over the Voyager system, is developed.

8.2 List of Contributions

We feel that this work has several contributions listed below:

- The infrastructure is independent from the user applications and system
  The infrastructure is a set of components running on the Proxy servers. It provides services to the user applications and collects information from the system.

- Object and process migration support
  The infrastructure supports proxy migration in order to get better services/connectivity.

- Multiple application support
  The infrastructure supports multiple mobile applications at same time. Applications are running in different working spaces.

- User mobility
  The infrastructure is not only balancing the workload among proxies, but also supporting user mobility.

- Environment information run-time collection
  The infrastructure collects environment information, such as memory and process queue length, at run-time

- A MPEG player application is developed and can be used in future work.
- An Object Migration Monitoring Interface program is implemented.

The infrastructure system supports the combination of these features, but still provides many possible future areas of research.

8.3 Future Work

We described the handoff process, (object migration), between proxy servers in the previous chapters. The related work of object migration support between client and proxies is already being explored in a companion project. In that case, the division of the application logic is dynamically changed at run-time based on the system information collected from client device and proxy. Daemons are running on mobile devices and proxies collecting system information. If the bandwidth of the wireless link is low, some workload (objects) at the proxy should be moved to the client side. In contrast, if the battery power of mobile device is low, some objects may shift to the proxy side. Object migration between clients and proxies should be transparent to mobile applications.

In Section 6.2.1, we mentioned that we do not have a proper method to gather bandwidth information at run-time. However, the bandwidth information is not only useful but also important for deciding current link conditions. We suggest modifying the device drives to detect the bandwidth rather than sending large size messages. In the future consideration of this work, such a device drive should be implemented and used to collect run-time bandwidth information. Alternatively, network management protocols such as SNMP could potentially provide the information.

We have not addressed the object migration costs yet. To precisely calculate the cost of the object migration is a challenging task, as well as estimating the size of objects at run-time. In the future development of this work, the size of the object and the cost of the migration should be considered as important parameters for determining the timing of migration for particular objects.

During our research, we have only implemented a mobile object-oriented MPEG-player application for testing purpose. Some other applications, such as a Web browser, an object-oriented distributed calendar, and a newsgroup reader, can be developed using the new version of Voyager platform. The reliability of the system can be tested further based on running those different applications.

Again, we only used one application in our test. It is important to have different applications running on a same proxy server because, in this case, a new problem is raised that how infrastructure is to decide which application should be moved away from the current proxy server at run-time. Some information, such as the size of the application (number of objects), could be gathered in order to help infrastructure to make the right decision. For example, based on the size of the applications, the infrastructure should always move the application with the largest size away. This is only one possible solution. We may do some more research works in the future.

The infinite migration problem may happen in the current version algorithm. We will improve the migration stability in the future based on the ideas stated before (page 72).

In the future work, we still need to consider the scalability to large cellular networks. Some other mobile code platforms should be explored in further.

Bibliography

[1] Angin O., A. T. Campbell, M. E. Kounavis, and R. R.-F. Liao, The Mobiware Toolkit: Programmable Support for Adaptive Mobile Networking, IEEE Personal Communications, August 1998, http://wwwcomet.columbia.edu/wireless

[2] Appenzeller G., K. Lai, P. Maniatis, M. Roussopoulos, E. Swierk, X. Zhao, and M. Baker, The Mobile People Architecture. Technical Report CSL-TR-99-777, January, 1999, http://mosquitonet.stanford.edu/mosquitonet.html

[3] Arndt O., B. Freisleben, T. Kielmann, and F. Thilo, Dynamic Load Distribution with the WINNER System, Proc. Workshop Anwendungsbezogene Lastverteilung (ALV'98), Munchen, Germany, 1998, pages 77-88. Published as Technical Report TUM-I9806, SFB 342/01/98 A, SFB 0342, Technische Universitat Munchen.

[4] Aurrecoechea C., A. T. Campbell, and L. Hauw, A Survey of QoS Architectures, http://www.ctr.columbia.edu/comet/members.html

[5] Badrinath B. R., A. Acharya, and T. Imielinski, Designing distributed algorithms for mobile computing networks, In Proc. Of the 14th Intl. Conf. On Distributed Computing Systems, May 1995, http://pdpwww.csie.nctu.edu.tw/doc/distr-mob-net.ps.gz

[6] Baggio A., System Support for Transparency and Network-aware Adaptive in Mobile Environments, Project SOR http://www-sor.inria.fr/ April 1998

[7] Balakrishnan H., S. Seshan, and R. H. Katz, Improving Reliable Transport and Handoff Performance in Cellular Wireless Networks, ACM Wireless Networks, December 1995, http://wind.lcs.mit.edu/~hari/papers

[8] Balakrishnan H., V. N. Padmanabhan, and R. H. Katz, The Effects of Asymmetry on TCP Performance, Proc. 3rd ACM/IEEE Intl. Conference on Mobile Computing and Networking (MobiCom). Budapest, Hungary, Sept. 1997, http://wind.lcs.mit.edu/~hari/papers

[9] Baldi M., and G. P. Picco, Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications 20th International Conference on Software Engineering (ISCE '98), Kyoto, Japan, Apr. 1998, http://www.polito.it/~baldi/curriculum/curriculum.htm#conference

[10] Banicescu I., S. H. Russ, S. Ghafoor, and M. Bilderback, Experiences from Integrating Algorithmic and Systemic Load Balancing Strategies, http://www.cs.msstate.edu/~ioana/publications.html

[11] Banicescu I., and S. F. Hummel, Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations. Proceedings of Supercomputing'95 Conference, December 1995, http://www.cs.msstate.edu/~ioana/publications.html

[12] Bates J., D. Halls, and J. Bacon, A Framework to Support Mobile Users of Multimedia Applications, University of Cambridge Computer Laboratory, Baltzer Journals 1996, http://www-tkn.ee.tu-berlin.de/bibl/ps/index.html#CAM

[13] Benno J. Overeinder, Peter M. A. Sloot, Robbert N. Heederik and L. O. Hertzberger, A Dynamic Load Balancing System for Parallel Cluster Computing, Future Generation Computer Systems, 12(1), pp. 101-115, May 1996.

[14] Bisdikian B., P. Bhagwat, B. P. Gaucher, F. J. Janniello, M. Naghshineh, and P. Pandoh WiSAP: A Wireless Personal Access Network for Handheld Computing Devices, IEEE Personal Communications December 1998, http://207.127.135.8/pci/private/1998/dec/Bisdikian.html

[15] Bolliger J., and T. Gross, A Framework-Based Approach to the Development of Network-Aware Applications IEEE Transactions on Software Engineering, Vol. 24, No. 5, pp 376-389, May 1998

[16] Chang H., C. Tait, N. Cohen, M. Shapiro, S. Mastrianni, R. Floyd, B. Housel, and D. Lindquist, Web Browsing in a Wireless Environment: Disconnected and Asynchronous Operation in ARTour Web Express, Proceedings of the Third Annual

ACM/IEEE Conference on Mobile Computing and Networking Budapest, Hungary, September 1997, pages 260-269

[17] Chevalier P-Y., D. Hagimont, J. Mossiere, and X. Rousset de Pina, Object Migration in the Guide System Bull-IMAG Systems, 2m rye Vignate, F-38610 Gieres, France, ftp://ftp.imag.fr/pub/GUIDE/doc/publications/96-TSI-Guide-general-PUB.ps.gz

[18] Davies N., G. S. Blair, K. Cheverst, and A. Friday, Supporting Adaptive Services in a Heterogeneous Mobile Environment, In Proc. Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, U.S., December 1994, http://snapple.cs.washington.edu:600/mobile/mcsa94.html

[19] Davies N., S. Pink, and G. S. Blair, Services to Support Distributed Applications in a Mobile Environment 1st International Workshop on Services in Distributed and Networked Environments, Prague, Czech Republic, 1995, http://www.yeditepe.edu.tr/MaROS/biblio.html

[20] Dearle A., Toward Ubiquitous Environments for Mobile Users, IEEE INTERNET COMPUTING, pages 23 - 32, January-February 1998

[21] Duchamp D., Issues in Wireless Mobile Computing, In Proc. Third Workshop. on Workstation Operating Systems, IEEE, Key Biscayne FL, April 1992, pp. 1-7.

[22] Fasbender A., and F. Reichert, Any Network, Any Terminal, Anywhere, IEEE Personal Communications April 1999, http://207.127.135.8/pci/private/1999/apr/Fasbender.html

[23] Forman G. H., and J. Zahorjan, The Challenges of Mobile Computing, UW CSE Tech Report #93-11-03 from ftp.cs.washington.edu

[24] Fox A., S. D. Gribble, Y. Chawathe, and E. A. Brewer, Adapting to Network and Client Variation Using Infrastructure Proxies: Lessons and Perspectives, IEEE Personal Communications August 1998, http://gunpowder.stanford.edu/~fox/pubs.html

[25] Friday A. J., G. S. Blair, K. W. J. Cheverst, and N. Davies, Extensions to ANSAware for advanced mobile applications, In Proc. International Conference on Distributed Platforms, Dresden, February 27 – March 1, 1996, http://www.comp.lancs.ac.uk/computing/research/mpg/most/reports/icdp.adrian.ps

[26] Fuggetta A., G. P. Picco, and G. Vigna, Understanding Code Mobility IEEE Transactions on Software Engineering, Vol. 24, No. 5, May 1998, http://www.polito.it/~picco/listpub.type.html

[27] Fujino N., T. Kimoto, and I. Iida, Mobile Information Service Based on Multi-Agent Architecture, IEICE TRANS. COMMUN., VOL. E80-B, NO. 10 OCTOBER 1997, ftp://ftp.cs.arizona.edu/japan/table.contents/ieice.TOC/comm/English/E80.B.10

[28] General Magic Odyssey 1.0 beta 2 system requirements. 1997, http://www.generalmagic.com/technology/odyssey.html

[29] Han R., P. Bhagwat, R. Lamaire, T. Mummert, V. Perret, and J. Rubas, Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing IEEE Personal Communications, pp 8 – 17, December 1998

[30] Hendrickson B., and K. Devine, Dynamic Load Balancing in Computational Mechanics, http://www.cs.sandia.gov/~kddevin/LB/tiling.html

[31] Hokimoto A., and T. Nakajima, Robust Host Mobility Supports for Adaptive Mobile Applications, http://mmmc.jaist.ac.jp:8000/

[32] Hokimoto A., K. Kurihara, and T. Nakajima, An Approach for Constructing Mobile Applications using Service Proxies, http://mmmc.jaist.ac.jp:8000/

[33] Imielinski T., and B. R. Badrinath, Mobile Wireless Computing: Challenges in Data Management. Communications of the ACM, Vol. 37, No. 10, pages 18-28, Oct. 1994

[34] Imielinski T., and S. Viswanathan, Adaptive Wireless Information Systems, In the Proceedings of SIGDBS Conference, Tokyo – Japan, October 1994, http://www.cs.rutgers.edu/dataman

[35] Joseph A. D., A. F. deIespinasse, J. A. Tauber, D. K. Gifford, and F. Knashoek, Rover: A Toolkit for Mobile Information Access, In Proc. Fifth ACM Symposium on Operating System Principles, pages 156 – 171, ACM, December 1995

[36] Kahle R. S., Java Mobile Agents. http://www.parallax.co.uk/cetus/oo_mobile_agents.html

[37] Katz R. H., Adaptation and Mobility in Wireless Information systems, IEEE Personal Communications Vol. 1 No. 1. Pages 6-17. 1994

[38] Kidston D., J. P. Black, T. Kunz, M. E. Nidd, M. Lioy, B. Elphick, and M. Ostrowski, Comma, A Communication Manager for Mobile Applications, In Proceedings of Wireless 98, the 10th International Conference on Wireless Communications, Calgary, Alberta, Canada, pages 103-116, July 1998.

[39] Knabe F., An Overview of Mobile Agent Programming Proceedings of the Fifth LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, number 1192 in Lecture Notes in Computer Science, Stockholm, Sweden, June, 1996. Springer-Verlag, http://www.cs.virginia.edu/~knabe/papers.html#lomaps96

[40] Koga Y., C. S. Hong, and Y. Matsushita, An Internetworking Architecture between TINA-Like Model and Internet for Mobility Services IEICE TRANS. COMMUN., VOL. E80-B, NO. 10 OCTOBER 1997, ftp://ftp.cs.arizona.edu/japan/table.contents/ieice.TOC/comm/English/E80.B.10

[41] Kunz T. and J. P. Black, "Adaptive mobile applications based on mobile code", presentation at the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99), New Orleans, Louisiana, USA, February 1999, http://www.research.att. com/conf/wmcsa99/program.html.

[42] Kunz T. and J. P. Black, "An architecture for adaptive mobile applications", to appear in the Proceedings of Wireless 99, the 11th International Conference on Wireless Communications, Calgary, Alberta, Canada, pp. 103-106, July 1999.

[43] Kunz T., "The influence of different workload descriptions on a heuristic load balancing scheme", IEEE Transactions on Software Engineering, Vol. 17, No. 7, pp. 725-730, July 1991. Reprinted in Scheduling and Load Balancing in Parallel and Distributed Systems, pp. 497 - 502, IEEE Computer Society Press 1995, ISBN 0-8186-65874.

[44] Le M. T., S. Seshan, F. Burghardt, et al., Software Architecture of the InfoPad System, Proceedings of the Mobidata Workshop on Mobile and Wireless Information Systems, Rutgers, NY, USA, 1994, http://infopad.eecs.berkeley.edu/infopad-ftp/papers/1994

[45] Le Gall D., MPEG: A Video Compression Standard for Multimedia Applications, Communications of the ACM, 34(4): 46 – 58, April 1991

[46] Lingnau A., O. Drobnik, and P. Domel, An HTTP-based Infrastructure for Mobile Agents, http://www.tm.informatik.uni-frankfurt.de/agents/www4-paper.html

[47] Lo G. H., and T. Kunz, Access Study of Dynamic Applications Partitioning in Mobile Computing – An E-mail Browser, In Workshop on Object Replication and

Mobile Computing (ORMC'96) of OOPSLA'96 at San Jose, California October, 1996, URL: http://www.diku.dk/distlab/workshops/ormc96/

[48] Marsh B., F. Douglis, and R. Caceres, Systems Issues in Mobile Computing, Matsushita Information Technology Laboratory, Technical Report MITL-TR-50-93, February 1993, http://www.yeditepe.edu.tr/MaROS/biblio.html

[49] Mitsubishi Electric ITA. Concordia developer's guide. 1998, http://www.meita.com/concordia/

[50] Mobile-IP working group homepage, http://www.ietf.cnri.reston.va.us/html.charters/ mobileip-charter.html

[51] MPEG Overview – Recent advances in video compression, http://www.stud.ee.ethz.ch/~rggrandi/intro.html

[52] Nakajima T., and A. Hokimoto, Adaptive Continuous Media Applications in Mobile Computing Environments http://mmmc.jaist.ac.jp:8000/

[53] Nidd M., T. Kunz, and J. Black, Wireless Application and API Design, Proceedings of the Fourth International IFIP Workshop on Quality of Service, Paris, France, pp. 83-85, March 1996. Appeared as GMD-Studien Nr. 282, Quality of Service - Description Modelling and Management.

[54] Noble B. D., M. Price, and M. Satyanarayanan, A Programming Interface for Application-Aware Adaptation in Mobile Computing, The USENIX Association, Computing Systems, Vol. 8 No. 4 Fall 1995, http://www.yeditepe.edu.tr/MaROS/biblio.html

[55] Object Management Group (OMG) Common Object Request Broker Architecture (CORBA), http://www.omg.com

[56] ObjectSpace Voyager and Agent Platforms Comparison, http://www.objectspace. com/Voayger/

[57] ObjectSpace. Voyager2.0.0 User Guide, http://www.objectspace.com/Voyager/

[58] Othman M., and S. Hailes, Power Conservation Strategy for Mobile Computers Using Load Sharing, Mobile Computing and Communications Review, Volume 2, Number 1, January 1998, http://www.cs.ucl.ac.uk/research/mobile/papers.html

[59] Remote Objects in Java homepage, http://splash.javasoft.com/pages/intro.html/

[60] Russ S. H., I. Banicescu, S. Ghafoor, B. Janapareddi, J. Robinson, and R. Lu Hectiling, An Integration of Fine and Coarse-Grained Load-Balancing Strategies In Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing (HPDC'98), pp. 106-113, Chicago, July 1998

[61] Scheuermann P., J. Shim, and R. Vingralek, A case for delay-conscious caching of Web documents, Computer Networks and ISDN Systems 29 (1997) pp. 997-1005

[62] Solomon J. D., "Mobile IP: The Internet Unplugged," Prentice-Hall:Upper Saddle River, NJ, 1998, 384 pp., ISBN 0-13-856246-6

[63] Tokyo Research Laboratory IBM, The APIs of the Aglets Framework, 1998, http://www.ibm.co.jp/aglets/api/packages.html

[64] Versteeg S., Comparison of Mobile Agent Toolkits for Java. June 19 1998, http://www.cs.mu.oz.au/~scv/macomp.ps.gz

[65] Watson T., Application Design for Wireless Computing. In Proc. Workshop on Mobile Computing Systems and Applications, pages 91–94. ACM/IEEE, December 1994

[66] Welling G., and B. R. Badrinath, An Architecture for Exporting Environment Awareness to Mobile Computing Applications, IEEE Transactions on Software Engineering, Vol. 24, No. 5, May 1998, http://dlib.computer.org/ts/books/ts1998/pdf/e0391.pdf

[67] Xu C., B. Monien, and R Lüling, An Analytical Comparison of Nearest Neighbor Algorithms for Load Balancing in Parallel Computers, Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95, pp. 472-479, IEEE Computer Society Press, April 1995.

[68] Yavatkar R., and N. Bhagawat, Improving End-to-End Performance of TCP over Mobile Internetworks. In Proc. Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, U.S., December 1994, http://snapple.cs.washington.edu:600/mobile/mcsa94.html

[69] Zaki M. J., W. Li, and S. Parthasarathy, Customized Dynamic Load Balancing for a Network of Workstations, 5th IEEE International Symposium on High-Performance Distributed Computing (HPDC), Syracuse, New York, August 1996, http://www.cs.rpi.edu/~zaki/papers.html#HPDC96

[70] Zenel B., A Proxy Based Filtering Mechanism for the Mobile Environment, June 1997 http://www.mcl.cs.columbia.edu/~baz

[71] Zhao X., C. Castelluccia, and M. Baker, Flexible Network Support for Mobility Proceedings of ACM/IEEE MobiCom'98, http://gunpowder.stanford.edu/publications/flexible.ps

Appendix

A Comparison of Technologies for Developing Mobile Applications

A 1 CORBA and RMI

The Common Object Request Broker Architecture (CORBA) was created by the Object Management Group (OMG) [55] to provide a heterogeneous distributed environment for program development. The OMG defines the higher-level architecture, which allows the applications that use the objects to communicate.

Based on object technology, CORBA focuses on providing software components that may be transparently used across a network by any application on any platform. This provides the framework for total application integration: the ability for any application to access and be able to process any information provided by the available objects and to communicate with other applications through these objects.

In the CORBA architecture, both client and object implementation are isolated from the Object Request Broker (ORB) by an Interface Definition language (IDL) interface. Every object is expressed in OMG IDL. Clients see only the IDL interface, hiding the implementation details completely. Requests do not pass from client to object, but are managed by an ORB. This allows the ORB to manage the distribution details completely, reducing the complexity of the client and object implementation code.

Java Remote Method Invocation (RMI) [59] provides a mechanism for calling methods of remote objects, that is objects that exist on a virtual machine different from the one making the method call. RMI is a package that was included in the 1.1 release of the Java Development Kit.

For an object to be remotely accessible, an interface must be written for the object that extends the Java RMI interface Remote and includes the prototypes of each method that can be called remotely. The remotely accessible object must be declared to implement this interface in the class declaration. An instance of this object needs to be created and registered with the RMI registry. The RMI registry is a program that is used to bind objects to an address. To use the remote object, the client program must first obtain an instance of it. This can be done by calling the RMI registry on the host where the remote object is registered, whereby the name to which the remote object is bound is passed in, and a reference with the type of the interface to the remote object is returned. The client program can now call any methods of the object that are declared in the interface.

Like CORBA, RMI does not support either object mobility or mobile agents. Once created, an object remains on the same machine for its lifetime. RMI supports synchronous messages, but does not support multicast messaging.

A 2 IBM Aglets

Aglets is a Java library written by IBM Tokyo Research Laboratories [63]. The version of Aglets looked at is the Aglet Software Development Kit 1.0. The Aglets API has five key classes: Aglet, AgletProxy, AgletContext, Message, and Event.

Aglets are executed in a similar way to runnable applets. Each aglet must implement the run() method. The run() method is invoked when an aglet starts execution. Its contents hence define how an aglet behaves. Aglets cannot access each other's methods or data directly, however each aglet has a proxy object that implements the AgletProxy interface. A proxy object is created automatically by the system for each new aglet that is created. Communication from one aglet to another goes through the proxy. The purpose of this interface is to provide a mechanism to control and limit direct access to aglets.

Aglets provides an object of type AgletContext to provide an interface to the Aglets server. There is one AgletContext object for each server. Each aglet is associated with an AgletContext object.

Aglet messages are encapsulated in a Message object. Each message contains a string that describes the kind of message and the message contents. The contents may be either a basic type, an object or a hashtable of values (using the java.util.HashTable). Messages in Aglets can be delivered from one aglet to another in the following five different ways:

- SendMessage(): Sends a message in a synchronous way.
- SendAsyncMessage(): Sends a message in an asynchronous way.
- SendOnewayMessage(): Sends a one-way message to the aglet. The message is sent asynchronously without the receiver sending a reply.

- SendFutureMessage(): Sends a message to the aglet. The sender aglet does not block but is returned a FutureReply that the sender can poll to see if the reply has come in.
- MulticastMessage(): An aglet can also multicast a message to all aglets running on the same server by invoking the multicastMessage() method that is part of the AgletContext interface.

Each aglet may implement a onMessage() method to handle incoming messages from other aglets. The onMessage() method usually contains a list of cases to handle different kinds of messages. If an aglet does not include a handler for a particular kind of message, that message is ignored and has no effect on the aglet.

Aglets allows an agent to move itself to a well-known location (URL). The dispatch() method is give the URL of the destination server as its argument. Other aglets or threads may request an aglet to migrate by calling the dispatch() method of the aglet's proxy. Aglets may listen to mobility events by implementing the MobilityListener interface. However, Aglets does not allow an agent to move to an object, even if the object is stationary. In addition, the platform does not support mobility of simple Java objects.

An aglet may be cloned. The AgletProxy class contains a method clone() which causes an aglet to be cloned. The cloned aglet is created on the same server as the original. It is of the same class and contains the same data values as the original aglet. The cloned aglet will begin execution. An aglet may listen to cloning events by implementing the CloneListener interface.

A 3 General Magic's Odyssey

Odyssey is the Java Agent toolkit written by General Magic [28]. The Odyssey API is small and simple. It offers the essential features needed to write mobile agent applications. The key classes in the Odyssey library are:

- OdysseyProcess: This is the base class of Odyssey enabled processes. Both the classes Place and Agent are derived from OdysseyProcess. The OdysseyProcess class contains a method called live() that subclasses override to define how the process behaves. The live() method begins execution after objects of this class have been created and initialized. The class also contains methods for controlling the execution of the process and identifying the process.
- Agent: An agent is a subclass of OdysseyProcess that can migrate from one place to another. An agent must always be at a place.
- Place: Places host agents.
- Worker: Odyssey provides the Worker class derived from Agent, to support a specialized kind of agent. A worker has a task list that will be systematically executed. A task is represented as a tuple of the name of the method to be executed to do the task and the location of the place where the task is to be done.

In Odyssey, an agent may migrate two ways. The Agent class contains the go() method, that when invoked will cause the agent to migrate to another place at the location given.

Communication between agents is done through meetings. Agents can only meet when they are at the same location. If an agent wants to meet with another, it requests the place to arrange a meeting by calling the meet() method. Each agent implements a method called meeting() that defines how the agent responds to requests for meetings.

Meeting() is passed the class name, the process identifier and the label of the agent requesting the meeting.

Odyssey agents can be transported from one agent system to another by any reliable byte stream. By default, Odyssey uses Java RMI (remote method invocation) to transport Odyssey agents. Odyssey supports two alternative transport mechanisms, DCOM and IIOP. DCOM (Distributed Component Object Model) is a Microsoft technology that is packaged together with Windows NT. It is not available on Unix. IIOP (Internet Inter-Orb Protocol) is a part of CORBA that is available from multiple vendors.

The Odyssey system has a clean design and has very strong semantics. It is sufficiently powerful to write mobile agent applications, but it is not a good choice for the project, which implements object mobility.

A 4 Mitsubishi Concordia

Concordia is produced by Mitsubishi Electric ITA [49]. Mobility is implemented in Concordia using itineraries. Each agent has an itinerary associated with it. An itinerary is a list of destinations where the agent is to travel. Each destination has a method name associated with it. The method named is eexecuted when the agent arrives at that destination. The Agent class has a method launch() that when invoked, causes the agent to execute its itinerary. The method prepareForTransport() is called each time the agent is about to migrate to another location, and finishedTransport() is called on arrival at the new host.

Concordia uses events as the method of communication between processes. The programmer subclasses EventType to define different types of events. The Concordia server runs an event manager. A process may register to listen to certain types of events.

A process may also register that it will post certain types of events. Each agent implements the handleEvent() method to define how events are handled.

Concordia includes a collaboration framework that enables multiple agents to work together and coordinate their actions. Agents within an application may form one or more collaboration units, known as agent groups. Concordia provides the abstract base class CollaboratorAgent for writing collaborator agents and AgentGroupImpl for defining groups. Agent groups are implemented as distributed objects which export a simple interface to collaborator agents, which, in turn, hold remote references to agent groups.

Concordia provides some impressive administration tools with the distribution. The program admin provides a graphical interface to the administration of Concordia. The program has three uses: managing the server services, launching agents and managing security. The program allows new agents to be created, their itineraries to be defined, and for them to be launched. Some other tools provided are an agent launch wizard, a command-line agent launching program and a graphical persistence browser for viewing and editing the contents of the persistence store.

Overall Concordia is a good and well designed system. It is easy to use, provides helpful tools and is well documented. Perhaps its greatest weakness is its lack of support for different modes of delivering messages. Like Aglets and Odyssey, Concordia does not support object mobility.

A 5 Summary

OMG's CORBA technology supports fundamental distributed computing. It allows developers to create remote objects and send them messages as if they were local. Some advanced features such as distributed garbage collection, different messaging modes, and a naming service are included. However, it does not support object mobility or mobile, autonomous agents. This critical weakness is not acceptable for developing mobile applications.

JavaSoft's RMI is a powerful technology for developing distributed systems. It supports network class loading through HTTP, and has a distributed garbage collector. However, with RMI it is difficult to achieve true location transparency for two reasons. First, all method signatures in a remote object must declare that they may throw a RemoteException. Second, remote RMI objects are always passed by reference, not by value. Because of these two restrictions, it is difficult, but not impossible, to build a system where objects can migrate between clients, servers, and other machines. For many systems this lack of mobility and location transparency is not a problem, but for others it can be a major headache.

The way mobility is handled is fairly similar for all the platforms. Each system has a method inside the agent class to cause the agent to migrate. Upon migration, a special method is called that the programmer can override to do any specific actions before leaving. There is a similar method to override that is called upon arrival. Aglets is more limited than the other libraries with regard to forcing the run method to be executed at each new location. The other libraries give an option to specify which method is to be executed at the destination. Odyssey and Concordia offer direct support for setting an agent's itinerary. Voyager does not explicitly have itineraries attached to their agents, but they are trivial to implement. Voyager has more general migratory support as it allows any object to migrate, not just agents.

One of the areas where the systems differ most is in the area of messaging. Aglets messages are sent as message objects, and handled by the handleMessage() method. Aglets cannot call each other's methods directly. In Voyager, messages are sent either by method call or by constructing a message object and calling the method invoke() to send it. Odyssey has the concept of agents having a meeting, whereby two agents need to agree to have a meeting and can then negotiate. Concordia uses events to send messages. Although the same effect can be achieved, Voyager's messaging function is easier to implement and use. There is a difference between libraries with regard to modes of how messages can be delivered. Aglets and Voyager support three fundamental different ways of delivering messages: synchronously, asychronously and future-reply. All libraries except Odyssey support multicasting. Aglets and Odyssey only allow agents to communicate. In Concordia and Voyager, any object can send and receive messages. Voyager messages can be sent to objects at a different server. This is not possible with the other platforms. The others only support communication between objects currently situated at the same location.

Aglets and Concordia both come with useful tools for managing mobile agents. Both have graphical interfaces to the agent servers with server management functionality. From the server interfaces it is possible to create agents, control their execution and destroy agents. Voyager and Odyssey do not come with a graphical interface to the server. Fortunately, in our experimental case, this feature is not that important.

All of the mobile agent systems examined provide viable frameworks for implementing a mobile agent system. All the libraries give support for agent migration and agent messaging. Voyager has greater capabilities than the other libraries. Voyager effectively provides an alternative implementation to Java RMI, with the additional capability of being able to move objects as well as remotely invoking methods. The other mobile agent libraries only allow agents to migrate. Only Voyager and Concordia enable persistence. Aglets and Voyager provide the most number of modes for transmitting messages. A side benefit of Voyager's focus on interface based programming is easier integration with other distributed systems such as RMI, CORBA, and DCOM. Voyager is the system of choice if its capability of object mobility (not only mobile agent) is required.

B Run Queue Length Estimation [3]

Most Unix/Linux systems only provide very inert load average values, the fastest of which is averaged over the last 60 seconds. As this index changes too slow to be of good use for load distribution decisions, WINNER transforms these values into an index more sensitive to load changes. To understand how this is done, one has to know the way UNIX kernels calculate their load average every I seconds:

Where a_n is the last load average I seconds age, a is the current length of the CPU run queue, a_{n+i} is the current load average to be calculated and β is the smoothing factor determining by which the old average a_n contributes to a_{n+i}. Because this smoothing is performed exponentially, a value of $\beta = e^{-i/60}$ is used for the 60 second average.

$$a_{n+i} = \beta a_n + (1 - \beta)a$$

Neglecting variations of the run queue length a and assuming an interval of 10 seconds between two consecutive load measurements performed by WINNER's node manager, we get an approximation for the average run queue length a out of two consecutive load values:

$$a_{n+10} = \beta^{10/i} a_n + (1 - \beta^{10/i})a$$

$$a_{n+10} = e^{-10/60} a_n + (1 - e^{-10/60})a$$

$$a = \frac{a_{n+10} - e^{-10/60} a_n}{1 - e^{-10/60}}$$

Using the above equation, WINNER calculates the average run queue length a during the last period. This value reacts much faster to load changes than a_n itself as it is reported by the operating system.