A MOBILE CODE TOOLKIT FOR ADAPTIVE MOBILE APPLICATIONS

By

Salim H. Omar

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

Master of Computer Science

Ottawa-Carleton Institute of Computer Science
(OCICS)

Carleton University
Ottawa, Ontario
April 2000

Abstract

The rapidly expanding technology of cellular communication and wireless communication, portable computers, and satellite services promises to make it possible for mobile users to have access to information anywhere and anytime. Users on a daily basis are using portable devices frequently. These types of devices can be classified primarily by their size, computational power, memory capacity, and power and battery lifetime. For example, Personal Digital Assistant devices (PDAs) are small portable computers run on AA batteries. They may be without disk and have more constrains in terms of memory capacity and computational power than other portable devices, which are called laptops, that have more computation power, memory, more storage capacity; however; their battery lifetime is shorter if we consider typical use of these devices.

Finding approaches to reduce power consumption and to improve application performance is a vital and interesting problem to be investigated. Many approaches have been developed to address this problem. They range from hardware to software level approaches. Our work is at the application layer too, where an approach for adaptive mobile applications is developed. In this thesis, we propose a mobile code toolkit for adaptive mobile applications that runs on WindowsCE platform. With this toolkit we combine JVMs on both the proxy server and the mobile device as one virtual machine from the application point of view to dynamically split application objects between JVMs according to the mobile environment.

Acknowledgments

I take this opportunity to express my sincere thanks to Dr. Thomas Kunz for being an excellent supervisor and mentor. All the meetings with him on this work were very effective and focused, without which this thesis could not have been completed.

I would like as well to thank the Bell Mobility, the NSERC, and the Faculty of the Graduate Studies at Carleton University for financially supporting this research.

I express my gratitude to my parents for their patience and support to achieve my goals in life. I thank several of my friends and well wishers, whose presence made my life at school very pleasant and interesting.

Last, but certainly not the least, I wish to dedicate this thesis as a tribute to the memories of my grandfather who always had a very positive and rational approach towards life and left unforgettable memories to the rest of the family.

Salim H Omar.
Ottawa, Canada.

In the Name of Allâh, the Most Beneficent, the Most Merciful.

(77)"Does not man see that We have created him from Nutfah (mixed male and female discharge semen drops). Yet behold! He (stands forth) as an open opponent.(78) And he puts forth for Us a parable, and forgets his own creation. He says: "Who will give life to these bones when they have rotted away and became dust?"(79) Say: (O Muhammad AW) "He will give life to them Who created them for the first time! And He is the All-Knower of every creation!" (80) He, Who produces for you fire out of the green tree, when behold! You kindle therewith. (81) Is not He, Who created the heavens and the earth Able to create the like of them? Yes, indeed! He is the All-Knowing Supreme Creator. (82) Verily, His Command, when He intends a thing, is only that He says to it, "Be!" and it is! (83) So Glorified is He and Exalted above all that they associate with Him, and in Whose Hands is the dominion of all things, and to Him you shall be returned."

(Quran) Chapter 36, verses 77-83

List of Tables

List of Figures

1. Introduction

The rapidly expanding technology of cellular communication and wireless communication, portable computers, and satellite services promises to make it possible for mobile users to have access to information anywhere and anytime. Users on a daily basis are using portable devices frequently. These types of devices can be classified primarily by their size, computational power, memory capacity, and power and battery lifetime. For example, Personal Digital Assistant devices (PDAs) are small portable computers run on AA batteries. They may be without disk and have more constrains in terms of memory capacity and computational power than other portable devices, which are called laptops, that have more computation power, memory, more storage capacity; however; their battery lifetime is shorter if we consider typical use of these devices.

Regardless of the classification of portable devices, some portable devices are supported, and some will be, in the near future, supported with wireless connection to information networks such as the Internet and Intranets.

The resulting computing environment is called mobile computing. Users of this environment are no longer required to maintain a fixed position in the network, and there is no restriction on their mobility.

Already, there are a number of general-purpose (Horizontal) and domain specific (Vertical) applications of mobile wireless computing in use. Taxi dispatcher, mail tracking, point of sale are examples of Vertical applications, and Mail-enabled applications and information services are examples of Horizontal applications.

Users who are carrying personal portable devices will be able to send and receive emails from any location as well as be informed about specific predefined conditions irrespective of time and location. Mobile computing will result in a new approach of computing.

Due to battery restrictions, the mobile device will be quite often powered off. Short periods of activity are most likely to happen. Reading or sending email, querying remote databases, for example, will be quite often discontinued or separated by considerable amount of time where the device is disconnected.

Since there is no restriction on the user mobility, the mobile device quite often will be used in different environments over short periods of time, where the user can cross between two different coverage areas of wireless communications. This process is called *hand-offs*.

Hand-offs are relatively straightforward in cellular voice communications due to the higher lose of information that can be tolerated; however, in data transfers, where the rate of data lose must be as low as possible, the hand-offs form a challenge.

1.1 Mobile Computing Challenges

Mobile computing poses new challenges. The major challenges can be categorized as following.

1. **Mobility Management and Scalability.**
2. **Wireless Communication.**
3. **Portability.**

1.1.1 Mobility Management and Scalability.

On the fixed network, mobile users are able to establish a connection from different data ports at different locations. Wireless connection enables virtual unrestricted mobility and connectivity from any location within the area of radio coverage. Mobility is a new important component in system design. It affects to a certain extent the network level data management as well as the application level.

Mobility of clients results in constantly changing topology of the system, calling for mobility of resources. Location management deals with mobile clients while configuration management refers to mobility of resources. In a mobile environment, the

location of the user can be considered as a variable whose value changes with every move from one location to another. Hence, location becomes a frequently changing piece of information.

The fundamental trade-off in location management is between searching and updating. For example, a user **A** wants to establish the location of a user **B**. Should user **A** search the entire network or should user **A** search only predefined location? Or, should the user **B** inform ever body about its move from one location to another?

Some work has been done on comparing different locating and addressing schemes. The problem itself is difficult since it involves several dimensions. Solutions that are optimal in terms of number of messages sent may show a poor performance in terms of latency. It is not clear how detailed the statistical profiles of users should be in order to provide a significant performance advantage. In general, mobility of hosts result in a new set of issues in distributed systems [1].

The less informed the sender is, the more search cost is incurred. Hence mobility substantially affects data placement. Since mobile hosts have severe resource constraints in terms of limited battery life and limited size of non-volatile storage, the burden of computation and communication load can not be distributed equally among static and mobile hosts.

The scale of the mobile environment goes far beyond any existing paradigms. Many predictions call for tens of millions of portable devices of varying classes that can move across a worldwide communication networks.

In location management, the total volume of transactions due to location updates may be higher by an order of magnitude than the capacity of the existing networks [2].

Due to frequent changes that may involve wide-area moves of large number of machines, scale plays a critical role. Scale can have major consequences for limited bandwidth resources. The increasing number of users requires using smaller and smaller cells because of the limited frequency systems. This in turn will complicate the location management due to increasing number of hand-offs.

1.1.2 Wireless Communication.

Mobile computers require wireless network access, although sometimes they may physically attach to the network for a better or a cheaper connection when they remain stationary. Wireless communications is much more difficult to achieve than wired one because the surrounding environment interacts with the signal, blocking signal paths and introducing noise and echoes [3]. Wireless connections are of lower quality than wired ones because of these reasons.

Lower bandwidths, higher error rates, and more frequent spurious disconnection are factors that make the wireless communications of less quality. These factors can in turn increase communication latency due to retransmission, retransmission timeout delays, error control protocol processing, and short disconnection.

Wireless connection can be lost or degraded also by mobility. Users may step out of the coverage of network transceivers or enter areas of high interference. Unlike the

typical wired networks, the number of devices in a cell varies dynamically, and a large concentration of mobile users may overload network capacity as well.

Today's computer systems often depend on the network. They may stop to work during network failures. Network failure is of greater concern for mobile computing designs than the traditional one since wireless communication is very susceptible to disconnections.

Either spending more resources on the network trying to prevent disconnections, or allocating more resources to enable systems to cope with disconnections more gracefully and work around them as much as possible is a primary solution.

The more autonomous the mobile computer is, the better it can tolerate network disconnections. Some applications, for example, reduce communication by running entirely locally on the mobile computer rather than splitting the application and the user interface across the network.

Mobile computing designs need to be more concerned about the bandwidth consumption and constraints than the fixed computers. Wireless communications deliver lower bandwidth than wired networks. Tables 1.1 and 1.2 show the bandwidth for various wireless and wired media.

Table 1.1: The delivered bandwidth of different wireless media

Type	Bandwidth
Infrared	1 Mbps
Radio communication	2 Mbps-10Mbps
Telephony (CDPD Cellular Digital Packet Data)	4 to 19.2 Kbps

Table 1.2: The delivered bandwidth of different wired media

Type	Bandwidth
Ethernet.	10-100 Mbps
FDDI	100 Mbps
ATM	155 Mbps

Network bandwidth is divided among the users sharing a cell. The deliverable bandwidth per user is a more useful measure of network capacity than raw measured bandwidth. Since this measure depends on the size of population, it is suggested that the bandwidth of wireless communications networks is measured by the bandwidth per cubic meter [4].

Mobile computing designs also strive to cope with a much greater variation in network bandwidth than the tradition one. Bandwidth can shift one to six orders of

magnitude between being plugged in versus using wireless access. Fluctuating traffic load seldom causes this much variation in available bandwidth in fixed networks.

An application can approach this change in bandwidth in one of the following ways.

1. It assumes high bandwidth connections and operates only while plugged in.
2. It assumes low bandwidth connections and does not take advantage of existing higher bandwidth.
3. It adapts to currently available resources, providing the user with a variable level of quality of service.

In contrast to most stationary computers, which stay connected to a single network, mobile computers encounter more heterogeneous network connections. As they may leave one network, they switch to another. Even in different places, they may experience different qualities. For example, a meeting room may have better wireless connection than the hallway of a section in a building. Even when plugged in, they may still concurrently connect through wireless connections. They may need to switch from one network interface to another especially when moving from indoors to outdoors, for instance, when switching from cellular coverage in the city to satellite coverage in the country. This heterogeneity makes mobile networking more complex than traditional networking

1.1.3 Portability.

Desktop computer are not expected to be carried around with their users. Their design allows them to reduce space, power, and cabling and heat dissipation. The design of mobile computers should strive for properties such as size, weight, durability and long battery life. Any specialized hardware to offload from the CPU tasks such as data compression or encryption should justify its consumption of power and in size and weight.

Batteries are the largest single source of weight in a portable computer. While reducing battery weight is important, too small a battery can undermine portability, which may lead the user to charge more frequently. Minimizing power consumption can improve portability by reducing battery weight and prolonging the life of a battery.

Power consumption is given by CV^2F, where C is the capacitance of the wires, V is the voltage swing, and the F is the clock frequency, there are three ways to reduce the power consumption. First, by reducing capacitance of wires by greater VLSI integration and multi-chip module technology. Second, by reducing the voltage levels by redesigning chips to operate at lower voltages. Chips operate at five volts, but to save power, some manufactures develop chips that work at 2.5 to 3 volts. Third, by reducing the frequency. Clock frequency can be reduced, trading off computation speed against the power savings. PDA products have adopted this idea as well as other notebooks. For example, the Sharp PC 6785 can save power by dynamically shifting CPU speed from 25 MHz to 10 MHz when it detects a shortage of power.

Power can be saved not only by design, but also by efficient use of operations. Power management software can power down some individual hardware components when they are in idle mode, for example, spinning down the internal disk or turning off screen lighting. Applications can conserve power by reducing computation, communication and memory. Since cellular telephone transmission typically requires about ten times as much power as reception, trading for more receiving can also save power [5].

Table 1.3: Power consumption break down by system component.

System Component.	Power Consumption Percentage Wise.
Display edge-light	35%
CPU/Memory	31%
Hard Disk	10%
Floppy	8%
Display	5%
Keyboard	1%

In conclusion, mobile computing is characterized by the previous constraints and challenges. These constraints are not as product of current technology, but they are related naturally to mobility. Together, they complicate the design of mobile information systems and require rethinking traditional approaches to information access and application design. Mobility intensifies the tension between autonomy and interdependency that is characteristic of all distributed systems.

The relative resource shortage of mobile elements as well as their lower trust and robustness argue for reliance on static servers. The need to cope with unreliable and low-performance networks, as well as the need to be sensitive to power consumption argues for self-reliance. Any feasible approach to mobile computing must strike a balance between these competing issues. This balance cannot be static as the environment of mobile computing changes; it must react and dynamically reassign the responsibility of client and server. In other words, the clients must be adaptive.

1.1.4 Thesis Argument.

Finding approaches to reduce power consumption and to improve application performance is a vital and interesting problem to be investigated. Many approaches have been developed to address this problem. They range from hardware to software level approaches as mentioned previously.

Previous work at the software level for mobile applications was to split statically, at design time, an application into a server and client, where the client executes at the mobile device and the server runs at a fixed host in the wired network. Splitting an application statically does not grantee the maximum quality of service to the users, especially in mobile computing environments due to the above-mentioned challenges. To improve quality of service to the users, at the fixed host, filtering mechanisms that work according to the current condition of mobile computing environment are deployed, which make mobile applications more adaptive.

Our work is at the application layer too, where an approach for adaptive mobile applications is developed. In this thesis, we suggested a new approach based on Greedy Graph partition for adaptive mobile applications, in which an application's objects will be split dynamically between the mobile device and fixed host according to the mobile device and fixed host's available resources and wireless network state.

This approach requires special infrastructure and tools rather than a specific application design. Mobile applications, especially ones that do intensive computation and communication, can be divided dynamically as a client and server between the wired network and the mobile device according to the mobile environment and to the availability of the resources on both the mobile device and the wired network. With this approach, more windows of adaptability to the mobile environment are possible. In addition, it allows the applications to have dynamic access to faster machines through faster servers. This will increase the performance of applications and reduce the power consumption on mobile devices since offloading computation to the wired network will reduce the CPU cycles and memory needed to achieve certain tasks at mobile devices.

In this thesis, Java will be used as primary developing language for applications as well as for implementing our toolkit. Since Java produces a portable executable code, it allows implementation of distributed computing easily. However, Java Virtual Machines are in early stages of development, particularly those work for the WindowsCE platform. They need to be extended to export the mobile computing environment

variables, such as available bandwidth, battery lifetime and power available at the mobile host as well as performance parameters such as CPU utilization. These extensions require the use of native languages, such as C/C++.

1.2 Contribution of Thesis

The contribution of thesis can be summarized as following:

- Development of an object mobility toolkit that dynamically and transparently moves objects from mobile devices to the proxy server. The toolkit is based on Java Serialization and Proxy Patterns, and works on a wide variety of devices including PDAs running WindowsCE.
- Suggesting a modified Greedy Graph Partitioning algorithm to be used for Load Sharing purposes to share load between the mobile devices and the proxy server.
- Implemented an MP3 Player in Java
- Demonstrates feasibility of dynamic load balancing approach to overcome client device and the bandwidth limitations.
- Published early insights in Parallel Distributed Processing Techniques and Applications (PDPTA'99) Conference, and in the Proceedings of the Eighteenth (IASTED) International Conference on Applied Informatics (AI'2000)

1.3 Thesis Outline

This thesis contains 8 chapters including this chapter. Chapter 2, **Background**, is mainly about languages, tools and infrastructures for building, implementing, and supporting distributed computing applications as well as mobile applications in general. These tools will be briefly discussed and related to our work. As an example of these tools, we will discuss Java RMI, CORBA, Jini and Voyager. Chapter 3, **Related Work**, surveys the previous and current work in the field of mobile computing; particularly previous work that uses *Proxy Servers* as an infrastructure for supporting mobile applications. Chapter 4, **Basic Idea and Preliminary Studies**, explores our main idea. Results of a case study show the feasibility of our approach, dynamically splitting application functionality. These results were published in the Proceedings of PDPTA'99 conference. In Chapter 5, **Dynamic Object Mobility Toolkit**, we propose the basic structure of our toolkit, showing the main structure of necessary software components and describing their functionality. The focus of this chapter will be on the monitoring part and the process of moving objects between two JVMs. Chapter 6, **Java MP3 Player,** explores an MP3 decoder, written in Java. Technical information will also be provided about the decoding algorithm. The *object graph* and *calling graph* of the Java MP3 decoder will be presented as well. We show complexity and performance of the MP3 decoder on different platforms including a WindowsCE device with a MIPS CPU, a laptop with a Pentium 133, and WindowsNT workstations with 300MHz Pentiums. In Chapter 7, **Experiments and Results**

we introduce a modified version of the Greedy Graph Partitioning algorithm to group strongly related objects, and we show results of using our approach with the ***Java MP3 player*** as an example. We run the algorithm with different parameters for the mobile environment and observe the results. We expect improvements in decoder performance as well as reducing power consumption for both laptop and WindowsCE devices. In Chapter 8, **Conclusion and Future Work,** we draw our conclusion and show the feasibility of our approach, and we indicate possible extensions of this work as areas of future research.

2. Background

2.1 Java Language and Virtual Machine

Java is a language that is claimed to be simple, object-oriented, network oriented designed, interpreted, robust and secure, architecture uniform, portable, high-performance, multithreaded and dynamic [6]. We agree with most previous characteristics of Java, but with the high-performance. We argue that Java to be high-performance is completely depends on the platform as well as whether any type of optimization is being used or not. For example, a JVM, with JIT compiler enabled, runs on a Pentium III 500 MHz device much faster than a JVM, with no JIT at all, that runs on WindowsCE handheld device with MIPS 75MHz CPU.

Java was designed as close to C++ as possible in order to make the language more comprehensible. Automatic garbage collection was added, thereby simplifying the task of Java programming but making the system more complicated.

A common source of complexity in many C/C++ applications is storage management, allocation and freeing memory. Having automatic garbage collection, the

Java language not only makes the programming task easier, but also it dramatically cuts down on the number of bugs in applications. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The Java interpreter and standard libraries have a small footprint. A small size is important for use in embedded systems and so Java can be easily downloaded over the net.

2.1.1 Object-Oriented

Object-oriented design is very powerful because it facilitates the clean definition of interfaces and makes it possible to provide reusable software components [7]. Object oriented design is a technique that focuses design on the data and on the interfaces to it. Object-oriented design is also the mechanism for defining "plug and play" modules through interface definition mechanism.

2.1.2 Network-Oriented

Java has an extensive library of routines that work easily with TCP/IP protocols like HTTP and FTP. This makes creating network connections much easier than using C/C++ libraries. Java applications can open and access objects across the net through URLs with the same ease that programmers are used to when accessing a local file system.

2.1.3 Robust

Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of stress on early checking for possible problems, late dynamic runtime checking, and eliminating situations that are error prone [8].

One of the advantages of a strongly typed language, like C++, is that it allows extensive compile-time checking so bugs can be found early. C++ inherits a number of ambiguities in compile-time checking from C, which namely are method/procedure declarations.

The linker understands the type system and repeats many of the type checks, which are done by the compiler to guard against version and mismatch problems. The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed. In addition, it is not possible to turn an arbitrary integer into a pointer by casting [9].

Very dynamic languages like Lisp, TCL and Smalltalk are often used for prototyping. One of the reasons for their success at this is that they are very robust. Java programmers can be relatively unconcern about memory issues because they do not have to worry about it being corrupted. Because there are no pointers in Java, programs cannot accidentally overwrite the end of a memory buffer. Java programs also cannot gain unauthorized access to memory, which could happen in C/C++.

Java forces programmers to make choices explicitly, because it has static typing, which the compiler enforces. Along with these choices comes a lot of assistance, for example, programmers can write method invocations and if something wrong happened, such as calling an undefined method or calling a method with incompatible arguments, they are informed about it at compile time.

2.1.4 Security

Java is intended for use in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. The authentication techniques are based on public-key encryption. There is a strong relationship between "robust" and "secure." For example, the changes to the semantics of pointers make it impossible for applications to copy access to data structures or to access private data in objects that they do not have access to. This closes the door on most activities of malicious code in Java since it does not allow memory pointer notion [10].

2.1.5 Architecture Uniform

Java was designed to support applications on networks. Networks are composed of a variety of systems with a variety of CPU and operating system architectures. To enable a Java application to execute anywhere on the network, the compiler generates an architecture-uniform object file format; the compiled code is executable on many processors, given the presence of the Java runtime system.

This is useful not only for networks but also for single system software distribution. With Java, the same version of the application runs on all platforms. The Java compiler achieves platform-independency by generating byte code instructions, which have nothing to do with particular computer architectures. They are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly [11].

2.1.6 Portability

Unlike C and C++, there are no implementation-dependent aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them. For example, **int** always means a signed two's complement 32 bit integer, and **float** always means a 32-bit IEEE 754 floating point number. Making these choices is feasible because basically all relevant CPUs share these characteristics.

The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows NT/95, WindowsCE and the Macintosh OS.

2.1.7 Interpreted Bytecode

Java bytecode is translated on the fly to native machine instructions and not stored anywhere. As a part of the bytecode stream, more compile-time information is carried over and available at runtime reflecting the internal structure of compiled source code.

2.1.8 High Performance

While the performance of interpreted bytecode is more than adequate for small computations. there are situations where higher performance is required. The bytecode can be translated at runtime into machine code for the particular CPU the application is running on. The bytecode format was designed with generating machine codes in mind, so the actual process of generating machine code is generally simple. Using Just-In-Time compiling technology, Java virtual machines can be speed up to ratio of 7 to 10 times faster [12].

2.1.9 Multithreaded

Multithreading is a way of building applications with multiple threads. Writing programs that deal with many things happening at once can be much more difficult than writing in the conventional single-threaded C/C++ style.

Java has a sophisticated set of synchronization primitives that are based on the well-known used monitor and condition variable paradigm. By integrating these concepts into the language, they are much easier to use. Other benefits of multithreading are to improve interactive responsiveness and real-time behavior, which is limited by the underlying platform. Running on top of other systems like Unix, Windows, the Macintosh OS, WindowsNT or WindowsCE limits the real-time responsiveness to that of the underlying system.

2.1.10 Embedded Java Platforms

Currently, embedded devices span a wide variety of consumer and business products, including devices such as smart mobile phones, pagers, PDAs, set-top boxes, process controllers, office printers, and network routers and switches. Embedded devices have dedicated functionality; they are designed exactly for a specific set of tasks. Since they are engineered for long life and high reliability, embedded devices include low-speed microprocessors and may have a limited amount of memory [13].

To meet performance and size requirements, embedded device manufacturers use a Real Time Operating System (RTOS) and custom development tools, compatible with devices' memory limitations. There are several different RTOS vendors that exist, each with a specific operating environment and many with strongly integrated and specialized development tools.

Early environments for embedded devices were developed in assembler. As these devices developed, some manufacturers shifted to higher-level languages like C and C++. Embedded device manufacturers have turned to the Java programming language to answer their needs and for the advantages that Java provides as mentioned in Section 2.1 There are JMVs that are available currently for Handheld PCs and other portable devices. Major software companies, namely Sun Microsystems and Microsoft, produce them. We use the product of Microsoft to build the applications and tools for Handheld PCs [14]. Their JVM supports the core functionality of Sun's JDK 1.1x, which is an appropriate choice for our work.

2.2 Distributed Systems and Tools

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, many computer languages support sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in application level protocols to encode and decode messages for exchange, and the design of such protocols is not trivial and can be error prone.

An alternative to sockets is Remote Procedure Call (RPC), which abstracts the communication interface to the level of a procedure call. Instead of working directly with sockets, the programmer has the feel of calling a local procedure, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. RPC systems encode arguments and return values using an external data representation, such as XDR [15].

RPC, however, does not translate well into distributed object systems, where communication between program level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require remote method invocation (RMI.) In such systems, a local surrogate (stub) object manages the invocation on a remote object. The following sections will describe briefly some architectures and tools that support object-oriented distributed systems.

2.2.1 Common Object Request Broker Architecture (CORBA)

As more industries are connected to the Internet and intranets, software development is becoming complex. The complexity of software development has produced a major revolution in systems development. Object-oriented computing is progressively becoming more typical. The other major revolution, which is occurring in the computer industry, is distributed computing.

Every major new distributed computing technology has committed to the promise of interoperability between heterogeneous systems and applications. Although connectivity between most types of operating systems platforms is available, interoperability at the application level remains an issue. Main factors include the inherent difficulty of distributed application programming and the lack of standard interfaces between applications.

CORBA is the product of OMG [16], which defines a higher-level facility for distributed computing. It provides standards for distributed objects architectures.

2.2.1.1 Distributed Objects

A distributed object is a piece of code that can live anywhere in the network or Internet. It breaks the restrictions of classical objects. Distributed objects are packaged as independent pieces of code that can be accessed by clients in the same machine or machines across the network through local/remote method invocations. The language and compiler used to create the objects are totally transparent to the clients. Clients need not

know where the object resides or the operating system it executes on. Distributed objects can message each other transparently anywhere in the Internet. CORBA clients just need the interfaces the server object publishes. The interfaces serve as the binding contract between clients and servers.

2.2.1.2 Components

Components are stand-alone objects that can be plugged and played across networks, applications, languages, tools and operating systems. Distributed objects are by definition components because of the way they are packaged. The distributed objects infrastructure makes components more autonomous, self-managing and collaborative [17]. The main idea behind software component technology is to provide software users and developers the same levels of plug-and-play application interoperability that are available to consumers and manufacturers of electronic parts.

2.2.1.3 Object Request Broker (ORB)

An object request broker (ORB) is the central component of CORBA. It is the middleware that establishes client-server relationships between objects. Using an ORB the client can transparently invoke a method on a server object. The server object can be on the same machine or on a remote machine in the network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass the parameters, invoke the method and finally return the results computed to the server. The client need not know any details about the object like the location of the object, the

programming language in which it is implemented, the operating system under which it executes or the platform on which it exists. The client-server relationship is applicable for any particular application. Objects can act as either client or server depending on the instance; however, CORBA does not support the moving of objects from one host to another. The following are the features of CORBA ORB.

- **Static and dynamic method invocation**

  CORBA ORB supports both types of method invocations, static and dynamic. It is possible either to statically define method invocations at compile time or dynamically discover them at run time. Thus, it is possible to have either strong type checking at compile time or maximum flexibility associated with late binding.

- **High-level language bindings**

  With CORBA ORB invoking methods on server objects, the use of a high-level language of choice (C, C++, Smalltalk) is possible. The client need not concern about the implementation details of the server objects as well. CORBA separates the interface of objects from their implementation, thus providing language-uniform data types that make it possible to call objects across language and operating system boundaries.

- **Self-describing system**

To provide facilities for dynamic invocation of methods CORBA provides an Interface Repository, which contains information describing the interfaces the server supports along with its parameters. The client uses this meta-data to invoke methods at run-time. The meta-data, information regarding classes of objects at compile time, is either generated automatically by a precompiled Interface Definition Language (IDL) [18] or by compilers that generate IDL directly from an OO language.

- **Local/remote transparency**

An ORB can run standalone or can be interconnected to other ORBs in other environments. ORB can broker interobject calls within a single process, multiple processes running within the same machine or machines running across the networks. The client is transparent to all the low level details like transports, server locations, and object activation.

- **Polymorphic messaging**

ORB can invoke a method call corresponding to a specific server object. Thus, more than one server object can have the same method name. This means that the same function call may have different effects depending on the object that receives it.

2.2.1.4 Limitations of CORBA

Standard CORBA does not address the main inherent complexities of distributed computing such as Latency, Fault Tolerance and, Deadlock. CORBA does not allow objects to be passed by value. Current implementation of CORBA lack efficient support for bulk data transfer, it does not consider garbage collection, and it does not address the issue of memory leaks [19].

2.2.1.5 Embedded CORBA

As communications technologies are maturing, there is a rapidly growing need for embedded devices that can communicate to components running on remote computing platforms. The applicability of such devices is extremely wide, from the users who want devices that can serve as web browsers, email, or Internet chat clients to the telephone companies that need to manage their network devices; which justifies the need for Mobile Computing research.

Embedded applications are often highly resource constrained and typically run on a less general purpose OS than Windows or Unix, for example, WindowsCE operating systems for PDAs. It is not the CORBA standard itself that would prohibit its use in an embedded environment. It makes few assumptions about underlying operating system functionality. Although the architecture may be well suited for distributed computing even in the context of embedded devices, CORBA products face significant challenges in

the embedded environments. To be specific, the barriers to CORBA's success in embedded environments include the following.

- CORBA implementations are not typically built upon micro-kernel architecture. This makes it difficult to modify the CORBA run-time down to its bare essentials, which is important for embedded systems development.
- CORBA implementations do not typically give developers low-level control over the management of system resources, e.g. heap allocation.
- CORBA implementations are not presently open enough to support developer-supplied extensions, such as adding new transport protocols and message passing formats.

Because the programming language of choice directly influences many of these issues and their potential solutions, it is assumed that developers are using the CORBA C++ language mapping. Other language mappings could conceivably render some of these issues to be arguable, while potentially introducing new ones.

Conventional CORBA implementations experience intricate binding of all CORBA features together such that developers are forced to include all features of the CORBA architecture to use any ORB features.

More adequate alternative approach is to define the absolute core functionality that anyone who uses the ORB will require, and any other extra features layered on top of

this core. The ORB can then be customized with additional features and functionality through dynamically loadable modules on those OS's that support such functionality. On those platforms that do not support dynamic loading, the ORB can be re-built with the desired features and functionality linked in.

This is similar to many popular embedded Operation Systems that allow developers to rebuild the operating system kernel with selective functionality turned on or off. Again, this technology is hardly sophisticated, however it does demand a change of viewpoint for ORB implementers. To better meet the needs of embedded systems programmers, CORBA standard implementations need to be redesigned.

2.2.2 Java Remote Method Invocation (RMI)

Java Remote Method Invocation allows programmers to write distributed objects using Java. RMI provides a simple and direct model for distributed computing with Java objects. These objects can be new Java objects, or can be simple Java wrappers around an existing API. RMI extends the Java model; write once run anywhere, to be run everywhere as well.

Because RMI is centered on Java, it brings the power of Java safety and portability to distributed computing. It connects to existing and legacy systems using the standard Java native method interface JNI. RMI can also connect to an existing relational database using the standard JDBC package [20]. The RMI/JNI and RMI/JDBC combinations let programmers use RMI to communicate with existing servers in non-Java languages, and to expand usability of Java to those servers when necessary.

There are many advantages of using RMI. At the most basic level, RMI is Java's remote procedure call (RPC) mechanism. It has several advantages over traditional RPC systems because it is part of Java's object oriented approach [21]. Traditional RPC cannot provide functionality that is not available on all possible target platforms.

2.2.3 Jini Technology

Jini connection technology is a Sun Microsystems invention designed to allow distributed systems of components to exist on many different platforms. It lets software and hardware components become smoothly integrated into a network through the use of Java technology. Jini connection technology lets anyone connect any device to any network in a simple manner, providing mechanisms for software services or hardware devices to automatically join together into a group of Jini devices. The Jini services architecture is built upon the Java distributed computing platform architecture [22].

Devices in a Jini network are connected using Java Remote Method Invocation (RMI) explained in Section 2.2.2. This enables the Jini system to be secure and allows Java objects to move between Java Virtual Machines (VM) to implement the discovery protocol, join protocol, and the lookup service [23].

To form a Jini network of devices and services, a registration process occurs with a lookup service. When a device is connected to the network, it performs the discovery process trying to locate the Jini lookup service at which it uploads all of its interfaces for all its services, and joins the Jini network. The lookup service also has the responsibility

to behave as a control center to connected clients to a particular service. When that happens, the interface for the requested service is copied to the client.

2.2.4 Voyager ORB

Voyager is a tool that offers an object request broker, with many additional features. Voyager ORB consists of a universal communication architecture, which allows Voyager programs to be universal supporting client and server bi-directional communication with others such CORBA, RMI and other ORB architectures. The universal naming service in Voyager allows access to many commercially available naming services through a signal Application Program Interface (API). The universal directory is a directory that can be shared by all clients. For instance, an RMI server can bind an object into the universal directory using the native API for RMI registry, and a CORBA client can search the same object using the CORBA naming service API [24].

The universal messaging layer in Voyager ORB allows different types of messages such as synchronous, one-way, or future, to be sent to an object regardless of its location or object model. In synchronous messaging, the method is blocked till the result comes back. However, in one-way messaging, the method is not blocked and the result is discarded. In future messaging, a result placeholder is returned so that the method will not be blocked, and the result will be probed and obtained later.

Voyager makes efficient use of the power of Java interfaces to make accessing a remote object as simple as accessing a local object. It implements the proxy pattern [25] to associate the object with its proxy. If a method call is made to a proxy object, it is then

forwarded to the associated object. If the object is local, then the method is executed directly, resulting in performance improvement. Otherwise, if the object is at a remote site, the proxy object will forward the call, serializing parameters, the method is invoked at the remote object, and the result is serialized again and returned. This also happens to the exceptions when they are thrown. If a remote exception occurs, it is caught at the remote site and thrown locally. Proxy objects in Voyager are created dynamically if they do not exist.

Voyager has Distributed Garbage Collection (DGC), which reclaims objects when there are no local or remote references to them. It uses the Delta Pinging algorithm that keeps the traffic required for garbage collection to a minimum [26].

Voyager allows the creation of remote objects. A remote instance of a class can be created and a proxy is obtained of that object. Classes can be dynamically loaded from one site to another using a build-in Hyper Text Transfer Protocol (HTTP), which allows any voyager program to serve classes without the need of an external web server. Voyager supports object mobility. It provides a set of APIs that ease this task and make it transparent to the programmer. With object mobility, Voyager supports also autonomous mobile agents. A programmer can develop mobile autonomous agents that move themselves between hosts and continue to execute upon arrival.

Unfortunate, Voyager is supported on certain platforms, and thus we could not use it as toolkit for moving object between the WindowsCE devices and a proxy server. As a result, we had to develop our own toolkit.

2.3 Summary

In this chapter a brief survey of the well-known distributed system tools and their advantages, was given. As we have seen, some distributed architectures and tools cannot work on embedded systems due to the limitation of the embedded systems.

For the thesis, we developed a Java distributed system, similar to Voyager toolkit that works on the WindowsCE platforms in addition to traditional platforms such as Windows98/NT. The design and the implementation of this toolkit will be discussed in detail in Chapter 5.

3. Related Work

3.1 Introduction.

The lack of local resources and physical security argues for reliance on servers. However, the lack of reliable, cheap communication as well as the variable costs to access services argues for self-reliance on the part of mobile clients. The challenge for mobile computing is to strike an appropriate balance between these two competing concerns. This balance is not static one. As the circumstances of a mobile client change, it must react and repartition duties between client and server. In other words, it must be adaptive. Such adaptation may occur anywhere along a spectrum characterized by two extremes: either everything on the client or the server.

The notion of moving processes or objects around to achieve better overall performance is a well-studied topic. Process or object migration has been used successfully for load balancing and improving resource utilization. Process migration suffers from a few drawbacks when implemented in full generality. For example, it is

difficult to deal with file descriptors when being migrated. Such systems have generally been implemented to work on homogeneous networks of workstations. However, many languages and virtual machines have been developed and tools exist to overcome this limitation. Java and other scripting languages such TCL are developed to be used with virtual machines that can run tools that migrate objects and processes to other virtual machines.

Mobility is the key to adaptation to the mobile environment. Only through watchfulness and prompt reactions can a mobile client offer acceptable services in spite of the problems that spell its existence. These include unpredictable variations in network quality, wide disparity in the availability of services, limitations of the resources at the mobile device imposed by weight and size constraints, concern for battery power consumption, lowered trust and robustness resulting from exposure and motion.

3.2 Layers of Mobile Computing

This section explores in brief the mobile computing layers.

- **Applications**: Often unaware of mobility, often communication intensive, application developers may be often reluctant to change their code to customize to mobile environments. Users like to use the same applications that they are used to in the backbone network
- **Middleware**: Designed to provide transparent computing abstraction to mobile users and applications, many portable devices have very rudimentary

middleware. Writing portable middleware is nontrivial given that there is not much standardization among portable platforms yet.

- **Native OS**: Vary significantly in terms of sophistication, not always friendly to network applications, limited programming capabilities, often non-standard development environments.
- **Protocols**: Typically a modified TCP/IP or custom network protocol stack needs to address issues of wireless channel error, mobility, location-independent addressing, and heterogeneity in terms of available resources.
- **Networks**: Wireless networks from local area (Wavelan, RangeLan, NetWave, [27,28,29]) to metropolitan area (CDPD, Ardis, [30,31]) Most of these networks offer low-bandwidth and limited services, thereby making it hard to write highly adaptive applications or network software. Portable devices range from smart phones to notebooks, and vary significantly in terms of processing power, display, memory, disk size, battery power, connectivity, and programming support.

Our focus in this chapter is on the first two of the mobile computing layers.

3.3 Adaptation to Mobile Data Access

Mobile clients face many challenges. These challenges render adaptation as the key to mobile data access. There are two approaches to adaptation: application-aware, which I will explore in Odyssey; and application-transparent which I will explore in the Coda File System. Odyssey is considered a compromise solution for application-aware adaptation. It falls between two extremes. At one extreme, adaptation is entirely the responsibility of the applications. At the other extreme, application-transparent adaptation, the system has full responsibility for adaptation and the resource management. This approach is exemplified by Coda, which is suitable for legacy application because they can run unmodified.

3.3.1 Application-Aware Adaptation for Mobility (Odyssey)

In Odyssey [32], a monitor is established to monitor resources such as CPU cycles; bandwidth and battery power, and to interact with each application to best exploit these resources. For example, when high bandwidth connectivity is lost due to a radio shadow, Odyssey detects the change and notifies the interested applications. Video application, for example, may respond by skipping frames, displaying fewer frames per minute, while a Web page client will display degraded versions of large images.

The Odyssey approach for adaptation is characterized as application-aware adaptation. The essence of this model is a collaboration effort between the system and the individual applications. The system monitors the resources levels, notifies applications of relevant changes, and enforces resource allocation decisions. Each application independently decides how best to adapt when notified.

3.3.2 Coda

Coda [33] is an application-transparent adaptive support system. Coda provides clients, particularly mobile ones, with highly available access to files. Coda presents a single, global namespace to clients organized in volumes, which are sub-trees of the namespace. Applications running on Coda clients use the standard UNIX file system interface. Desktop applications can continue to run on mobile clients without modification. The client cache manager, *Venus*, is solely responsible for coping with the consequences of mobility. Coda clients are in regular use over a wide range of networks such as 10 Mb/s Ethernet, 2 Mb/s radio, and 9600-baud modems. Coda deals with the best and worst possible network conditions, and it adapts to conditions between these end points. As a starting point in understanding how Venus adapts to varying network conditions, we first explore the best case: high quality, fast LANs. In such a situation, Venus is said to be *strongly connected*. When an application opens a file in Coda, Venus checks to see if the file is already cached. If it is not, Venus fetches the file from a server to its local disk cache. When a client caches a file from the servers, it also obtains a

callback – a promise to be told if another client updates the file. When a changed file is closed, a copy of the new file contents is sent back to the servers. The servers notify any clients, with callbacks, for any file that it has changed. This is known as a *callback break*. Experience shows that this approach to maintaining file cache coherence offers excellent scalability and performance.

There is a broad range of conditions between strongly connected and disconnected operation. Coda users can operate clients over 2 Mb/s radio links, and over modems as slow as 9600 baud. As network bandwidth decreases, the importance of reordering or delaying network traffic to preserve the illusion of strong connectivity increases. To preserve the strongly connected illusion, Venus endeavors to satisfy most demand cache misses as soon as possible, and delays other traffic as necessary. These decisions are made at as high level in the system as possible. How to reschedule network traffic is revisited as available network quality changes. Adaptive decisions are made in three key areas; cache coherence, reintegration, and demand cache fetches.

3.4 Tools and Middleware for Adaptive Mobile Applications

Mobile applications need to be capable of responding to time-varying wireless-QoS and mobile-QoS conditions. Wireless transport and adaptation management systems should therefore be capable of transporting and manipulating content in response to changing mobile network quality of service conditions. Mobile signaling should be capable of establishing suitable network support for adaptive mobile services. Medium

access controllers must be capable of sharing the wireless link capacity among mobile devices supporting adaptive quality of service assurances when possible. In the following sub-sections, we will explore the major tools and middleware that support adaptive mobile applications.

3.4.1 Communications Manager for Mobile Applications

The goal of Comma [34] was to create architecture and an application programmer interface (API), for adaptive applications. The API provides a simple and powerful way for application developers to access the information required to easily incorporate adaptive behavior into their application. It provides easy-to-use methods to access this information, a wide variety of operators and ranges available to provide the application the information it needs when it needs it, a small library to link with to minimize the overhead placed on the client and to minimize the amount of data that needs to be transferred between the clients and the servers.

In a future release, the communication could be changed to use XDR [35] and pack the transferred data more efficiently than is done currently. Comma is not a network management protocol, and it is not designed as a replacement for SNMP [36]. A Comma application could certainly provide the same functionality as an SNMP manager by polling each Comma server on a network for the required SNMP variables. However, this was not the motivation behind Comma.

3.4.2 Rover Toolkit

The Rover toolkit [37] offers applications a distributed-object system based on the client-server architecture. Clients are Rover applications that typically run on mobile hosts, but could run on stationary hosts as well. Servers, which may be replicated, typically run on stationary hosts and hold the long-term state of the system. Communication between clients is limited to peer-to-peer interactions within a mobile host (using the local object cache for sharing) and mobile host-server interactions; there is no support for remote peer-to-peer or mobile host-mobile host interactions. The Rover toolkit provides mobile communication support based on two ideas: re-locatable dynamic objects (RDOs) and queued remote procedure call (QRPC). A re-locatable dynamic object is an object with a well-defined interface that can be dynamically loaded into a client computer from a server computer, or vice versa, to reduce client/server communication requirements. Queued remote procedure call is a communication mechanism that permits applications to continue to make non-blocking remote procedure calls even when a host is disconnected; requests and responses are exchanged upon network reconnection.

The key task of the programmer when building a mobile-aware application with Rover is to define (RDOs) for the data types manipulated by the application, and for data transported between client and server. The programmer then divides the program into portions that run on the client and portions that run on the server; these parts communicate by means of QRPC. The programmer then defines methods that update

objects, including code for conflict detection and resolution. To use the Rover toolkit, a programmer links the modules that compose the client and server portions of an application with the Rover toolkit. The application actively cooperates with the runtime system to import objects onto the local machine, invoke well-defined methods on those objects, export logs of method invocations on those objects to servers, and reconcile the client's copies of the objects with the servers. Earlier work on Rover introduced the Rover architecture, including both queued RPC and re-locatable dynamic objects. Some suggested enhancements to the toolkit extend the design and implementation of QRPC and RDOs with compressed and batched QRPCs.

There are several steps involved in porting an existing application to Rover or creating a new Rover based application. Each step requires the application developer to make one of several implementation choices. While Rover does not provide any tools for building applications, it does provide a consistent framework. The first step is to split the application into components and identify which components should be present on each side of the network link. It is very important that application developers think carefully about how application functions should be divided between a client and a server. The division will be mostly static, as most of the file system components will remain on the server and most of the GUI components will remain on the client. However, those components that are dependent upon the computing environment (network or computational resources) or are infrequently used may be dynamically generated. For example, the search operation performed by a client could be dynamically customized to

the current link attributes: over a low latency link, more work could be done at the client and less at the server, and vice versa for a high latency link. Likewise, a client could pre-fetch the main portion of an application's help information but less frequently referenced portions could be loaded on demand. Once the application has been split into components, the next step is to appropriately encapsulate the application's state within objects that can be replicated and sent to multiple clients. Also, the application developer must decide which mechanisms to use for notifying users of the status of displayed data.

3.4.3 Sumatra.

Sumatra [38] is an extension of the Java programming environment that supports adaptive mobile programs. Platform independence was the primary basis for choosing Java as the base for Sumatra. In the design of Sumatra, the Java language was not altered. Sumatra can run all legal Java programs without modification. All added functionality was provided by extending the Java class library and by modifying the Java interpreter without affecting the virtual machine interface. Policy decisions concerning when, where and what to move are left to the application. The high degree of application control allows programmers to easily explore different policy alternatives for resource monitoring and for adapting to variations in resources. Sumatra has two additional programming abstractions besides Java abstractions: *object groups* and *execution engines*. An *object group* is a dynamically created group of objects. Objects can be added to or removed from *object groups*. All objects within an *object group* are treated as a unit for mobility related operations. This allows the programmer to customize the granularity

of movement and to amortize the cost of moving and tracking individual objects. This is particularly important in languages like Java because every data structure is an object and moving the state of one object at a time can be expensive. An *execution engine* is the abstraction of a location in a distributed environment. It corresponds to an interpreter executing on a host. Sumatra allows *object groups* to be moved between *execution engines*. An *execution engine* may also host active threads of control. Objects in an *object group* are automatically marshaled using type information stored in their class templates. When an object group is moved, all local references to objects in the group (stack references and references from other objects) are converted into proxy references, which record the new location of the object. Some objects, such as I/O objects, are tightly bound to local resources and cannot be moved. References to such objects are reset and must be reinitialized at the new site. The class template for an object (and the associated bytecode) can be downloaded into an *execution engine* on application request.

Method invocations on proxy objects are translated into calls at the remote site. Type information stored in class templates is used to achieve RPC functionality without a stub compiler. Exceptions generated at the called site are forwarded to the caller.

Sumatra provides a resource-monitoring interface, which can be used by applications to register monitoring requests and to determine current values of specific resources. When an application makes a monitoring request, Sumatra forwards the request to the local resource monitor. If the monitor does not support the requested operation, an exception is delivered to the application.

3.4.4 Mobiware.

Mobiware [39] is based on a methodology of open programmability [40] for the introduction, control and management of new adaptive mobile services. It provides a set of open programmable CORBA interfaces and objects that abstract and represent network devices and resources, providing a toolkit for programmable signaling, adaptation management and wireless transport services.

Mobiware provides a foundation for open programmable mobile networking that is suited toward managing the evolving service demands of adaptive mobile applications and dealing with the inherent complexity of delivering scalable audio, video, and real-time services to mobile devices.

Built on an adaptive quality of service API, Mobiware consists of a set of controllers that interact with transport, network and medium-access controller distributed objects that maintain application-specific adaptive quality of service needs. This API is specifically designed to quantitatively address the wireless-QoS and mobile-QoS needs of adaptive mobile applications. Mobile applications use this API at the transport layer specifying *a utility function* that maps the range of observed quality to bandwidth. The observed quality index refers to the level of satisfaction perceived by an application at any moment. The *adaptation policy* captures the adaptive nature of mobile applications in terms of a set of adaptation policies fast, smooth, after handoff or never. These policies allow the application to control how it moves along its utility curve as resource availability varies.

A simple set of adaptation policies is used in Mobiware to capture how an application wishes to respond to instantaneous bandwidth availability. A mobile multimedia application's range of perceptible quality is strongly related to how and when it responds to resource changes. Frequent variations between what may be considered optimal and minimum utility or even the frequent small change around an average application quality may be not appropriate to many applications. Mobile applications use this API to specify flow utility functions and adaptation policy. The adaptive-QoS API allows applications to associate temporal or event-based dimensions with their utility functions.

3.5 Summary.

From the previous work, we do note that the adaptation parameters being focused on are the network states and the mobile device computation power: bandwidth and the network latency and CPU cycles. The bandwidth and network latency were the main parameters upon which the adaptation would take place. In previous adaptation approaches, it is up to the application to the decide how to react to changes to the network state. The reaction could be filtering data, reducing the size of data, entirely changing the content of the data, or limiting the computation of specific tasks if necessary. This argues for exporting the network state as well as available resources of the mobile device to the mobile applications to be designed to be adaptive.

On the other hand, the automation of adaptation to the resources was not explored. There are a lot of similarities between our work and the work in Sumarta. Both Sumarta and our work use extended Java Virtual Machine for portability issues and the ease of use of the language especially for implementing object mobility toolkits. The main difference between our toolkit and Sumarta toolkit is that Sumarta entirely leaves the adaptation policy under the control of applications. The applications are fully responsible for moving objects between the mobile device and the proxy server or reducing the computation of tasks. In other words the reaction to changes in the environment is left to the application. However, to build a fully automated toolkit to automate the adaptation process at run time, application behavior must be known before hand. Capturing and modeling the application behavior, in our opinion, is a difficult problem to solve.

In our work, adaptation to the change in the resources and environment is partially left to the toolkit. We try to automate some adaptation policies transparently to applications. For example, instead of reducing the computation power for a specific task, first we try to move and execute the task remotely at powerful machines in favor to reduce the CPU cycle at the mobile device, which in turn may result in reducing the power consumption at the mobile device too. If this adaptation policy costs more, then we signal the application to reduce the computation for the specific task. In other words, we try first to use available remote resources to achieve the same task; otherwise, as last resort, we let the application do the adaptation.

Our goal is to provide a toolkit that would help the adaptation process and not fully automated. We try to make the adaptation process as transparent to applications as possible. In our design of the toolkit we consider extra parameters to network state parameters; namely the power consumption and relative CPU speed between the mobile device and the proxy server; however, we assume that applications are still be aware of these parameters as well. Thus our goal is to design a Java toolkit that combines two Java Virtual Machines as one virtual machine application point of view, and to automate the object load or distribution between the two Java Virtual Machines. The overall goal is to reduce the power consumption and to increase the performance of the Java application for PDAs.

4. Basic Idea and Preliminary Studies

4.1 Introduction

Power management is one of the obstacles that make portable computer less useful. Battery power limits the utilization of portable computers to be used anytime and anywhere. Quite often, AC power connections are not available, and the portable device must work on battery power. However, battery life of existing ones and batteries that are expected to exist in the future will not be sufficient for many situations. The projection on progress in battery technology shows that only a 20% improvement in battery capacity will occur over the next decade [41]. Users either alter their behavior or limit their use of the portable device to preserve the battery life. If capacity of battery power cannot be improved, another alternative solution is to find ways to reduce power consumption over time, provided that these solutions do not have a great impact on the user.

Many researchers have investigated this problem [42,43,44]. Solutions range from slowing down CPU clock rate [45] to intelligently managing screen and disk during idle periods or turning off some computer components that are not in use [45,46,47,48]. Some

of these solutions made their way into commercial use, which gives a real indication of the importance of power management.

Wireless communication is becoming more common in portable devices. In the absence of a wired connection, a wireless connection allows to maintain network connectivity, allowing remote file access and sending and receiving emails as well as browsing the Internet. Hence, they open a new window of opportunity to overcome some of the portable device computation limitations. However, wireless communications is regarded as contributing to the power management problem since sending and receiving data consume a considerable amount of power. Nevertheless, it can also be used to save a significant amount of power, as demonstrated with a simple example in this chapter.

In portable devices, performing tasks for a user drains power. Some of these tasks must be performed locally due to the nature of the tasks, for example, graphic user interfaces and the information that is required to be displayed to the user. However, other tasks could be executed anywhere provided that they return results back to the portable device. Hence if the power cost model for sending tasks elsewhere and getting the results back is less than the cost model of performing the task locally, then remote execution could save a significant amount of power. In this chapter, we will explore this idea by supporting portable devices through a dedicated machine called proxy server. A proxy server is a fast machine, compared to mobile devices, which a mobile device can have access to or communicate with directly in fixed networks. Thus, executing tasks remotely

may also help decrease the response time and improve the performance of mobile applications.

4.2 Background on Load Sharing in Distributed Systems

In general, load sharing or balancing can be defined as a strategy in which every processor, in distributed systems, would have an equal load. However, some researchers distinguish between load balancing and load sharing as follows. Load balancing is defined as a strategy in which every processor having the same load is the targeted goal. Load sharing is defined as a strategy which attempts to have processors share the load. A load sharing strategy involves two policies. A transfer policy decides when a job must be transferred, which depends on the number of jobs that are waiting to be served in the local queue. The location policy decides which host should the jobs transfer to. This is done either by randomly choosing a host or by analyzing workload information, which may be obtained either by probing a subset of hosts or by collecting the information periodically. If information is collected periodically, then the optimal period must be determined as well.

Many load-sharing algorithms have been proposed, and they can vary from ones that do not use system state information- random algorithms - to ones that use global state information. Optimal Load Sharing Algorithms is an example for algorithms that use global state information [49].

4.3 Mobile Computing and Load Sharing Algorithms

The main idea behind mobile computing is to enable the users to access a network or the Internet regardless of their location. Ideally users should get the same quality of service as if they were connected to the network using a wired connection. This is not possible since drawbacks of wireless connections such as instability in bandwidth and latency will play a major factor in degrading the quality of service. In addition to the wireless connection drawbacks, the limitation of the portable device in terms of computation power and memory capacity as well as power consumption will also affect the quality of service that the user obtains. Even though these limitations are being addressed by many technologies, it is most likely that performance difference between the wired and wireless connections remain [50]. [51] suggests that the computation, especially intensive ones, and communication should be done at the proxy side as much as possible for the reason that portable devices have very limited resources.

By offloading computation and communication to a proxy server in the wired network, reductions in computation and power consumption on the portable device can be achieved. In order to conserve power, transmitting data must be kept at a minimum, since the power consumption for transmitting a message is often higher than receiving a

message of the same size in wireless devices [48]. Optimizing applications so that they require fewer operations also reduces power.

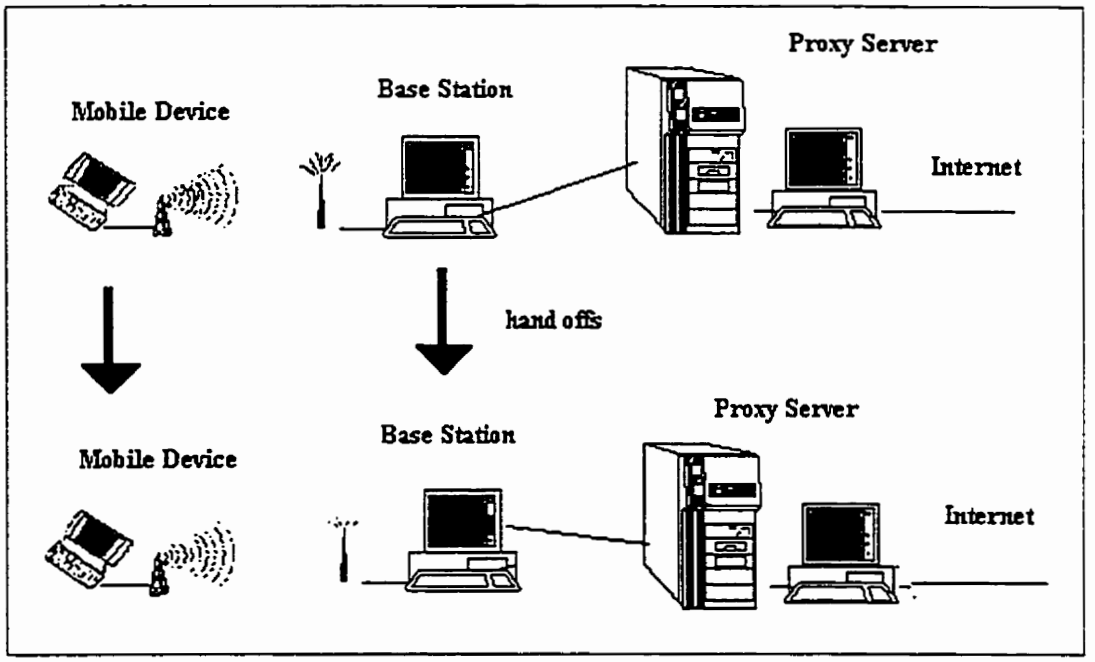

Figure 4.1: General Proxy Infrastructure

Assumptions made regarding load sharing in distributed systems are no longer valid in wireless networks, where mobility introduces new challenge. For example, the location policy selects an appropriate host for the job to be transferred. In wireless systems, the only fixed host is the proxy server the portable device communicates with directly as indicated in Figure 4.1. Thus jobs can only migrate to this proxy server. Therefore, using any existing location policy between the proxy server and the fixed hosts, in case the proxy server decides to delegate jobs to other fixed host in the network, is not a concern for the portable device. In other words, the other fixed hosts in the

network are transparent to the portable device as long as the computation is performed and the results returned when needed.

To probe and collect information, related to monitoring and load balancing decision, from the proxy server and the portable device, probing messages are needed. The overhead caused by probing messages is not negligible since the bandwidth in wireless networks is limited and the cost model for send and receiving data is not cheap in terms of power consumption at the portable device.

4.4 Java and WindowsCE

Load sharing algorithms depend on the homogeneity of the platforms in the network when transferring jobs from one host to another in a fixed network. This assumption is not valid when it comes to the variety of mobile computer platforms in practice today. It is, therefore, not possible to support the proxy server with all the platforms that possibly connect to it at any moment in time. Using Java Virtual Machine as the uniform platform will help solve this but at an extra cost, overhead, which is not negligible especially for small portable devices such as handheld devices.

In our study, we used a handheld WindowsCE 2.0 device to measure the power consumption as well as the response time of Java applications. To build power-aware applications in Java we had to extend the Java Virtual Machine on the portable device to be able to trace power consumption. Using Java Native Interface, a subset of Advanced

Power Management APIs that are supported by WindowsCE is exported to extend the Java Virtual Machine [52].

Since the Java Virtual Machine for WindowsCE is relatively immature, some problems had to be addressed and solved. One of them was that the values returned by the "System.currentTimeMillis()" were rounded up to milliseconds that are divisible by 1000. We fixed this bug by exporting the correct time in milliseconds through the Java Native Interface.

De/serialization of objects is the key for storing objects state and moving objects between Java Virtual Machines. Proxy objects are needed to forward messages to the real objects, which may reside in remote Java Virtual Machines. Remote Method Invocation protocols are mainly responsible for de/marshaling methods' parameters and return objects, when invoking methods on remote objects.

Unfortunately, the De/serialization functionality of objects in WindowsCE Java Virtual Machine is malfunctioning. Namely, the serialization algorithm was not using serialization version code as specified in the Object Serialization Specification [53].
To de/serialize objects, we had to fix this bug by forcing the algorithm to use our serialization version code.

WindowsCE Java Virtual Machine by itself does not support object mobility. Utilities and packages need to be developed to support object mobility. For example, Voyager [54] has very extensive facilities to support object mobility for Java programs. Remote Method Invocation [55] helps to develop Java applications that can invoke

methods on remote objects. However, these utilities and packages are built to work on PCs rather than on PDAs, due to the PDA limitations.

We developed a utility that can work on PDAs that supports object mobility. It allows moving objects dynamically and according to the state of the mobile device and the mobile computing environment. Chapter 5 discusses in detail the structure, framework, and API, of this tool.

4.5 Experiments

Since the main goal of using a proxy server is to save power and increase the response time for an application by migrating jobs, the same can be achieved using the notion of mobile code. Java has the ability to serialize objects at one host and to load and execute them at runtime at another host. We identified the important parameters based on which dynamically adaptive mobile applications and toolkits can take decisions as to which subset of an application's objects must be migrated and executed at the proxy side according to a specific criteria.

Some obvious parameters such as power consumption cost model, bandwidth, and relative CPU speeds are used in our case study. To show that load sharing can have a major impact on the response time and power consumption, we studied CPU-intensive Java applications. We did choose to implement a float matrix multiplication algorithm of order O (N^3) in Java as an example of CPU intensive computations that occur during coding and decoding multimedia files in MP3, JPEG and MPEG format. Multimedia

support will have potential relevance in future mobile applications. The same matrix multiplication algorithm was implemented and locally executed at the handheld device. Then, it was executed remotely at the proxy server and results returned back to the handheld device. We measured response time and power consumption in the handheld device and simulated the transmission and reception costs due to unavailability of the wireless connection card for the handheld device that we used, HP 620LX. The response time measured when remote execution occurred includes the cost of sending the code as well.

The battery of the handheld device that is used can provide (7.2 V*1.35 A) 9.72 W. If we assume that the general power consumption of the handheld device is 5 W-hr, then the battery will last for 2 hours approximately. Table 4.1 shows receiving and transmitting power costs for a WaveLAN PCMCIA wireless card [56]. We assumed in our power consumption calculations these values as constant values although in reality theses values are variable depend on the distance of between sender and receiver devices. In this thesis we did not consider the power consumed caused by the mobile device being idle during the handover process.

Table 4.2 shows the power consumption equations used in our case study, where the **Ts** is the transmitting time, **Tr** is the receiving time, and the **Te** is the execution time of a process in seconds. For example, if the available bandwidth is 19.2 Kbps, and the data being send is 1024 bytes, then the power consumed when sending the data is 0.00036166 W-hr.

Table 4.1: Receiving and Transmission Costs of Wave LAN PCMSCA Card

State	Watts
Receiving	1.52W
Transmitting	3.1W

Table 4.2: Equations of Power Consumption Costs for Transmission, Receiving, and Computing

Power consumed when	Equation
Transmitting	$3.1 \times \frac{Ts}{3600}$ W-hr
Receiving	$1.52 \times \frac{Tr}{3600}$ W-hr
CPU	$1.8 \times \frac{Te}{3600}$ W-hr

We also run another experiment that involves studying response time as well as power consumption of decoding a GIF image locally at the handheld device and remotely at the proxy server by sending the decoded result back to the handheld device as pix-map [57].

4.6 Results

To show that sending Java objects to the proxy server can help improve mobile application performance, we varied bandwidth, proxy server CPU speed and data size being sent and received over the wireless connection and observed the response time as well as the power consumption on the mobile device. The data transfer may include the object code or not since it is possible that the object code already exists on the proxy server. Results show that the available bandwidth is an important factor in determining whether an object should be migrated or not. We varied bandwidth between 1562.5 bits/sec and 200kbits/sec. For higher bandwidth, better application performance was achieved, and also less power was consumed on the portable device.

The following graphs show the response time for multiplying two float matrices of size 10x10 and 400x400 as well as the power consumption when the object is executed locally and remotely, where object code size is 1.27 KB and

- RRT1C: response time when executing the matrix multiplication object at a 200Mhz Pentium proxy server and the data being sent includes the object code.
- RRT1NC: response time when executing the matrix multiplication object at a 200Mhz Pentium proxy server and the data being sent does not include the object code.
- RRT2C: response time when executing the matrix multiplication object at a 300Mhz PentiumII proxy server and the data being send includes the object code.

- RRT2NC: response time when executing the matrix multiplication object at a 300Mhz PentiumII proxy server and the data being sent does not include the object code.
- LRT: local response time when matrix multiplication is executed on portable device.
- LPC: power consumption on the portable device when executing the matrix multiplication object on the portable device.
- RPC: power consumption on the portable device when executing the matrix multiplication object at the proxy server, which includes power consumption of sending and receiving data including object code.
- RPNC: power consumption on the portable device when executing the matrix multiplication object at the proxy server, which includes power consumption of sending and receiving data without object code.

In the following graphs some curves are overlapped, and this is mainly because code size is amortized by data size being sent with the code. In Figure 4.5 we choose not to show the RPNC curve because of not noticeable difference with PRC curve.

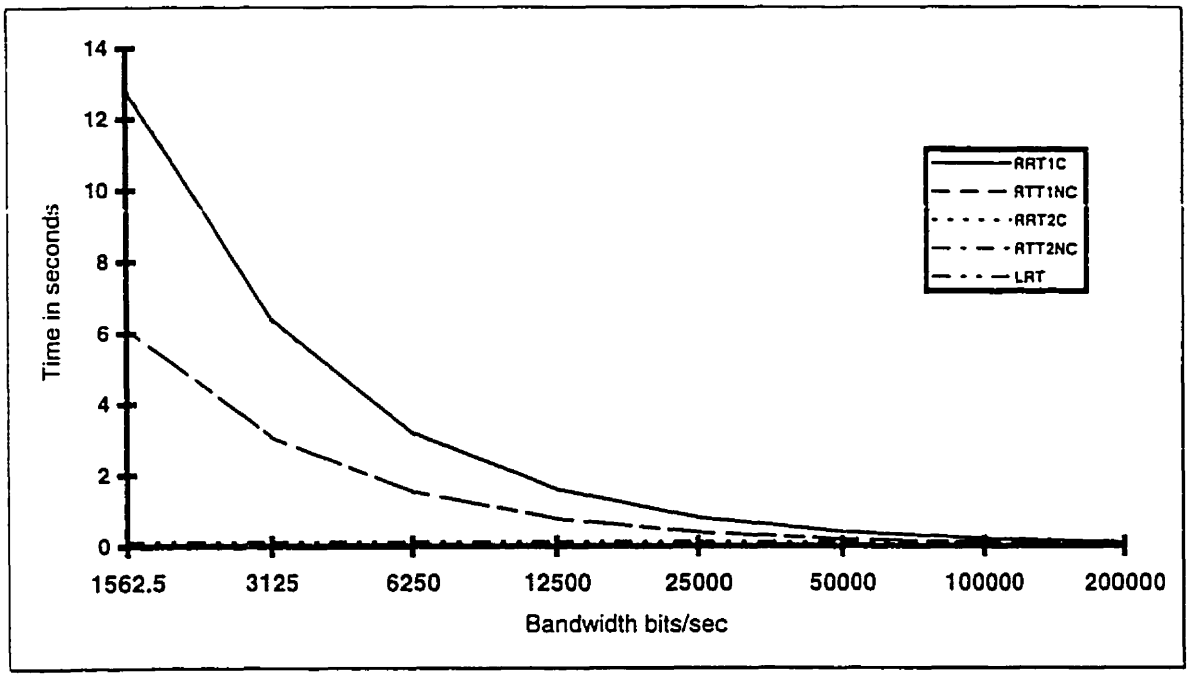

Figure 4.2: Response time for matrix multiplication of size 10x10 locally and at the Proxy side.

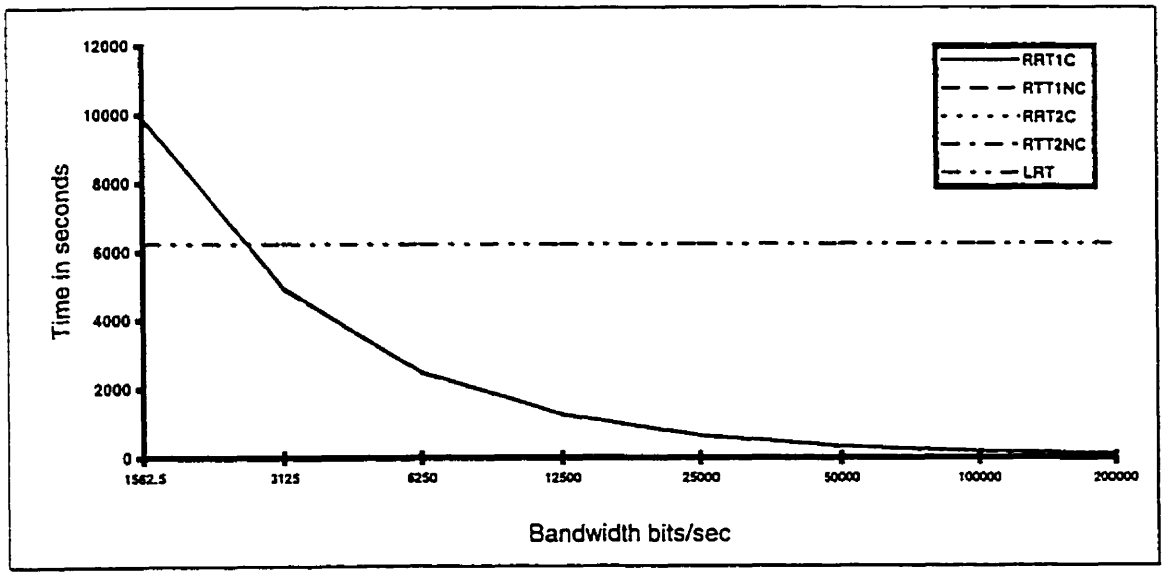

Figure 4.3: Response time for matrix multiplication of size 400x400 locally and at the proxy side.

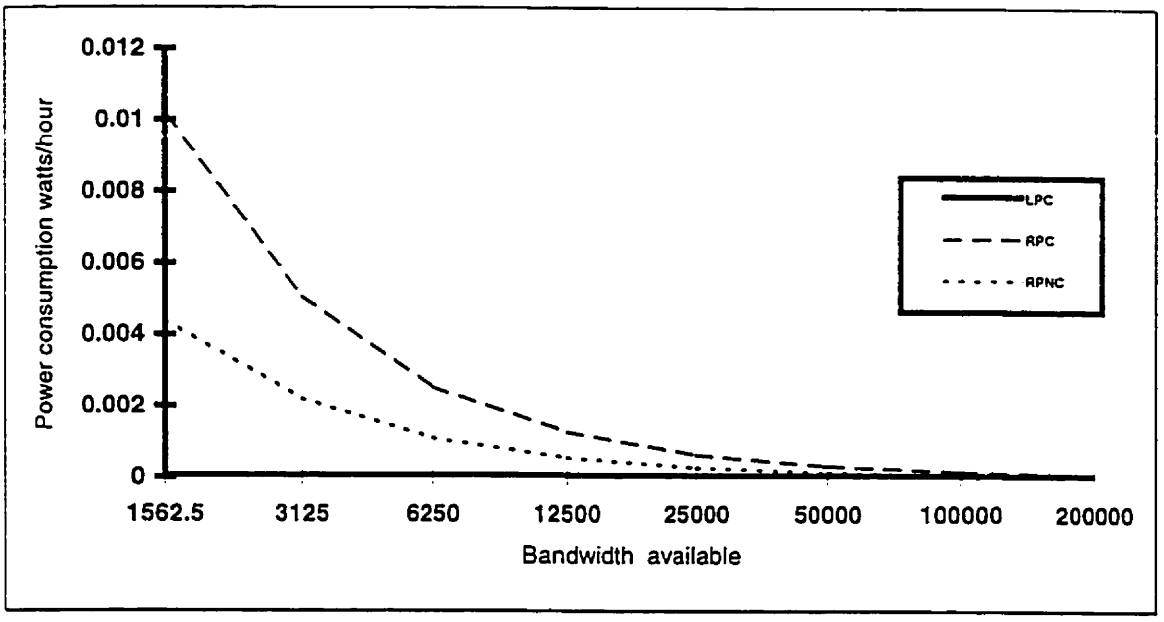

Figure 4.4: Power consumption for matrix multiplication of size 10x10

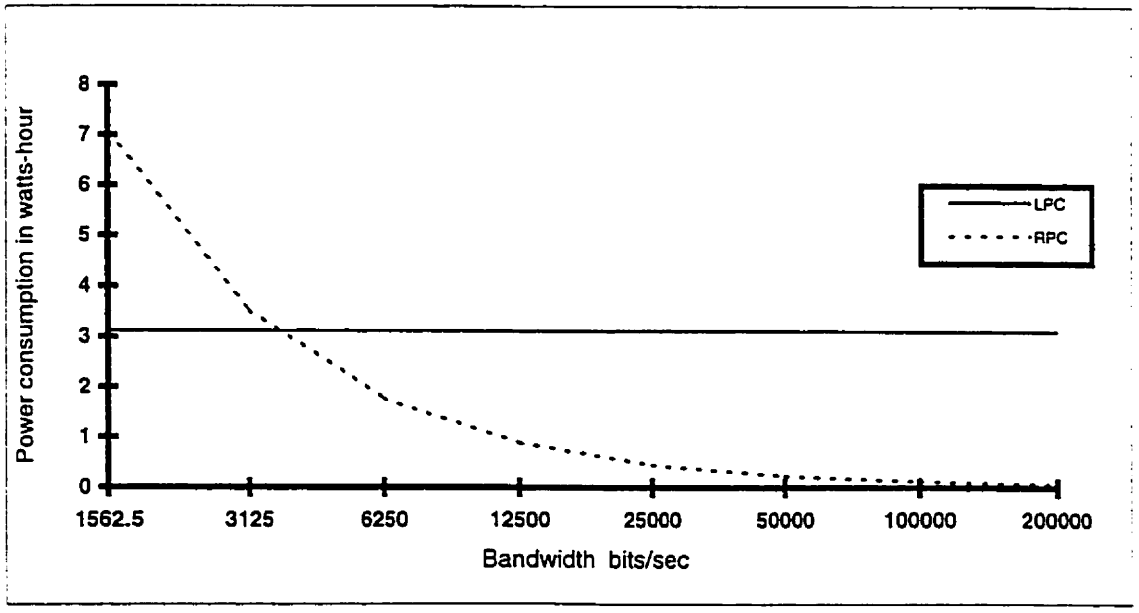

Figure 4.5: Power consumption for matrix multiplication of size 400x400

4.7 Discussion

Figure 4.2 and Figure 4.4 show the response time for matrix multiplication of size 10x10 and 400x400 respectively. We choose these sizes to represent small and large problem sizes. Both graphs show the importance of the available bandwidth in determining the tradeoffs between response time and the power consumption costs. Figure 4.2 indicates that for small problem sizes, to increase the response time, relatively high bandwidth is required (100000 bits/sec or more) for offloading to be beneficial. Figure 4.3 shows that for large problem sizes, offloading is beneficial at relatively low bandwidths (from 3000 bits/sec and more) to increase the response time.

Figures 4.4 and Figure 4.5 show the power cost for small and large problem sizes respectively. For small problem sizes, it needs again relatively high bandwidths (200000 bits/sec or more) to start reducing the power consumption; however, for large problem sizes, the reduction in power consumption happens at relatively low bandwidths (starting from 3500 bits/sec).

From previous figures, the tradeoffs depend on problem size, bandwidth, and the relative CPU power between the mobile device the proxy server, which is indirectly deduced from the problem sizes.

For non-trivial problem sizes, the results show that interesting tradeoffs appear at low bandwidths, although the response time could be very high even when using the proxy server strategy. However, obtaining low response time as well as saving power for

small problems requires relatively high bandwidths (200000 bits/sec or more), which seems feasible with the WaveLAN wireless technology and third generation cellular systems. Even at data rates that are achievable with today's wireless technologies, offloading of computationally intensive components of an application appears promising. As a result we are motivated to continue explore and demonstrate via prototype prove that it is possible to increase the performance and/or reduce power consumption.

5. Dynamic Object Mobility Toolkit

5.1 Introduction

Programs that use mobility as a mechanism to adapt to resource changes have two main requirements that are not shared with other mobile programs. First, they need to monitor the level and quality of resources in their operating environment. Second, they need to be able to react to changes to resource availability. In this chapter, we describe the design and implementation of our object mobility toolkit, an extension of Java that supports resource aware mobile programs for PDAs.

Mobile programs can move an active process or task from one site to another during execution. This flexibility has many potential advantages. For example, a program that searches distributed data repositories can improve its performance by migrating to the repositories and performing the search on-site instead of fetching all the data to its current location.

Applications running on mobile platforms can react to a change in network bandwidth by moving network intensive computations to a proxy host on the static network as indicated in Chapter 4. The primary advantage of mobility in these situations is that it can be used as a tool to adapt to variations in the operating environment.

Applications can use online information about their operating environment and knowledge of their own resource requirements to make judicious decisions about placement of computation and data. However, in our toolkit, we try to automate this process in two ways. First, by adapting to changes in the mobile environment by sharing the load between the PDA and the Proxy host. Second, we allow applications to use online information about the mobile environment to trigger their own adaptation polices. If the first approach results in a satisfactory QoS to the user, then the second option need not be executed.

Many systems provide some form of support for program mobility. The simplest form of support is the ability to download code and execute it to completion at a single site. Omniware [58], Safe-TCL [59], Java [60] are examples for such systems. Other systems like Avalon [61], NCL [62] REV [63] and Obliq [64] allow programs in execution to initiate computation on remote nodes and wait for their completion. The most sophisticated support is provided by systems like Agents [65].

5.2 Designing Mobile Applications

Traditional applications consuming many resources do not run efficiently on mobile computers. An approach to solve this problem is to divide the application at design-time into two pieces, one in the mobile host and the other at the stationary computer. The piece consuming fewer resources would be running on the mobile host, and the other would be running on the stationary computer. Another approach is to divide an application at run-time into two pieces. As the resources and the environment change, the two components will be reconfigured accordingly.

Two issues are important for realizing application adaptation. The first is that the operating system must support a mechanism of notifying applications of changes in the mobile environment. The second is to provide a systematic way to build adaptive applications embodied in frameworks and toolkits.

5.3 Overview of Proxy Server

The central concept of our framework is the proxy server host. A proxy server is an intermediate device that communicates with servers in the Internet using standard Internet protocols as shown in Figure 5.1. The mobile device and the proxy server may communicate through protocols suitable for wireless connections, such as I-TCP [66] or standard TCP. A typical proxy server can be used for the following:

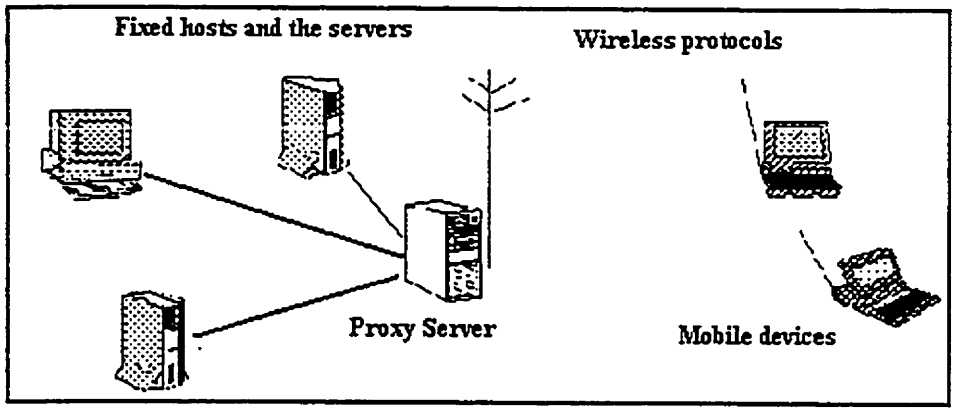

Figure 5.1: Proxy Sever infrastructure

- A proxy server can work as a filter receiving data from the Internet and compress received data according to the need of the mobile device. For example, color video streams are converted from gray color to back-and-white color, the quality of audio streams can be altered from stereo to mono, or the replay sampling frequency can be reduced to minimize the size of data over the wireless connection.
- An application can use the resources of the proxy server to increase the performance and decrease the power consumption by deploying applications' selected objects on the proxy server. For example, offloading a heavy computation objects such as decoders to a proxy server reduces the CPU cycles on the mobile device as shown in the previous chapter

5.4 Dynamic Object Mobility Toolkit

Distributed systems for mobile hosts are difficult to design due to the constraints of the environment. One approach to simplify the design and implementation of mobile computing applications is to provide a uniform programming language level of abstractions through which all mobility related events and actions are reported and performed. Functionality of applications is encapsulated within this high level abstraction and makes an application easily portable.

To deal with theses issues, a mobile application must be designed using Object-Oriented Design, and run on homogenous platforms. A mobile application is developed by composing objects containing functionalities. We call the composition of objects the object graph. A mobile application can have two object graphs. One resides at the mobile device, and the other resides at the proxy server. These graphs change according to the mobile environment and the resources of both devices. An object in the object graph can be a filter, buffer or computation object. The Object graph will be reconfigured as needed whenever there is a change in the mobile environment. We built a framework using Java, which notifies objects about changes in the mobile environment and the configuration of the device.

5.4.1 Overview of the Toolkit

The programming support we propose can be classified into:

- Support for information delivery to the application.
- Support to allow an application to react suitably.
- Dynamic Object Mobility.

The toolkit has a set of APIs, which provide the required functionality for moving objects dynamically. The choice of the Java language was motivated by the properties of the language and the portability issues that Java platform offers. Figures 5.2 and 5.3 show the main structure of the distributed toolkit. There is not much difference between the structure of the toolkit at the mobile side and the proxy side, except that the migrating-objects decision is taken at the proxy side since the decision process of the taking decision consumes CPU cycles, which would consume power as well. The following is a brief description of the block diagram in Figure 5.2 and Figure 5.3. Figure 5.4 shows part of the class diagram that interests us more for our thesis.

- **Mobile/Proxy Device State and Information:**

This unit is responsible of monitoring and delivering the state of the mobile or proxy device as events to the Object Server. Changes in the bandwidth or changes in the power status are examples of the events that this unit exports.

- **Code Storage:**

This unit works as storage of the validated classes files (bytecode) at the mobile device. At the request of the proxy device, the code will be transferred to the proxy.

- **Object References and Profiling (Object graph):**

This unit contains the representation of the application's objects along with the profiling information about these objects. These information will be send to the proxy sever to be analyzed and the proxy server will decide which object must be shipped to its side according to the mobile environment.

- **Object Server**

This unit is considered the main core of the toolkit. It runs a thread that listens continuously to all the commands from a remote object sever. Commands can be related to moving objects or related to the remote invocation of a method on a remote objects.

- **Remote Method Invocation Protocol**

This protocol is used to marshal and un-marshal a method's parameters

- **Dynamic Decision**

This unit is responsible for the analyzing of the profiling information of application's objects. It resides only at the proxy server. Having decided which objects need to be executed at the proxy server, it will issue a command to the remote object server to download the objects.

- **De/Serialization State of Objects Protocol**

  This contains the implementation of the serialization protocol if the JVM does not implement one.

- **Communication Control Layer**

  To simulate wireless links in terms of the low bandwidth, we chunk the data streams being sent through the communication layer into pre-determined sizes based on the empirical tests. We introduce a controllable amount of delay between data chunks to finally get a simulated slow like. This allows us control the throughput dynamically at run time for testing purposes. The following equation is used to determine the simulated throughput. Changing both the delay and the chunk size changes the throughput.

$$Throughput(Kb/s) = \frac{10(ms) \times 838(Kb/s) \times Chunksize(bytes)}{Delay(ms) \times 1024(bytes)}$$

Therefore in the implementation of this layer we provided two APIs that control these values. Please refer to the example provided in appendix B for how to uses APIs.

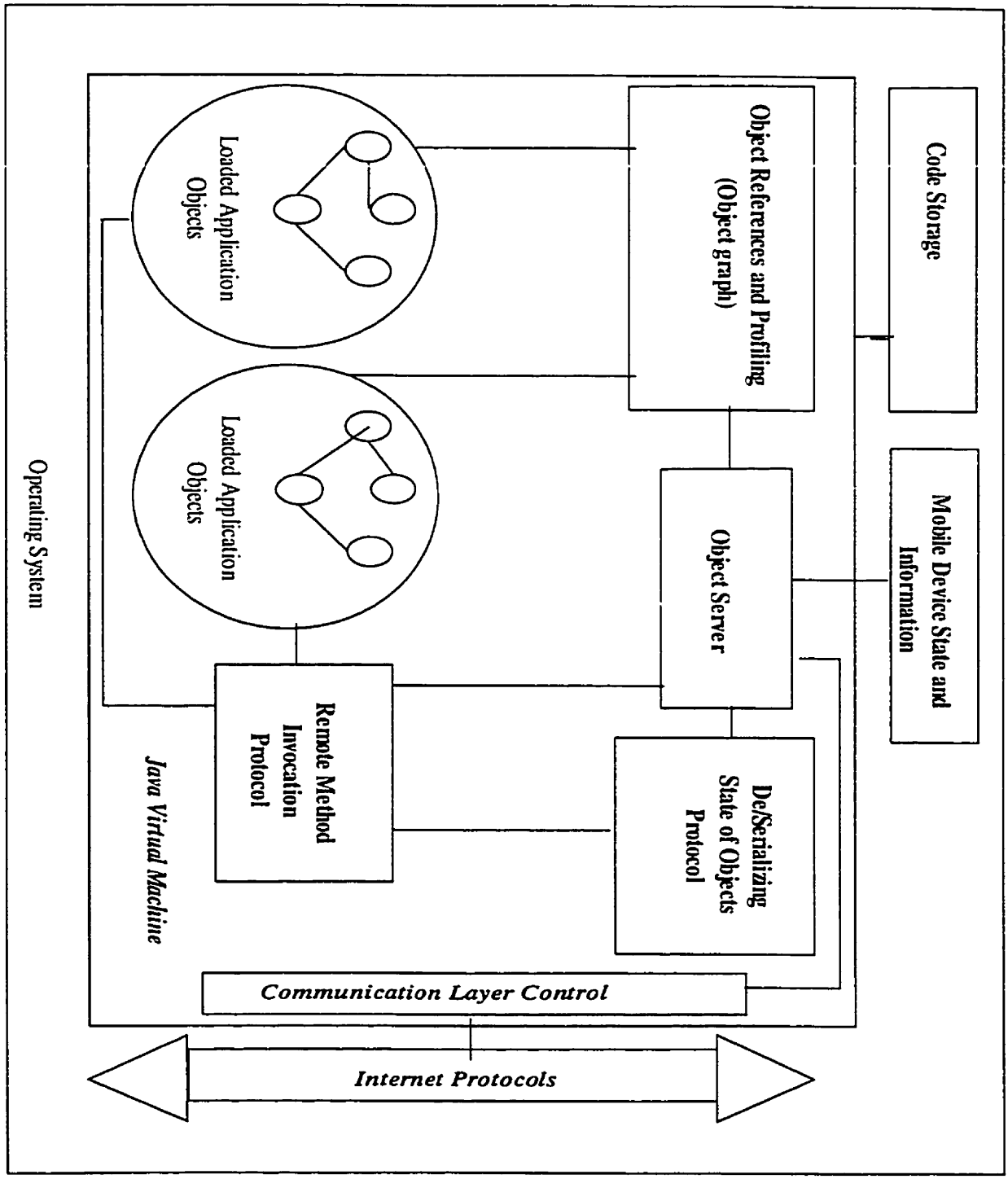

Figure 5.2: Object Mobility Toolkit infrastructure at the Mobile Device.

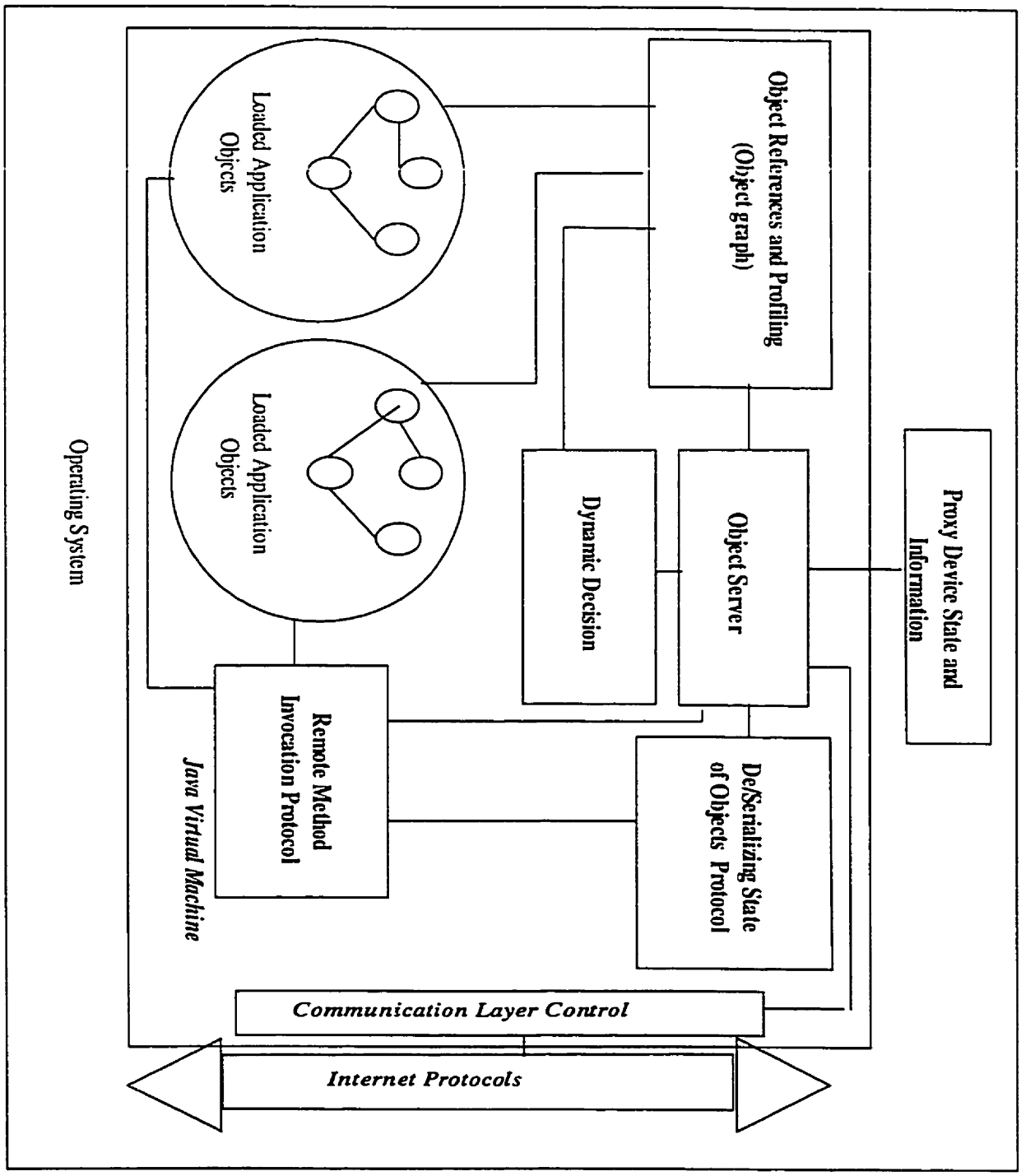

Figure 5.3: Object Mobility Toolkit Infrastructure at the Proxy Server.

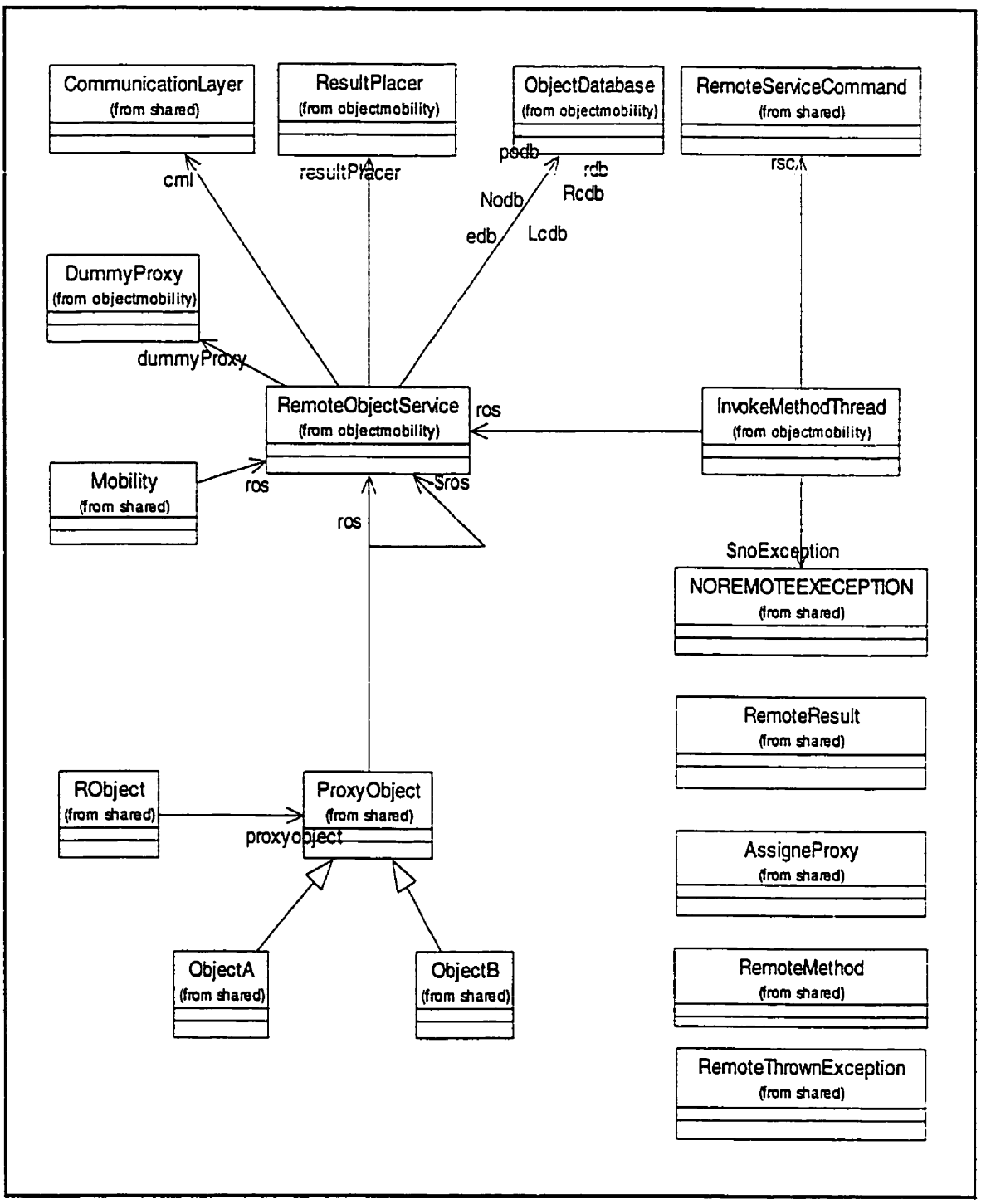

Figure 5.4: Part of Class diagram for Object mobility toolkit.

As mentioned earlier, a mobile computing application needs to be aware of resource availability and changes in the mobile environment. Thus special abstractions must be provided in order to deliver these changes to an application. We model all changes as events, which are delivered to objects. Interested objects in an application must define an event handler through which the events, such as change in the power state and bandwidth of link, can be handled. Since Java does not support pointer notion, using Java Reflection classes and Interfaces facilitates this.

Both the state and computation of an application may be partitioned between the mobile device and the proxy server. The degree of partitioning ranges from just executing the user graphic interface to executing the entire application on the mobile device. We propose a Greedy Graph Partitioning algorithm [67] for load-sharing purposes [68] to be used with this toolkit for handheld PCs. The decision to move objects is made at run-time and depends on environment factors, mainly relative CPU speeds and link bandwidth. Chapter 7 describes the algorithm and results. The toolkit provides the following functionality:

- Migrate the object to a remote host.
- Fold the object back.
- Objects migrating decision are made dynamically and initiated by the proxy server

5.4.2 Design of the Toolkit

The core design of toolkit is based on a simplified implementation of Proxy Object pattern, Object Remote Procedure Call and Object Mobility. Our goal is to have an extended Java Virtual Machine enabled with a toolkit that facilitates the mobility of objects between mobile host and proxy server in a dynamic manner, transparent to the application designers and users. This toolkit is designed to work on PDAs. Because of the PDA constraints mentioned earlier, the toolkit must have a small memory footprint. Other existing ORBs and object mobility toolkits do not support the handheld platforms or they have too big memory footprint, and that was the motivation behind developing our own toolkit.

To start moving objects of an application between hosts, the notion of a remote reference is required. Java does not support a remote reference of objects automatically, but it supports the notion of interfaces, which is the key of the implantation of the proxy pattern [69].

Interfaces are formal declarations of methods supported by implementation objects. Most distributed systems rely on a standard way for defining interfaces describing sets of services of an object. Object Interfaces are very similar to classes in object-oriented programming languages. Each interface consists of a set of service declarations. Each service is declared in a similar fashion as an object-oriented method; a named operation that may carry arguments, results, and exceptions. Arguments and results may consist of any arbitrary data, including control parameters, names or

references to other components. However, interfaces do not provide an implementation of an object.

Serialization is the process of taking the member data of an object and representing it as a serial stream of bytes, usually for the purpose of storing the data in a file or database. Serialization, when combined with a socket connection can also be used to transmit the state of objects from one place to another. All of these language elements taken together open the way to distributed-object computing tools.

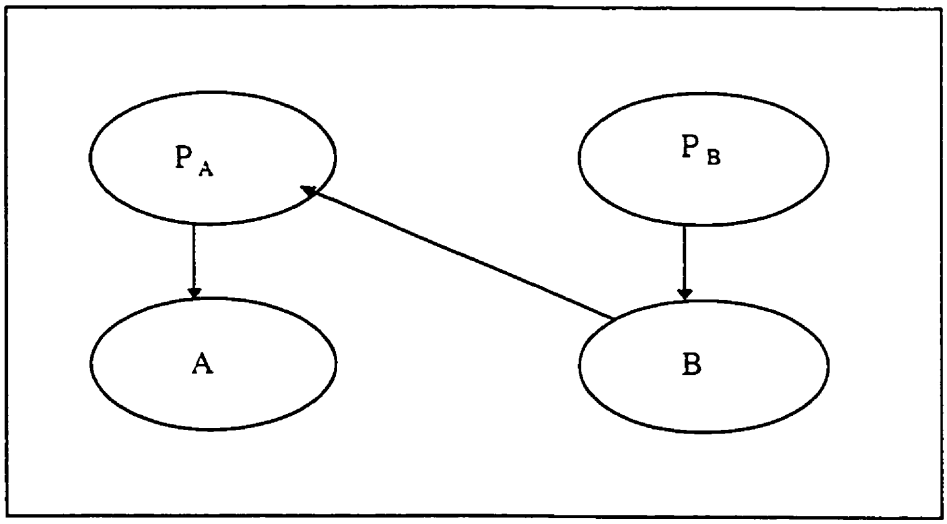

Figure 5.5: Proxy Objects with their associated Objects (P_x is a proxy of the objectX)

5.4.3 Proxy Objects

Hot swap techniques [69] are impossible to implement in Java since Java does not support pointers. To achieve a similar effect, every movable object of an application that will work on our toolkit needs to be associated with a proxy object that has the same interface as the movable object. Other objects will not reference real objects directly, but they reference them through their proxies, as Figure 5.5 illustrates, in which Object B references Proxy of Object A not Object A it self.

This will facilitate moving object without warring about changing references of other objects to it.

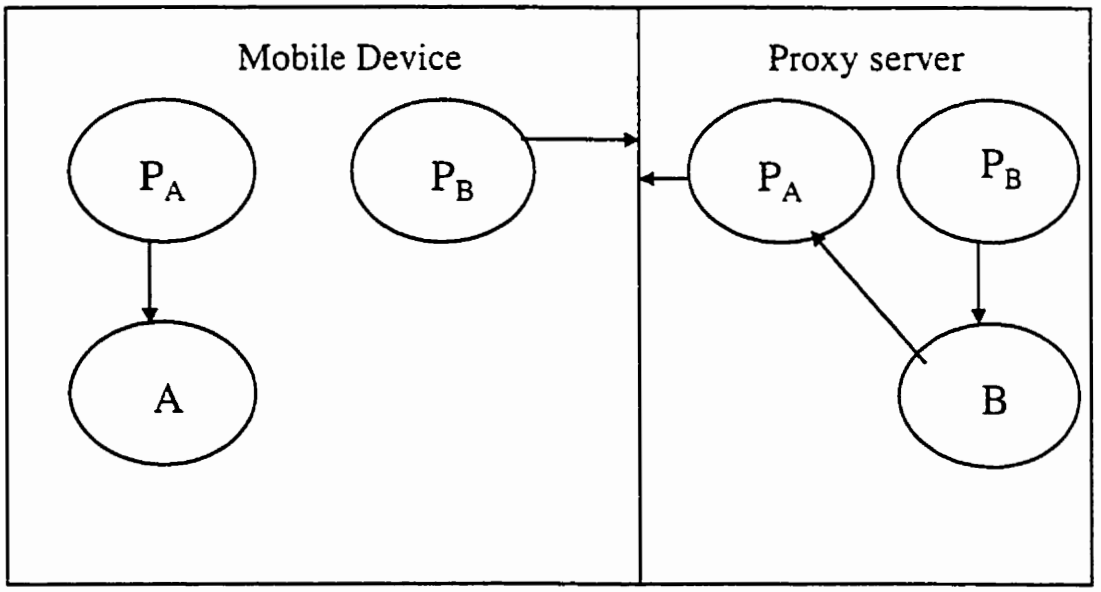

Figure 5.6: Moving Object B from Mobile Device to the Proxy Server.

To create a new class or object with an identical interface of another object in Java, in other words a proxy object, there are special automatic tools provided to create the image of another class that appears like the original to clients using it. These tools use Java reflection to inspect a class and get all information required to build a proxy object. However, the proxy object does no real work by itself. Instead, a proxy object uses network communications or delegates the communication job to other objects to create and remotely control an instance of the real object it represents. In other words, the proxy object acts as a mediator between the caller and the real object. It is through proxies that all methods calls local or remote are made.

Figure 5.6 demonstrates moving Object **B** in Figure 5.5 from mobile device to the proxy server. Moving Object **B** will not require moving Object **A** to the proxy server as well. However, at the proxy server, a proxy of the object A must be created to forward the calls to Object **A** at the mobile device. In fact, Java does offer APIs that control the serialization such that the proxy of Object **A** need not be created in this case, but deserialized only with small changes in the state of the proxy object to make it a consistent proxy object.

Figure 5.7, demonstrates moving Object **A** in Figure 5.5 to the proxy server. Moving object **A** does not require changing the reference of Object **A** in Object **B** since object **B** does reference the proxy object of **A**. Any calls from Object **B** to the Object **A** will be forwarded remotely through the proxy object of **A** at the mobile device. The

proxy object of **A** at the proxy device will allow other objects to reference the object A without effecting the flexibility of moving object **A** again to the mobile device.

Every proxy object created in the toolkit is assigned a local and a remote reference counters. These counters are updated whenever proxy object referenced locally or remotely. These counters are used to determine when the proxy and its associated object be claimed by the garbage collector.

Whenever a proxy object is not being reference remotely and locally, it will be finalized and garbage collected. If the associated object of this proxy is local, then associated object will be finalized and claimed again by the garbage collector as well. If the associated object is remote, then the proxy object will inform the remote object server to decrement the remote reference counter for the associated object at the remote site, which in turn could make the object claimed by the remote garbage collector if there is no more references locally or remotely to the associated object.

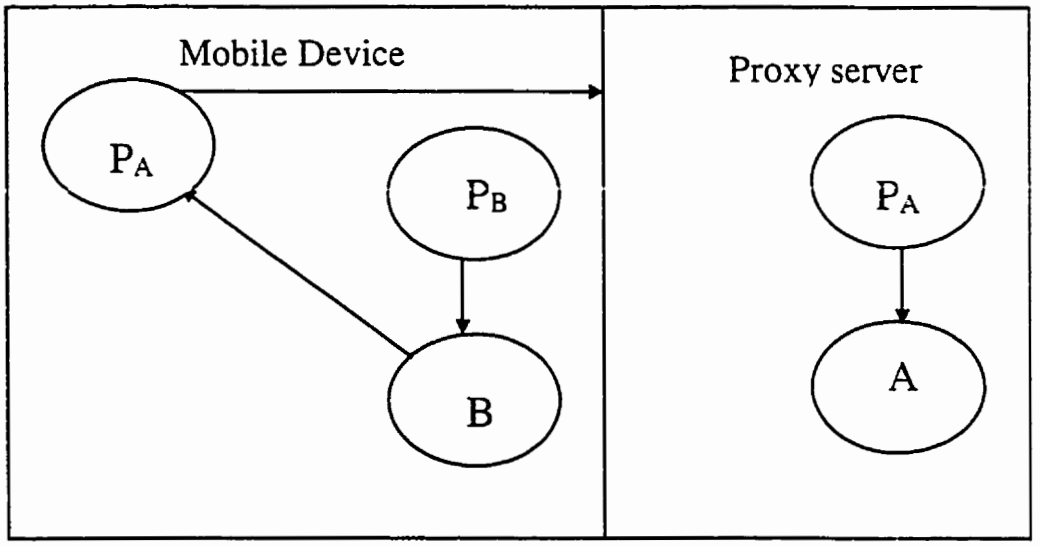

Figure 5.7: Moving Object A to the Proxy Server.

The process of creating a proxy object can be automated by using Byte Code Engineering tools [70]. These tools can create proxies of objects on the fly and use the Java Class Loaders to bring the proxies into the Java Virtual Machines. Voyager uses the same approach to automate generating proxy objects whenever it needs.

This approach is relatively slow. In the toolkit we did not automate the process of generating proxy objects due to the limitation of the PDA that we work on but not due the limitation of the Java Virtual Machine.

5.4.4 Java RMI Protocol

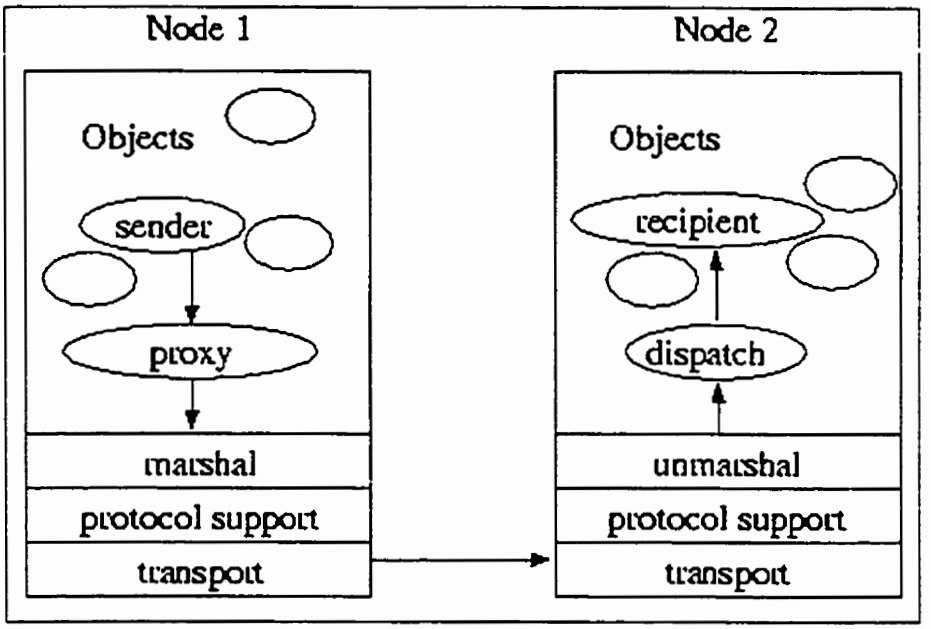

Figure 5.8: Main Structure of Remote Invocation Method Protocol.

Figure 5.8 illustrates the basic structure of invoking a method locally and remotely from a proxy object. When the toolkit is initialized at the mobile device and the proxy server, a socket connection is established during the initialization process of the toolkit. The same connection is used to send commands between virtual machines that are running on both sides. For every local object that is created, a proxy object that holds the same interface as the object is created as well and assigned a unique number that represents the associated object. This unique number will identify the associated object as long as the associated object is alive, either local or remote. When an object is to be moved to the proxy server, the proxy as well as the object both will be serialized and

shipped to the remote server, where they are deserialized and enabled for use remotely. If an associated object of a proxy object is moved, the local copy of the associated object will be finalized locally. When an object referencing a proxy is being serialized, both will be serialized, but not the associated object of the proxy being referenced.

If a method is invoked on a proxy object, the proxy will know whether the associated object is local or remote. If the associated object is local, then the proxy will forward the call to the associated object through the reflection mechanism. If the associated object is a remote object, then the proxy object will send a request to the remote server asking for execution of the remote method on the specified object. Every remote execution method request is associated with a unique number that is used to keep track of the results and the exceptions that might happen when invoking a remote method. In this request, the object identification number, method identification and method parameters are sent. Upon receiving them at the remote server, the remote server will start searching for the right proxy object to have a reference to it. If the proxy object is found, then the server will ask the proxy to invoke method. Having invoked the method, results and exceptions are sent back to the local server which in turn will dispatch them to the right method through the unique trace number mentioned before. Every proxy object will be in waiting state while waiting for the exceptions or results to be back. We do not use the polling mechanism to check for the exceptions and results, but we do use Java monitors to consume less CPU cycles and to improve the performance of the toolkit.

It is important to note that serialization can be done on any data type imaginable, from a simple integer to a height-balanced tree. Serialization makes it possible to use remote objects and their methods just as one would use local objects, with almost no restrictions in the form or structure of the data types involved.

5.4.5 Distributed Garbage Collection

Knowing and determining when objects are no longer in use is a problem in distributed computing toolkits. To deal with this, for simplicity, a reference counter based distributed garbage collection is used in our toolkit since Java does not explicitly free objects from remote memory or remote Java Virtual Machines. If an object that is not being referenced locally and remotely, then it should be finalized. This requires sending messages between the mobile device and the proxy server to keep the object reference counters updated.

5.4.6 Transportation Layer

For our experimental purposes, we build wrapper classes in Java for TCP/IP streams to control characteristic of the link. We introduce delay values between chunks of data being sent to simulate the low throughputs.

5.4.7 Toolkit Overhead

The following table shows the measured overhead of the toolkit against the Voyager toolkit. The measured values were taken on device that works on Pentium 350 MHz

Table 5.1: Measured overhead of Voyager and our toolkit

	Voyager	Our toolkit
Memory	2620 KB	204 KB
Moving object	142 ms	80 ms
Calling a method	23 ms	110 ms

From the previous table we can claim that our toolkit can be used for small portable devices such as Palms and Handheld devices. The memory requirement of our toolkit is small compared to the Voyager allowing it to be embedded in theses small devices. The cost of sending the objects compared to Voyager is still better; however, we still need to improve our remote method protocol used in our toolkit. A suggested improvement is explained in the chapter 7 and 8.

6. Java MP3 decoder

To demonstrate the feasibility of our idea, we implemented a resource intensive application, an MP3 player in Java. This chapter discusses this sample application, and we provide more details on the experiment and results in the next chapter, Chapter 7.

6.1 Introduction

Digital audio compression allows for efficient storage and transmission of audio data. There are various audio compression techniques, which offer different levels of complexity, compressed audio quality, and amount of data compression.

This chapter surveys techniques used to compress digital audio signals. This chapter starts with a summary of the basic audio digitization process and ends with the description of a sophisticated audio digital compression called MPEG layer 3, through relatively simple digital audio compression called u-law and adaptive differential pulse code modulation.

6.2 Digital Audio Data

The digital representation of audio data offers many advantages such as high noise resistance, stability and reproducibility. Also it allows the efficient implementation of many audio-processing functions such as mixing, filtering and equalization through digital computers. The conversion from analog to digital signals begins by sampling the audio input in regular intervals and quantizing the sampled values into a discrete number of evenly spaced levels. The digital audio data consists of sequences of binary values representing the number of quantizer levels for the audio sample.

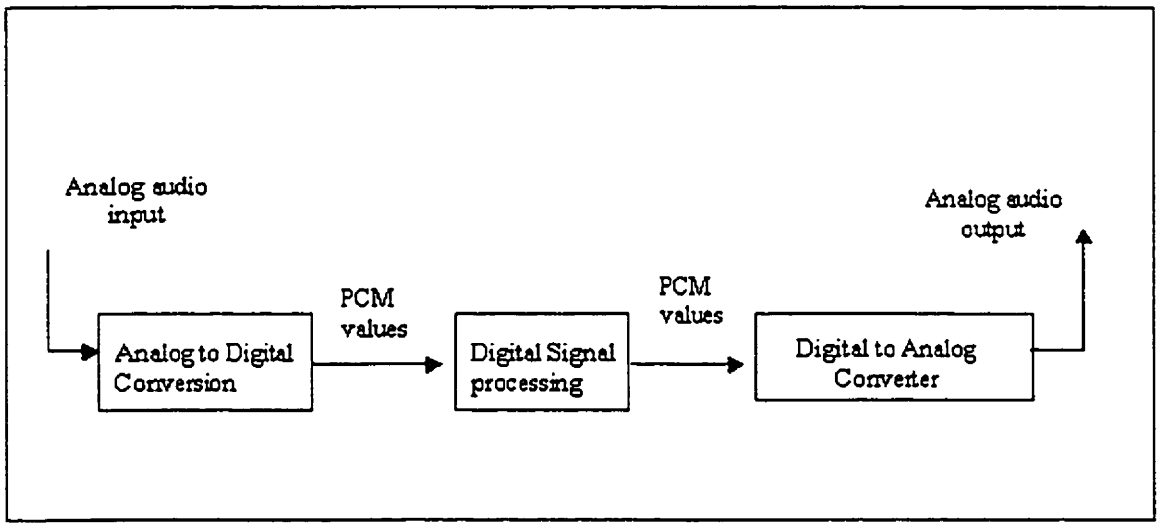

Figure 6.1: Digital Audio Process

The method of representing each sample with an independent code word is called pulse code modulation, PCM. According to Nyquist theory, a time-sampled signal can

represent up to the half of the sampling rate [71]. Typical sampling rates range from 8 KHz to 48 KHz. The 8 KHz rate covers a frequency range up to 4 KHz and provides adequate coverage for human voice. The 48 KHz rate covers a frequency range up to 24 KHz and more than adequately covers the entire audible frequencies range, which for humans, typically, extends to 20 KHz.

The number of quantizer levels is a power of 2 to make full use of a fixed number of bits per audio sample to represent the quantized values. With uniform quantizer step spacing, each additional bit has the potential of increasing the signal/noise ration by roughly 6 decibels (dB). The typical number of bits per sample used for digital audio ranges from 8 to 16 bits, which results in dB values ranging from 48 to 96 respectively. To put these values in perspective, 0 dB represents the weakest audible sound pressure level; 35 dB is the noise level inside a quite home. 125 dB is the loudest level before the discomfort starts [72].

Compared to most digital data types, data rates associated with uncompressed digital audio are substantial. The audio data on a compact disc with 2 channels of audio sampled at 44.1kHz with 16 bits per sample requires a data rate of about 1.4 Mbps.

So there is a clear need for some form of compression to enable more efficient storage and transmission of this data. There are many forms of audio compression techniques, which differ in the trade-off between the encoder and the decoder complexity, the compressed audio quality and the amount of data compression. In Section 6.3, low, medium, and high complexity techniques are presented.

6.3 Audio Compression Techniques

6.3.1 u_law Audio Compression

The u-law transformation is a basic audio compression technique. The transformation is essentially logarithmic in nature and allows the 8 bits per sample output codes to cover the dynamic range equivalent to 14 bits of linearly quantized values. This transformation offers a compression ratio of (number of bits per source sample/8) to 1. Unlike linear quantization, the logarithmic step spacing represent low amplitude audio samples with greater accuracy than higher amplitude values. This makes the signal/noise ratio of the transformed output more uniform over the range of amplitudes of the input signal. The u-law transformation is

$$y = \left\{ \begin{array}{l} 255 - \dfrac{127}{\ln(1+\mu)} * \ln(1+\mu \mid x \mid) \textit{for } \mathrm{x} \geq 0 \\ 127 - \dfrac{127}{\ln(1+\mu)} * \ln(1+\mu \mid x \mid) \text{ for } \mathrm{x} < 0 \end{array} \right\}$$

The u-law transformation is commonly used in North America and Japan for ISDN 8 KHz sampled, voice grade, digital telephony service.

6.3.2 Adaptive Differential Pulse Code Modulation (ADPCM).

The ADPCM encoder takes advantage of the fact that neighboring audio samples are generally similar to each other. Instead of representing each audio sample

independently as in PCM, the ADPCM encoder computes the difference between each audio sample and its predicted value and outputs the PCM value of the differential.

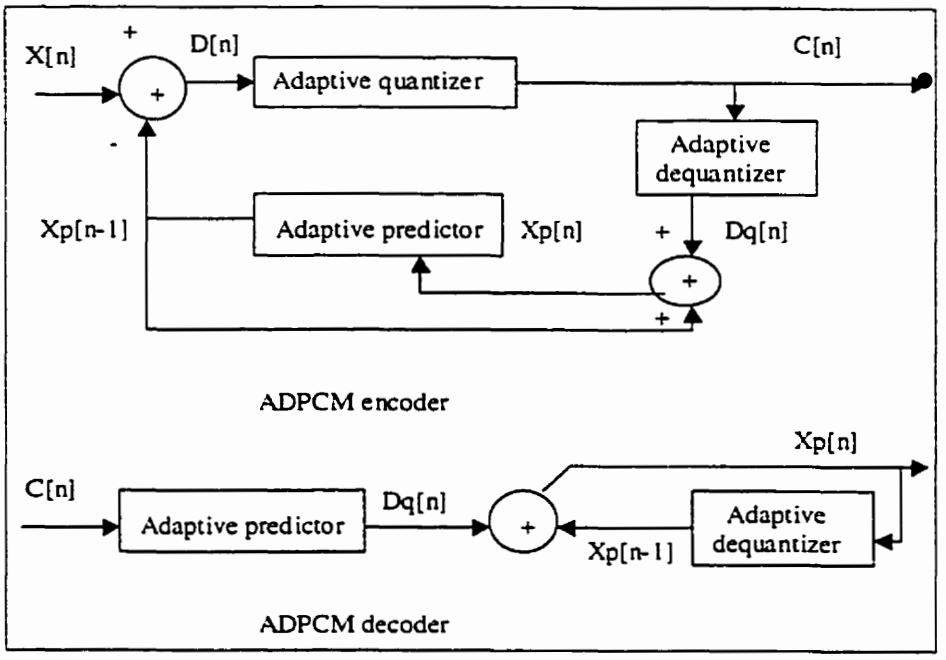

Figure 6.2: ADPCM Decoder/Encode

ADPCM coder can adapt to the characteristics of the audio signals by changing the step size of either the predictor or the quantizer or by changing both. The method of computing the predicted value and the way the predictor or the quantizer adapt to the audio signal vary among different coding systems. Some ADPCM systems require the encoder to provide side information with differential PCM values. This side information can serve two purposes. First, in some ADPCM systems, the decoder needs the additional information either to determine the predictor or quantizer step size or both. Second, the

data can provide redundant contextual information to the decoder to enable recovery from errors in the bit stream or to allow random access entry into the coded bit stream.

The ADPCM algorithm proposed by Interactive Multimedia Association offers a compression ratio of (number of bits per source sample)/4:1. The simplicity of this encoder lies in the predictor. The predictor value of the audio sample is simply the decoded value of the immediate previous audio sample. Thus the predictor block in Figure 6.2 is merely a time delay element whose output is the input delay by one audio sample interval. Since this predictor is not adaptive, side information is not necessary for the construction of the predictor.

6.3.3 MPEG/Audio Compression.

The Motion Picture Experts Group audio compression algorithm is an International Organization for Standardization (ISO) standard for high fidelity audio compression. It is one part of a three-part compression standard. With the other two, Video and Systems, the complex standard addresses the compression of synchronized video and audio at a total bit rate of roughly 1.5 Mbps.

Like u-law and ADPCM, the MPEG audio compression is lossy; however, the MPEG algorithm can achieve transparent, perceptually loss-less compression. The high performance of this compression algorithm is due to the exploitation of auditory masking. This masking is the perceptual weakness of the ear that occurs whenever the presence of a strong audio signal makes weaker audio signals imperceptible. This noise-masking phenomenon has been observed by a variety of scientists [73]. Empirical results in [73]

show that the ear has limited frequency selectivity that varies in sharpness from less than 100Hz for the lowest audible frequencies to more than 4 kHz for the highest. Thus the audible spectrum can be divided into critical bands that reflect the resolving power of the ear as a function of the frequency. The following table lists the critical bandwidths.

Table 6.1: Critical Band Boundaries

Band Number	Frequency (Hz)	Band Number	Frequency (Hz)
0	50	14	1970
1	95	15	2340
2	140	16	2720
3	235	17	3280
4	330	18	3840
5	420	19	4690
6	560	20	5440
7	660	21	6375
8	800	22	7690
9	940	23	9375
10	1125	24	11625
11	1265	25	15375
12	1500	26	20250
13	1735		

Because of the ear's limited frequency resolving power, the threshold for noise masking at any given frequency is solely dependent on the signal activity within a critical band of that frequency. Figure 6.3 illustrates this property.

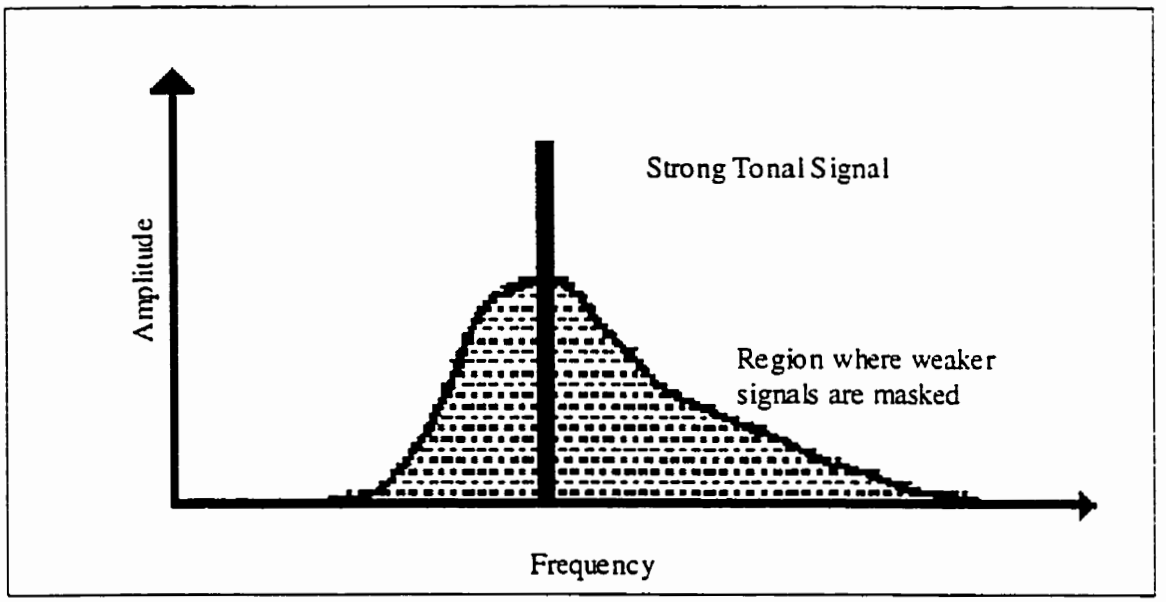

Figure 6.3:Audio noise masking

For audio compression, this property can be capitalized by transforming the audio signal into the frequency domain, then dividing the resulting spectrum into sub-bands that approximate critical bands, and finally quantizing each sub-band according to the audibility of quantization noise within that band. For optimal compression, each band should be quantized with no more levels than necessary to make the quantization noise inaudible.

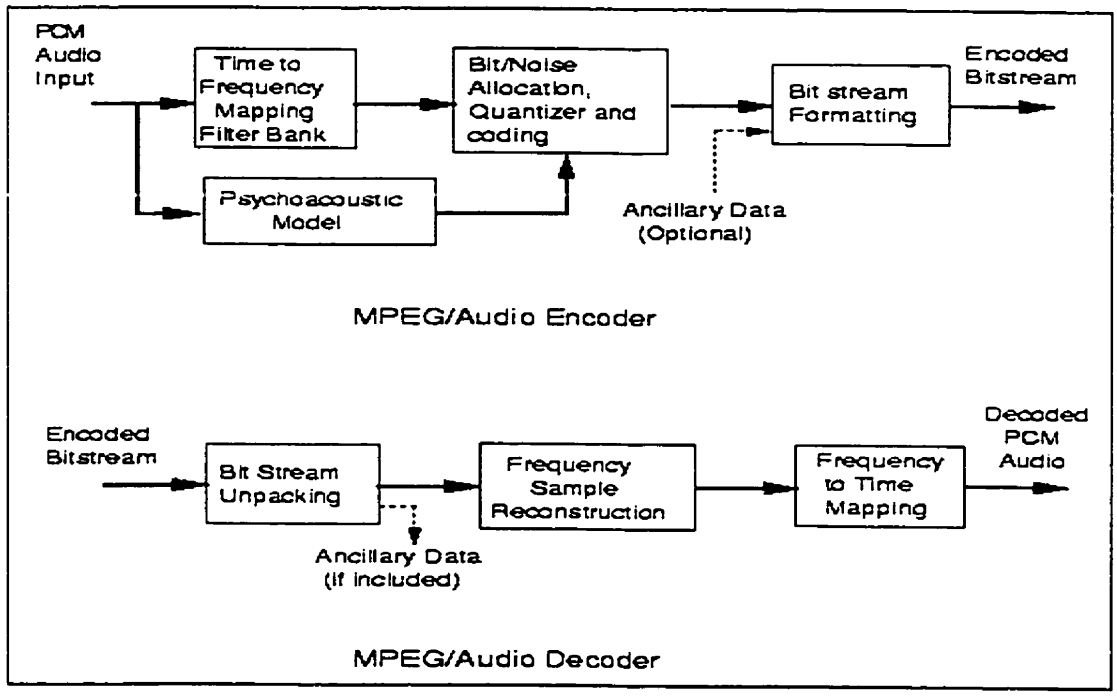

Figure 6.4: MPEG/ Audio Encode/Decoder

In the MPEG encoder/decoder diagrams of Figure 6.4 [74,75], encoding closely parallels the process described above. The input audio stream passes through a filter bank that divides the input into multiple sub-bands. The input audio stream simultaneously passes through a psychoacoustic model that determines the signal-to-mask ratio of each sub-band. The bit or noise allocation block uses the signal to mask ratios to decide how to apportion the total number of code bits available for the quantization of the sub-band signals to minimize the audio samples and formats the data into a decodable bit stream. The decoder simply reverses the formatting and constructs the quantized sub-band values, and finally transforms the set of sub-band values into a time-domain audio signal.

The MPEG/audio standard has three distinct layers for compression. Layer I forms the most basic algorithm, and Layers II and III are enhancements that use some elements found in Layer I. Each successive layer improves the compression ratio, but it increases the complexity cost of encoder and decoder.

The Layer I algorithm uses the basic filter bank found in all layers. This filter bank divides the audio signal into 32 constant-width frequency bands. The filters are relatively simple and provide good time resolution with reasonable frequency resolution relative to the perceptual properties of the human ear. The design is a compromise with three important concessions. First, the 32 constant width bands do not accurately reflect the ear's critical bands [78]. Figure 6.5 illustrates this.

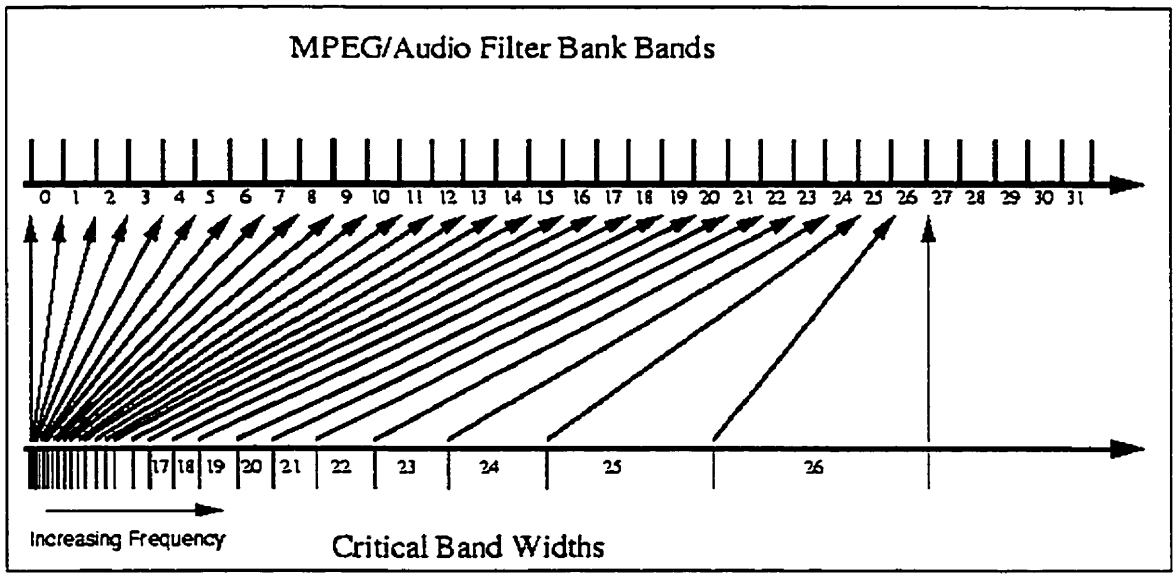

Figure 6.5: MPEG/Audio Filter bandwidths vs. Critical bandwidths.

The bandwidth is too wide for the lower frequencies so the number of quantizer bits cannot be specifically tuned for the noise sensitivity within each critical band. The included critical band with the greatest noise sensitivity dictates the number of quantization bits required for the entire filter band. Second, the filter bank and its inverse are not loss-less transformations. Even without quantization, the inverse transformation would not perfectly recover the original input signal. Fortunately, the error introduced by the filter bands has a significant frequency overlap. A signal at single frequency can affect two adjacent filter bank outputs.

The filter bank provides 32 frequency samples, one sample per band, for every 32 input audio samples. The Layer I algorithm groups together 12 samples from each the 32 bands. Each group of 12 samples receives a bit allocation and, if the bit allocation is not zero, a scale factor. Coding for stereo redundancy compression is slightly different. The bit allocation determines the number of bits used to represent each sample. The scale factor is a multiplier that sizes the samples to maximize the resolution of the quantizer. The Layer I encoder formats 32 groups of samples, 384 samples, into a frame. Besides the audio data, each frame contains a header, an optional cyclic redundancy code check word, and possible ancillary data.

The Layer II algorithm is a simple enhancement of Layer I. It improves compression performance by coding data in larger groups. The Layer II encoder forms frames of 3 by 12 by 32, 1252 samples per audio channel. In contrast, Layer I codes data in single groups of 12 samples for each sub-band, while Layer II codes data in 3 groups

of 12 samples for each sub-band. There is one bit allocation and up to three scale factors for each trio of 12 samples. The encoder encodes with a unique scale factor for each group of 12 samples only if necessary to avoid audible distortion. The Layer II algorithm also improves performance over Layer I by representing the bit allocation, the scale factor values, and the quantized samples with a more efficient code.

Layer III algorithm is a much more refined approach [74,75]. Although it is based on the same filter bank found in Layer II, Layer III compensates for some filter bank deficiencies by processing the filter outputs with Modified Discrete Cosine Transform (MDCT) and I. Figure 6.6 shows a block diagram of the process.

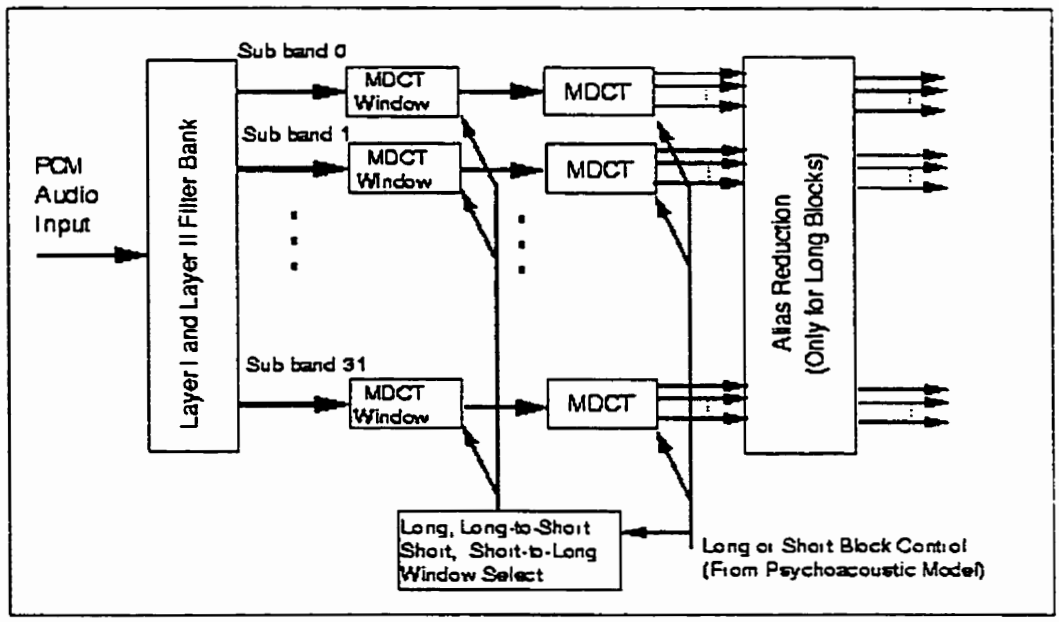

Figure 6.6: MPEG/Audio Layer III Filter Bank Processing, Encoder Side

The MDCTs further subdivide the filter bank outputs in frequency to provide better spectral resolution. Because of the inevitable trade-off between time and frequency resolution, Layer III specifies two different MDCT block lengths: a long block of 36 samples or a short block of 12. The short block length improves the time resolution to cope with transients. A long block with a specialized long-to-short or short-to-long data window provides the transition mechanism from long to short block. Layer III has three blocking modes; two modes where the outputs of the 32 filter banks can all pass through MDCTs with the same block length and a mixed block mode where the 2 lower frequency bands use long blocks and the 30 upper bands use short blocks.

Other major enhancements over Layer I and Layer II are listed as following.

- **Alias reduction:** Layer III specifies a method of processing the MDCT values to remove some redundancy caused by the overlapping bands of Layer II filter bank.

- **Non-uniform quantization:** The Layer III quantizer raises its input to the ¾ power before quantization to provide a more consistent signal to noise ratio over the range of quantizer values. The re-quantizer in the MPEG/audio decoder linearizes the values by raising its output to the $^4/_3$ powers.

- **Entropy coding of data values:** Layer III uses Huffman codes to encode the quantized samples for better data compression [76].

- **Use of a bit reservoir:** The design of the Layer III bit stream better fits the variable length nature of the compressed data. As with Layer II, Layer III processes the audio data in frames of 1152 samples. Unlike layer II, the coded data representing these samples does not necessary fit into a fixed-length frame in the code bit stream. The encoder can donate bits to or borrow bits from the reservoir when appropriate.

- **Noise allocation instead of bit allocation:** The bit allocation process used by Layer I and II only approximates the amount of noise caused by quantization to a given number of bits. The Layer III encoder uses a noise allocation iteration loop in which the quantizers are varied in an orderly way, and the resulting quantization noise is actually calculated and specifically allocated to each sub-band.

MP3 is a very powerful and popular audio format on the Internet; however, the decoder demands a lot of CPU cycles. So it is a challenge to download and play MP3 files.

6.4 Java Implementation of an MP3 Decoder

To obtain code portability, a Java version of an MP3 decoder was needed. We implemented an MP3 decoder in Java by converting C/C++ source code to Java, and we optimized it to work with our infrastructure and to demonstrate the concepts explored in this thesis. This MP3 decoder application requires a fast CPU to decode the coded sound due to the complexity of its encoder/decoder algorithm, which makes it an ideal candidate to demonstrate the need for fast static hosts, i.e. proxy servers, to support the mobile devices and PDAs that could run such type of CPU consuming applications.

6.4.1 Class Diagram of The Java MP3 Player

Figure 6.7 shows the class diagram of the Java MP3 player. Which highlights the architecture of the decoder basically. What is important for our thesis is to identify and represent the instances (objects) into an object graph. This object graph consists of nodes that represent instances of classes with CPU time consumed to achieve their functionality. Also, this graph consists of edges that represent the cost of method calls and the data volume being transferred between nodes.

The CPU time and the edges cost can be deduced from the call graph that is provided by a toolkit called JProbe [77]. This toolkit is basically an instrumented JVM that monitors an application objects and the method calls between objects. Figure 6.8 shows the instances of the classes used to decode an MP3 song. Tables 6.2 and 6.3

contain profiling information regarding nodes and edges in the object graph of the MP3 decoder, which is required as input to the GGP algorithm in Chapter 7. This profiling information is measured on a device that runs on 350 MHz Pentium.

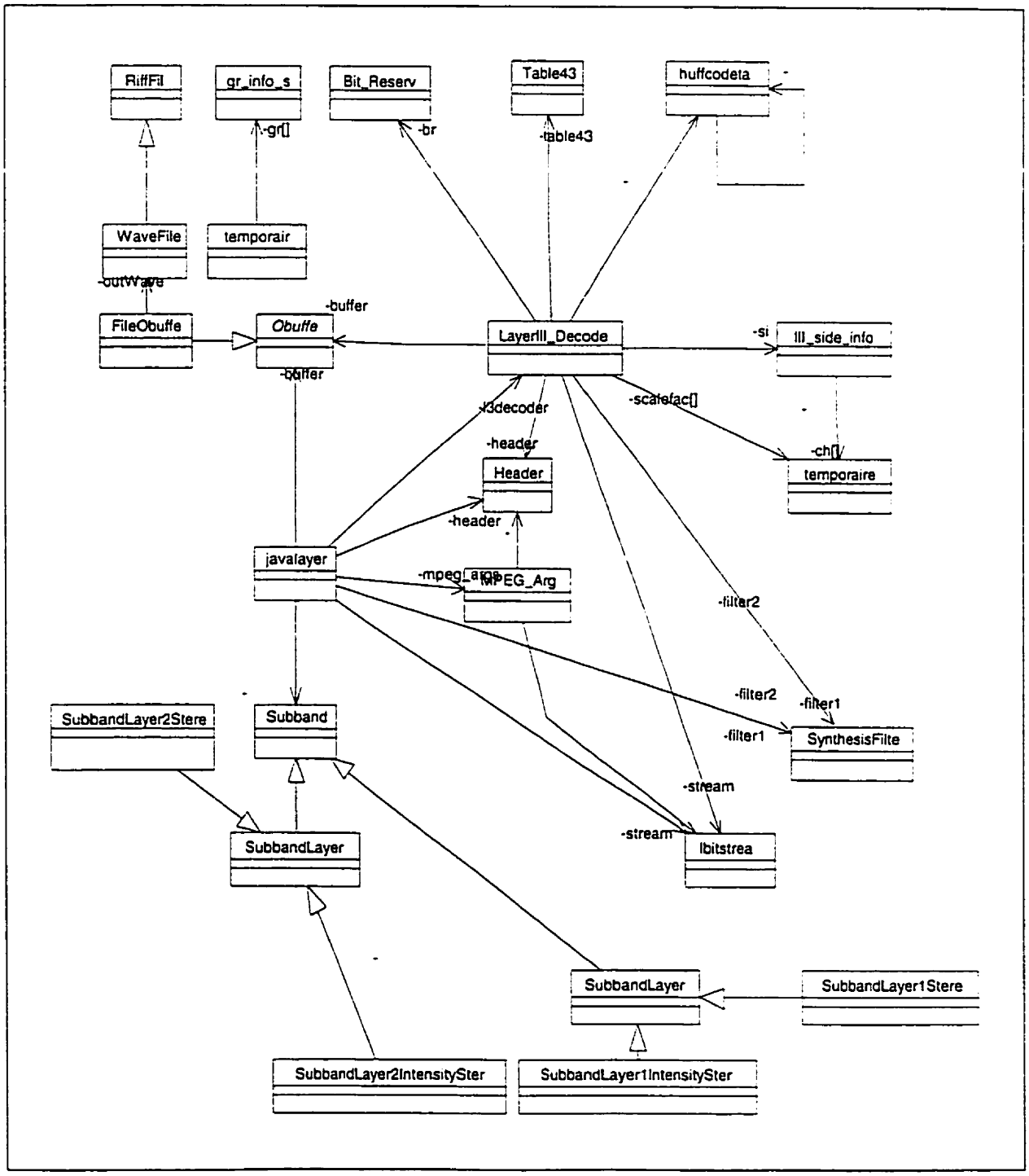

Figure 6.7: Class Diagram for Java MP3 Decoder

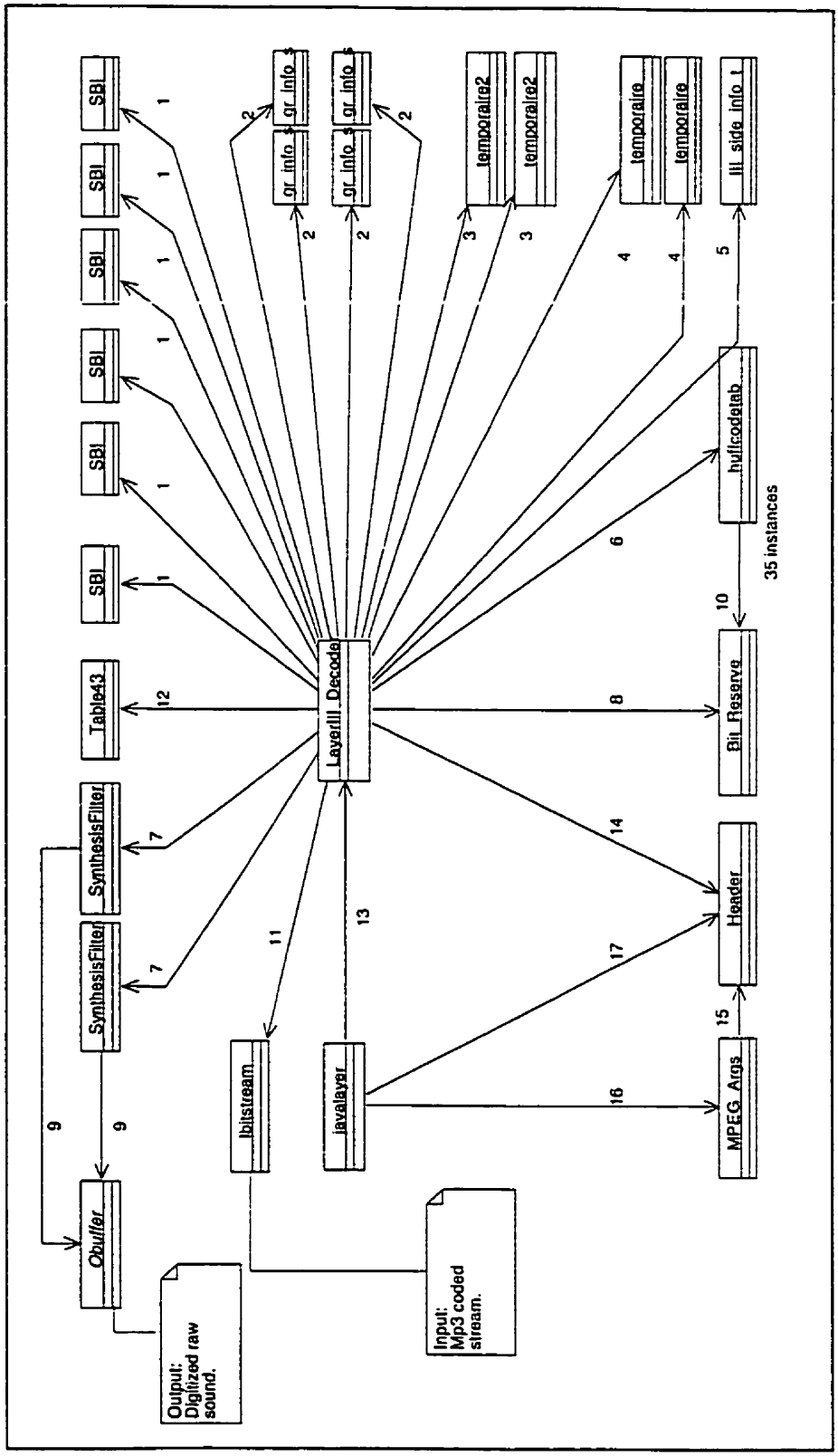

Figure 6.8: Object graph of the MP3 decoder.

Table 6.2: Objects sizes (in bytes) and the average CPU time in milliseconds for decoding 1 frame of an MP3 song on PentiumII 350 MHz.

Object Name	Object Size	Instances	Code Size	Calls/Frame	Avg CPU Method Time	Avg CPU time/class	Avg CPU Time/instance
Table43	28	1	107344	620	1.28169E-05	0.00794	0.00794
Bit_Reserve	16666	1	1430	3355	0.000149	0.50058	0.50058
SBI	223	6	2905	97	0.001191	0.11560	0.01926
gr_info_s	376	4	5195	7184	3.46479E-05	0.24891	0.06222
temporaire2	376	2	1409	1687	5.28169E-06	0.00891	0.00445
Temporaire	593	2	1819	52	0.008529	0.44351	0.22175
Header	765	1	9245	11	0.000242	0.00267	0.00267
III_side_info_t	959	1	2022	35	0.004775	0.16715	0.16715
Ibitstream	1972	1	5301	449	0.000649	0.29177	0.29177
huffcodetab	2526	35	45493	693	0.005722	3.96590	0.11331
SynthesisFilter	4414	2	18724	4824	0.000185	0.89451	0.44725
LayerIII_Decoder	25114	1	47146	160	0.00669	1.07052	1.07052

Table 6.3: Objects sizes (in bytes) and the average CPU time in milliseconds for decoding 1 frame of an MP3 song on Pentium 133 MHz.

Object Name	Object Size	Instances	Code Size	Calls/Frame	Avg CPU Method Time	Avg CPU Time/class	Avg CPU Time/instance
Table43	28	1	107344	620	6.16493E-05	0.03822	0.03822
Bit_Reserve	16666	1	1430	3355	0.000717	2.40782	2.40782
SBI	223	6	2905	97	0.005732	0.55606	0.09267
gr_info_s	376	4	5195	7184	3.46479E-05	0.24891	0.06222
temporaire2	376	2	1409	1687	2.54049E-05	0.04285	0.02142
Temporaire	593	2	1819	52	0.041025	2.13332	1.06666
Header	765	1	9245	11	0.000242	0.00267	0.00267
III_side_info_t	959	1	2022	35	0.022972	0.80402	0.80402
Ibitstream	1972	1	5301	449	0.000649	0.29177	0.29177
Huffcodetab	2526	35	45493	693	0.027527	19.07631	0.54503
SynthesisFilter	4414	2	18724	4824	0.000185	0.89451	0.44725
LayerIII_Decoder	25114	1	47146	160	0.032182	5.14924	5.14924

Table 6.4: Objects sizes (in bytes) and the average CPU time in milliseconds for decoding 1 frame of an MP3 song on handheld device.

Object Name	Object Size	Instances	CodeSize	Calls/Frame	Avg CPU Method Time	Avg CPU time/class	Avg CPU Time/instance
Table43	28	1	107344	620	0.001486	0.92179	0.92179
Bit_Reserve	16666	1	1430	3355	0.017307	58.0680	58.0681
SBI	223	6	2905	97	0.138250	13.4103	2.23505
gr_info_s	376	4	5195	7184	0.004019	28.8736	7.21840
Temporaire2	376	2	1409	1687	0.000612	01.0336	0.51679
Temporaire	593	2	1819	52	0.989387	51.4482	25.7241
Header	765	1	9245	11	0.028170	0.30987	0.30987
III_side_info_t	959	1	2022	35	0.554004	19.3902	19.3902
Ibitstream	1972	1	5301	449	0.075379	33.8454	33.8454
huffcodetab	2526	35	45493	693	0.663857	460.053	13.1443
SynthesisFilter	4414	2	18724	4824	0.021509	103.7642	51.8821
LayerIII_Decoder	25114	1	47146	160	0.776133	124.1814	124.1814

Table 6.5: Edge weights of the object graph in Figure 6.8 of the MP3 decoder.

Link	Number of calls/Frame	Avg number of calls/instance/frame	DataIN	DataOut	TotalDataIn	TotalDataOut	Total /instance
1	97	16.2	44	44	711.3	711.3	2231
2	7192	1798	42	42	75516	75516	240932
3	429	214.5	50	46	10725	9867	31317
4	48	24	42	42	1008	1008	3216
5	10	10	43	42	430	420	1350
6	692	19.8	52	48	1028.114	949.0285	2965.714
7	144	72	42	40	3024	2880	9504
8	587	587	43	43	25241	25241	79832
9	1	1	580	0	580	0	630
10	2768	79.09	42	44	3321.6	3479.771	10755.6
11	1	1	0	417	0	417	467
12	619	619	44	44	27236	27236	85422
13	1	1	64	40	64	40	154
14	14	14	40	44	560	616	1876

6.4.2 Performance experiment

To measure the performance of the MP3 player, we executed it on two PC devices with Intel CPU architecture. The first is a workstation, which runs at 350 MHz PentiumII and the second is a laptop, which runs at 133 MHz Pentium processors. The results were taken when the only application running on both devices was the Java MP3 player. We define performance index of the decoder by the percentage ratio of the decoding and playing time

Figure 6.9 shows that the MP3 decoder requires relatively high CPU speed to start playing the decoded sound in real-time. In Figure 6.9, the performance index of less than 100% indicates that the decoder cannot produce enough decoded sound to play it in real time; however, it is possible to use buffer mechanisms to buffer decoded sound before playing it, and the buffer size depends on the performance index value.

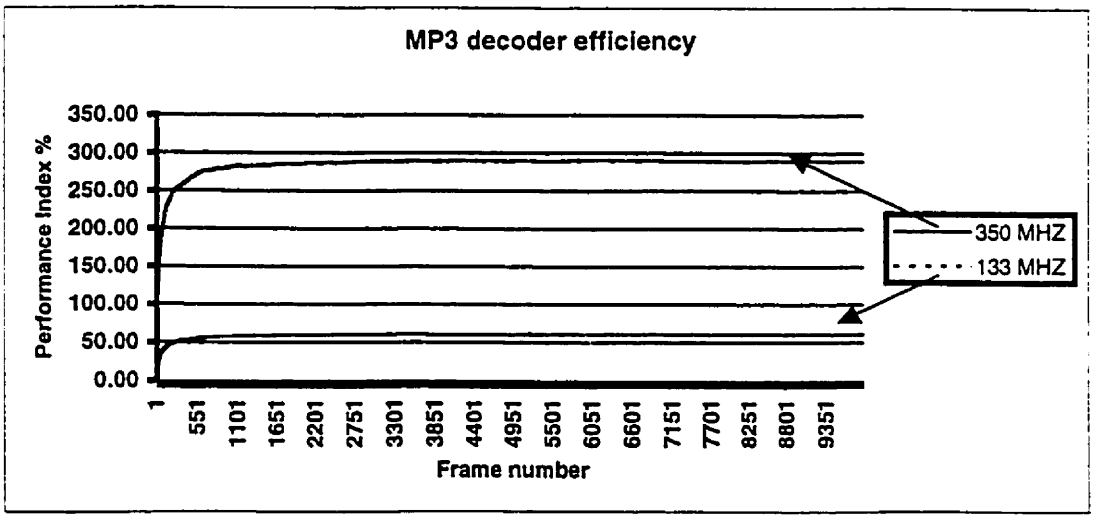

Figure 6.9: The measured performance of MP3 decoder on two different CPU speeds.

Performance index of values greater than or equal to than 100% indicate that decoder can play decoded sound in real time. However, buffering is needed to not overwhelm the playing rate, and again the buffer size depends on the performance index.

The MP3 decoder requires approximately 38 frames to decode stereo sound for 1 second. The frame size is 417 bytes. The decoder produces sound with play rate of 44100 Hz for both channels per second, which we convert to mono and sample down the playing rate to 11025 Hz per second so that it can be played on both laptop and PDAs devices we use. The required bytes to present a second of a mono channel with sampling rate of 11025 Hz and 16 bits to present a sample are 22050 bytes.

7. Experiments and Results

Our solution is based on Greedy Graph Partitioning algorithm to be used for load sharing purposes, in which an object graph is partitioned into clusters of logically and strongly related objects. There are two types of clustering algorithms: **sequence-based clustering techniques**, which transform the object graph into a linear sequence of objects which are segmented from left to right into partitions, and **partition-based clustering techniques** which transform the object into clusters of objects. The partition-based techniques can be classified into two categories:

- ***Constructive algorithms*** build a partition from scratch.
- ***Iterative algorithms*** starts with some initial partitioning and repeatedly try to improve this partitioning.

Under sequence based clustering, the cluster graph is transformed into a linear sequence of objects, which is then sequentially assigned to clusters. Under partition based clustering, the cluster graph is partitioned into object partitions. The goal is to minimize the total weight of those edges of the cluster graph that cross page boundaries.

The first technique, constructive partition-based, is of more interest for us since it meets our requirement for a partitioning algorithm. They produce better clustering quality with reasonable runtime compared to the iterative algorithms [79].

The clustering problem is closely related to the graph-partitioning problem where some graph is to be partitioned into several disconnected (partitions). The object graph (OG) is constructed considering objects as vertices and the inter-object references as directed edges. Clustering algorithms partition the OG by assigning objects to equally sized pages. Instead of the OG, clustering algorithms often use a more specific graph as input that is derived from the OG and/or from information about the applications' access behavior. The derived graph is called the clustering graph (CG). The vertices and edges of the CG are labeled with weights.

The set of edges of the CG may be a multi-set. However, in order to simplify the problem, we transform the CG into a simple graph, i.e., every edge occurs only once, by accumulating the weights of edges between the same start and terminal vertices. For a given partitioning of the CG, the total weight of all edges crossing partition borders (page borders) is the **external costs** of this partitioning.

The clustering problem is to find a partitioning of the CG such that the size of each partition, i.e., the total size of its objects, is less or equal the page size and the external costs are minimized.

It is far more expensive to dynamically gather information on the access behavior between objects since it requires monitoring applications objects, which has a none-negligible overhead especially in the slow and not optimized JVMs on PDAs. In the reminder sections we assume the weights of the edges and vertices weights of the object graph to be given either by dynamic or static access analysis.

Our approach consists of two steps. The first step is based on the partitioning algorithm in which we divide the object graph of an application into partitions that would give the minimum weight of edges that cross the partitions boundaries. The second step determines whether moving one of these partitions to the proxy server would be beneficial.

7.1 The Greedy Graph Partitioning Heuristics (GGP)

Because of the very good clustering results but poor runtime performance of known partition-based clustering algorithms, we have chosen a newly developed partitioning algorithm called Greedy Graph Partitioning Heuristics. The algorithm was first proposed in [79]. It is strongly related to the subset optimization problem for which greedy algorithms often find good solutions very efficiently.

The input for the algorithm is an object graph, which consists of the weighted edges and vertices. The output of the graph is a list of partitions, which minimize external cost between the connected partitions.

7.2 Simple GGP Algorithm

The GGP algorithm is based on a simple greedy heuristic that was developed for computing the minimum-weight spanning tree of a graph. First all partitions are inhabited by a single object, and all partitions are inserted into a ***PartList***. For all objects O_1, O_2 connected by some edge in the CG with weight $W_{o1, o2}$ a tuple (O_1, O_2, $W_{o1, o2}$) is inserted into the list ***EdgeList***. All tuples of ***EdgeList*** are visited in the order of descending weights. Let (O_1, O_2, $W_{o1, o2}$) be the current tuple. Let P_1, P_2 be the partitions to which object O_1 and O_2 belong. If P1! = P2 and if the total size of all objects assigned to P_1 and P_2 is less than the page size, the two portions are joined. Otherwise, the edge is merely discarded and the partitions remain invariant. Figure 7.1 shows the GGP algorithm outline.

Let **E** be the number of edges of the object graph, then the runtime complexity of the algorithm is **O (E log E)** since the dominating factor in the algorithm is the Sorting algorithm. The runtime complexity of the algorithm can be reduced to **O(E)** if the sorting algorithm is eliminated. To eliminate the sorting algorithm the edges list has to be sorted during the static analysis.

- **INPUT:** The object graph;
- **OUTPUT:** A list of partitions;
- Let **PageSize** = Maximum number of object is a cluster;
- **PartList**: = Empty List;
- Assign each **object** in the object graph to a new **partition** and insert this partition into **PartList.**
- Let **EdgeList** be a list of tuples of the form (O_1, O_2, Wo_1o_2), where Wo_1o_2 is the total weight of all edges between O_1 and O_2.
- **If (Dynamic analysis is used) then Sort Edge List by descending weights;**
- **Foreach** (O_1, O_2, Wo_1o_2) in Edge List **do**
- **Begin**
    - **Let P_1, P_2** be the partitions containing objects O_1, O_2;
    - **If(P_1 and P_2 are movable partitions) then**
    - **Begin**
        - **If (P_1 != $P_2$and the total size of all objects in P_1 and P_2 is less than the Page Size** then
        - **Begin**
        - Move all **objects** from **P_2 and P_1**
        - Remove **P_2** from **Part List**;
        - **End if;**
    - **End if;**
- **End foreach;**

Figure 7.1: Outline of simple GGP Algorithm.

This algorithm is not optimal. During each iteration, the GGP algorithm takes the edge with maximum weight from the EdgeList and tries to join the partitions of the objects incident to that edge. However, this is not necessarily the best decision. The following example illustrates the weakness of the algorithm, and how it can be improved.

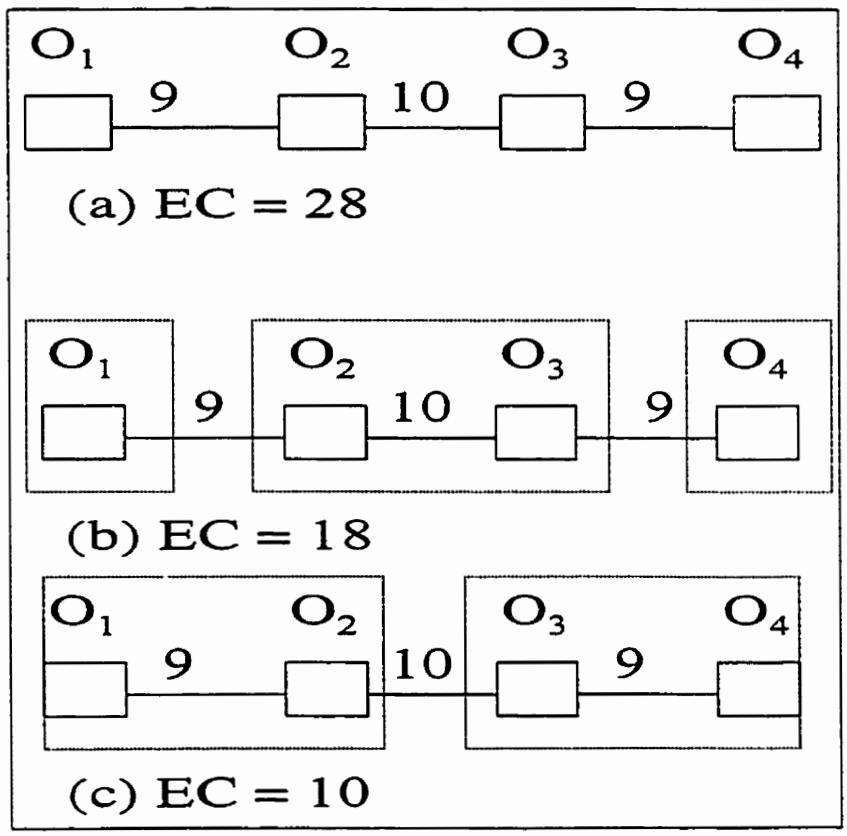

Figure 7.2: An example of Non-optimal GGP Clustering based on Page Size = 2.

With the assumption that objects sizes are uniform and maximum page capacity is 2 objects, Figure 7.2 (a) shows the maximum external costs when no partitions are considered. Figure 7.2 (b) shows the external costs when the simple GGP algorithm is used. Figure 7.2 (c) shows the optimal case that a clustering algorithm should give with the object graph in Figure 7.2 (a) is as input.

The improvement to the simple GGP algorithm is represented in the idea of bounded look-ahead. The bounded look-ahead is to detect situations where it is advantageous to reject the current edge, i.e., the edge with maximum weight, and to consider other edges first. A full description of the algorithm and quantitative analysis of the algorithm is described in [79]. Our object graph does not contain situations where the look-ahead feature added to the simple algorithm produces better results. Thus using the simple algorithm or using the new algorithm will result in the same partition.

7.3 Experiment

We run the simple GGP algorithm with the object graph representing object graph of the MP3 player. We change the following parameters to demonstrate their importance. The parameters are:

- ***Bandwidth***
- ***Relative CPU speeds (Mobile CPU: Proxy CPU)***
- ***Cluster Size or Page size of the simple GGP.***

We varied the bandwidth, relative CPU speed, and cluster size. To figure out which partitions are most beneficial to be moved, we assume that object weights are uniform. This is because at any time shipping object will happen once only; but calling the shipped objects from the mobile device over the wireless link could happen more than once. Thus our concern is more on the number of calls and the data volume being moved between mobile device and the proxy server rather than object sizes during execution of the application. However, partition weights, the sum of object weights in the partition, are still used to determine an estimate of the response time and the power consumption cost when that partition moved to the proxy.

Fixing the bandwidth, the relative CPU speed, and varying page size N, where N is number of objects in object graph, we run the simple GGP algorithm to obtain a number of partitions each of which contains at most N objects. Which of these partitions is helpful in increasing performance or decreasing power consumption if shipped is determined through calculations and the user preference. These calculations include the partition weight, total CPU time consumed by this partition, and the total edges weight emerging from the partition. We ship only one partition at a time. Selecting two partitions is not an appropriate choice since their objects are not strongly related to each other. Otherwise, since page size is being varied, those objects will eventually be combined together in one page if they are indeed strongly related.

To simplify the calculation, first we represent the fixed resources, such as an Internet server, in object graph as dummy object with no CPU time and weight in the object graph. However, if there is an edge between the dummy object and any other object in the object graph, then the edge will represent traffic volume between the dummy object and the other objects. Since the dummy object has no CPU time and weight, the edge weight is the only factor that will be considered in the decision process. Any edge that has one of its end nodes as a dummy object can hold a positive or negative value, a positive value means that the dummy object is fixed at the mobile device and a negative value means that the dummy object is fixed at a remote site. Figure 7.3 shows this.

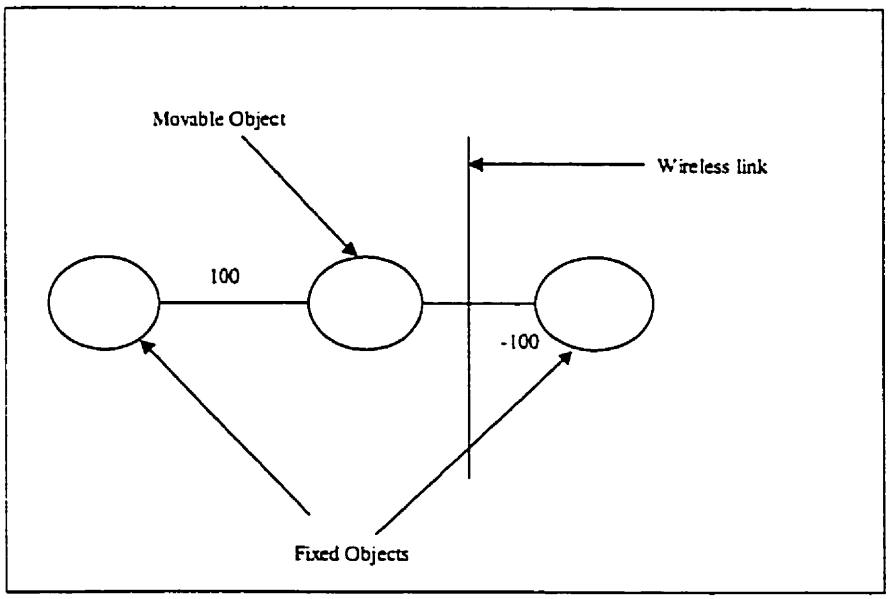

Figure 7.3: Representing fixed resources as dummy objects in object graph.

In the object graph in Figure 6.8, we substitute two objects with dummy objects: ***Ibitstream*** and ***Obuffer***. ***Ibitstream*** represent the source of the coded audio, and ***Obuffer*** represents the decoded audio. **Obuffer** is always fixed at the mobile device. Second, since the input for the simple GGP algorithm requires an undirected graph, we add the directed edges between any two objects to represent the total traffic between objects. Since sending and receiving costs of Wireless cards are different, the sending cost is higher than the receiving cost; we use the weighted average, based on volume of data in end directions, in our calculations with an undirected graph.

Our goal is to obtain accurate results; however, through the experiments, we noticed that there is a lot of overhead introduced by the object mobility toolkit discussed in Chapter 5. We compared the response time of invoking a method in a remote object once using our toolkit and using the Voyager toolkit. The method used in this comparison does not take parameters and does not return values; however, it does throw an exception. Voyager takes about 23 ms on average and our toolkit takes 110 ms on average to invoke the method remotely. There are two reasons for Voyager's superior performance in this case. First, the communication protocol used in implementing RMI introduces fewer overheads for the same amount of data being sent. The second is that Voyager uses native interface as well as native processes for the object servers, which improves the performance of the toolkit; however, this prevents Voyager from being portable across platforms.

In our toolkit, we do not use native interface to process Java objects, but we use native interface to export power consumption events. The RMI protocol being used in our toolkit is object based. Any command being sent to Remote Object Server is a serialized Java object that holds information about the target object, the method and its parameters. The Remote Object Server de-serializes the command into Java Object and ultimately will invoke the method on the object. This simplifies the implementation of the toolkit; however, due to the serialization protocol overhead, the ratio of the overhead to data is high. To put this in perspective, Table 7.1 shows the overhead associated with serializing objects, which argues for improvement of the RMI and/or serialization protocols specification such that it can be used more efficiently over wireless links. It is worth to notice that **Object Size** of a primitive type wrapper can be calculated as following:

Object Size = 60 + ***Primitive data size + wrapper full class name length.***

For example, to serialize a Long object, the required data size is 82 bytes.

Table 7.1: The Ratio of Object Size to Java Primitive Types.

Data Type	Data size	Object size when serialized
Byte	1	75
Short	2	77
Integer	4	81
Float	4	79
Double	8	84

In the following graphs and tables, which are based on Table 6.2 to Table 6.5 and Figure 6.8, the following legends are used to indicate the various mathematical modes being used.

- **PN** (Partition Number) is an index that represents the group of objects
- **NP** (Number of Partitions) The number of partitions generated when running the simple GGP with a specific page size.
- **EC** (External Cost) is total weight of all edges that cross-generated partitions in bytes.
- **TCPUT** (Total CPU Time) is the total CPU time of all nodes of the object graph in mille-seconds.
- **PW** (Partition Weight) is a partition weight in bytes i.e. size of the object if serialized in bytes.
- **LTRT** (Local Total Response Time) is the response time when all objects are executed locally, equation (7.1).
- **PCPUT** (Partition CPU Time) is the CPU time that is consumed by the partition in mille-seconds. In other words it is the total CPU time of all objects in the partition.
- **PEW** (Partition Edges Weight) is total weight of edges that emerge from the partition in bytes not including negative edges.

- **PNEW** (Partition Negative Edges Weight) is the total weight of negative edges that emerge from a partition in bytes.

- **RTPR (NFD)** (Response Time when Partition is at the Remote Site) is the total response time when a partition is moved remotely. It is a function of how many times the partition is being used remotely, NFD. The time is in mille-seconds, equation (7.2).
- **LTPC** (Local Total Power Consumption) is the total power consumption of the entire object graph being executed locally. The measuring unit is watts/hour, equation (7.3).
- **PCPR (NFD)** (Power Consumption when a Partition is at the Remote site) is the power consumption at the mobile device when a partition is moved to the proxy and being executed NFD times, equation (7.4).
- **RTIP** (Response Time Improvement Percentage) is the percentages gained in response time if a specific partition was executed remotely, equation (7.5).
- **PCIP** (Power Consumption Improvement Percentage) is the percentage of the power consumption reduction in the mobile device, equation (7.6).
- **PS** (Partition Size) is the number of objects in a specific partition.
- **BW (Bandwidth/Throughput)** is the available through put for the partitions to move through in kilobytes.
- **RCPUS** (Relative CPU Speeds) is the relative CPU speed between the mobile device and the proxy server.

- **SNEWS** (Summation of Negative Edges Weights in Systems) is the summation of the total negative weights in the entire system in bytes.
- **NFD** (Number of Frames Being decoded).
- **LPerformance** (Local Performance) is the player performance index value when decoding locally.
- **RPerformance** (Remote Performance) is the player performance index value when decoding remotely.

$$LTRT = TCPT + \left(\frac{|SNEWS * 8|}{BW} \right)$$

Equation (7.1)

$$RTPR(NDF) = NDF \times \left((TCPT - PCPUT) + \left(\frac{PCPUT}{RCPUS} \right) \right) + \left(NDF \times \left(\left(\frac{PEW \times 8 + (|SNEWS| + PNEW) * 8}{BW} \right) \right) \right) + \left(\left(\frac{PW \times 8}{BW} \right) \right)$$

Equation (7.2)

$$LTPC = 1.8 \times \left(\left(\frac{TCPT/1000}{3600} \right) \right) + 2.35 \times \left(\frac{\left(\frac{|SNEWS|}{BW \times 1000} \right)}{3600} \right)$$

Equation (7.3)

$$PCPR(NDF) = NDF \times \left(1.8 \times \left(\frac{TCPT - PCPUT}{1000 \times 3600} \right) \right) + 2.35 \times \left(NDF \times \left(\left(\frac{PEW \times 8}{PW \times 1000 \times 3600} \right) + \left(\frac{(|SNEWS| + PNEW) \times 8}{BW \times 1000 \times 3600} \right) \right) + \frac{PW \times 8}{BW \times 1000 \times 3600} \right)$$

Equation (7.4)

$$RTIP = \frac{LTRT \times NDF - RTPR(NDF)}{LTRT \times NDF} \%$$

Equation (7.5)

$$PCIP = \frac{LTPC \times NDF - PCPR(NDF)}{LTPC \times NDF} \%$$

Equation (7.6)

We use the previous equations to calculate an estimated response time as well as the power consumption costs, where the value 2.35 is the average power consumption of sending and receiving data over a wireless link using Table 4.1

Running the simple GGP algorithm with N objects of which M are static objects results in N-M possible clusters; however, we choose only some of these for discussion purposes.

7.4 Results

To explore the effect of the following parameters we varied them and observed the efficiency of the player.

1. ***Bandwidth available.***
2. ***Relative CPU speeds (Mobile CPU: Proxy CPU).***
3. ***Cluster Size or Page size of the simple GGP.***

To observer the importance of the first parameter, the bandwidth available, we choose certain low and high bandwidths. For PDA as a client, we selected low bandwidths; we did choose 19.2 Kb/sec to represent CDPD. For high bandwidths we did choose 1000 Kb/sec to represent the set of bandwidths that can be obtained from Wireless Ethernet cards.

To observer the importance of the second parameter, the relative CPU speeds, we did fix the bandwidth to 1000 Kb/sec so that the bandwidth does not become a bottleneck

between the mobile device and the proxy device. The third parameter is varied from 1 to N-M while varying the other parameters. This is because of the nature of the simple GGP algorithm. The previous experiments are run for a PDA and a laptop as client devices. The PDA runs on a RSIC processor of 75 MHz and the laptop runs on a Pentium processor of 133 MHz. The proxy server runs on 350 MHz Pentium II.

The performance of Java Applications depends primarily on the performance of the JVM. Since both laptop and the proxy server run relatively high performance JVMs using JIT compilers, the relative Java application performance on both CPUs is maintained. However, JVM on the PDA we use is very slow and does not support JIT compilers, the relative CPU speed degrade considerably. We measured the relative CPU speed between the PDA, laptop and the proxy and found to be 1:116 and 1:4, respectively. Therefore, while varying bandwidth, the relative CPU speeds are fixed to 1:2 and 1:116 for PDA as a client, and 1:4 for laptop as client.

In the following section, Local Performance and Remote Performance factors are based on decoding 38 MP3 coded audio frames, with the assumption that output is mono, with sampling rate of 11025, and 16 bits per sample.

Table 7.2 An estimation of the Response Time and Power Consumption for PDA as Client with BW = 19.2kb/s and RCPUS = 2.

BW = 19.2Kb/s		RCPUS = 2						
CS	NP		PN	PS	RTIP	PCIP	LPerformance	RPerformance
1	58	MIN	56	1	-51597.91	-63768.99	2.4923	4.821E-05
		MAX	19	1	-53.35185	-65.343	2.4923	1.6252
20	39	MIN	30	20	-14694.8	-18150.99	2.4923	0.0168
		MAX	19	1	-53.35185	-65.34	2.4923	1.6252
30	29	MIN	40	30	-9299.763	-11478.89	2.4923	0.0265
		MAX	19	1	-53.35185	-65.34	2.4923	1.6252
56	3	MAX	54	56	-116.7979	-117.54	2.4923	1.1496

Table 7.3 An estimation of the Response Time and Power Consumption for PDA as Client with BW = 19.2kb/s and RCPUS = 116.

BW = 19.2Kb/s		RCPUS = 116						
CS	NP		PN	PS	RTIP	PCIP	LPerformance	RPerformance
1	58	MIN	56	1	-51592.136	-63768.99	2.4923	0.0048
		MAX	19	1	-52.450	-65.34	2.4923	1.6349
20	39	MIN	30	20	-14678.134	-18150.99	2.4923	0.0169
		MAX	19	1	-52.44947	-65.34	2.4923	1.6349
30	29	MIN	40	30	-9277.01	-11478.89	2.4923	0.0266
		MAX	19	1	-52.449	-65.34	2.4923	1.6349
56	3	MAX	54	56	-76.715	-117.54	2.4923	1.4104

Table 7.4 An estimation of the Response Time and Power Consumption for PDA as Client with BW = 1000kb/s and RCPUS = 2.

BW	1000Kb/s	RCPUS	2					
CS	NP		PN	PS	RTIP	PCIP	LPerformance	RPerformance
1	58	MIN	56	1	-1202.2133	-1562.5096	3.0422	0.2336
		MAX	19	1	-0.1511	0.5803	3.0422	3.0376
20	39	MIN	30	20	-324.1147	-408.2565	3.0422	0.7173
		MAX	19	1	-0.151	0.5803	3.0422	3.0376
30	29	MIN	40	30	-190.2272	-228.4218	3.0422	1.0482
		MAX	19	1	-0.1511	0.5803	3.0422	3.0376
56	3	MAX	54	56	46.0909	94.6216	3.0422	5.6432

Table 7.5 An estimation of the Response Time and Power Consumption for PDA as Client with BW = 1000kb/s and RCPUS = 116.

BW	1000Kb/s	RCPUS	116					
CS	NP		PN	PS	RTIP	PCIP	LPerformance	RPerformance
1	58	MIN	56	1	-1195.1591	-1562.5096	3.0422	0.2348
		MAX	19	1	0.9503	0.5803	3.0422	3.0714
20	39	MIN	30	20	-303.8007	-408.2565	3.0422	0.75340
		MAX	19	1	0.9503	0.5803	3.0422	3.07143
30	29	MIN	40	30	-162.448	-228.4218	3.0422	1.15917
		MAX	19	1	0.95038	0.5803	3.0422	3.07143
56	3	MAX	54	56	95.0166	94.6216	3.0422	61.0477

Table 7.6 An estimation of the Response Time and Power Consumption for Laptop as Client with BW = 19.2kb/s and RCPUS = 4.

BW	19.2Kbit/s	RCPUS	4					
CS	NP		PN	PS	RTIP	PCIP	LPerformance	RPerformance
1	58	MIN	56	1	-242934.7352	-250704.429	11.7333	0.0048
		MAX	19	1	-255.2646	-263.4238	11.7333	3.3026
20	39	MIN	30	20	-69254.4757	-71469.3350	11.7333	0.0169
		MAX	19	1	-255.2646	-263.4238	11.7333	3.3026
30	29	MIN	40	30	-43884.0493	-45287.4589	11.7333	0.0266
		MAX	19	1	-255.2646	-263.4238	11.7333	3.3026
56	3	MAX	54	56	-731.9275	-755.1185	11.7333	1.4103

Table 7.7 An estimation of the Response Time and Power Consumption for PDA as Client with BW = 1000kb/s and RCPUS = 4.

BW	1000Kb/s	RCPUS	4					
CS	NP		PN	PS	RTIP	PCIP	LPerformance	RPerformance
1	58	MIN	56	1	-31278.9412	-39487.9175	78.7069	0.2508
		MAX	19	1	-31.4053	-39.6075	78.7069	59.8962
20	39	MIN	30	20	-8892.3819	-11225.4662	78.7069	0.8752
		MAX	19	1	-31.4053	-39.6075	78.7069	59.8962
30	29	MIN	40	30	-5614.8727	-7087.4938	78.7069	1.3772
		MAX	19	1	-31.4053	-39.6074	78.7069	59.8962
56	3	MAX	54	56	-28.9268	-34.7293	78.7069	61.0478

From previous tables, available bandwidth is an important factor. Tables 7.2 and 7.3 show that if the bandwidth is the bottleneck in the system, no matter what the relative CPU speed is, neither reduction in power consumption nor increases in MP3 player performance can happen. However, if the bandwidth is not the bottleneck, then the relative CPU speed becomes decisive factor in increasing the performance and the decreasing power consumption at the mobile device. Tables 7.4 and 7.5 show that it is possible to save power and increase performance of the MP3 player if the entire decoder will be executed remotely and the PDA only works as sound player. However, increasing performance of the MP3 player it does not mean that the MP3 player becomes an efficient player unless the performance index is 100% or more.

The decrease in the power consumption happens when the available bandwidth is high. This is because the lower the bandwidth is; the longer it takes to transmit data, which in turn, cause more power consumption. Table 7.4 and Table 7.5 show that with high bandwidths, regardless of the relative CPU speed, there is a considerable gain in power consumption.

The computation power of the mobile device is an important factor as well. Table 7.6 and Table 7.7 show that local efficiency of the player is always higher than remote efficiency even though the available bandwidth in Table 7.7 is sufficient to handle the decoded sound and the computational power is quit high at the proxy server. This argues for the use of multi-set of multi proxies' infrastructure to support multi mobile users rather than one proxy server infrastructure for multi mobile users [80].

Results show that it is not always beneficial to start shipping code to gain performance and/or decrease power consumption. They totally depend on the graph topology and the data traffic volume between objects. The following observation is specifically for the MP3 player mentioned in Chapter 6. The Tables 7.4 and 7.5 show that there is a considerable decrease in power consumption as well as an increase in the performance of the MP3 player; however, Tables 7.6 and 7.7 show that it is not worth shipping the MP3 decoder remotely due to high degradation in both power consumption and performance.

Previously mentioned parameters are important in determining which of application objects should be offloaded to the proxy server. The GGP determines which objects of an object graph should be clustered together; but it does not determine which objects should be moved. The previous equations help in deciding which of the objects should be offloaded to the proxy server if offloading is beneficial.

Generally, the previous tables indicate that for the PDA case, it is worth offloading the entire decoder to the proxy server if there is high bandwidth and the relative CPU speed is high as well.

In Table 7.5, partition number 56 that holds the complete decoder is the only cluster that allows the increase in performance of the MP3 decoder and decrease in the power consumption. This is because none of the other clusters give better results with the same environment conditions. The output of the GGP algorithm depends completely on the topology of the object graph. In the MP3 object graph, the amount of the traffic

between the objects is very high. This is because of the nature of the MP3 decoder not to mention the centralized topology of the MP3 decoder, which favors shipping the entire decoder rather shipping a part of the decoder. If the graph topology as well as the amount of the traffic between objects changes, it might be possible to increase performance and reduce the power consumption by partially shipping objects of the object graph rather than the entire object graph.

We used voyager with the MP3 player to offload certain clusters from laptop to the proxy server, measure the efficiency of the player, and compare them with the estimated values in Table 7.7. The player decodes 38 frames (approximately one second to play) and converts the stereo output to mono of sampling rate 11025 MHz with 16 bits per sample. Table 7.8 shows the chosen clusters and the partition numbers, the local efficiency, and the remote efficiency when the certain clusters of objects are moved remotely to the proxy server.

Table 7.8: Efficiency of the MP3 player when certain clusters are shipped remotely for execution.

BW	1000Kbit/sec		
Cluster size	#PN	Local Performance index%	Remote Local Performance index
1	19	76.24	61.89
30	40	76.24	2.35
56	54	76.24	65.40

The measured values in Table 7.8 shows that there is no much difference between them and the estimated values in Table 7.7. Thus, the equations 7.1 to 7.6 can be used to have an estimation of the cost model of offloading application objects remotely.

8. Conclusion and Future Work

In this chapter we summarize the thesis and introduce possible improvements and future work on the adaptive mobile toolkit.

8.1 Conclusions

Finding approaches to reduce power consumption and to improve application performance is a vital and interesting problem to be investigated. On many levels, approaches have been developed to address the problem of reducing power consumption and increasing the response time. They range from hardware to software level approaches as mentioned previously.

One of the approaches is to divide a mobile application statically at design time into a server and client model, where the client executes at the mobile device and the server runs at a fixed host in the wired network. Splitting an application statically does not guarantee the maximum quality of service to the users, especially in mobile computing environments due to the mobile computing environment challenges and the highly dynamic fluctuation of available resources.

To improve quality of service to the users, at the fixed host, filtering mechanisms that work according to the current condition of the mobile computing environment are deployed, which make mobile applications more adaptive. However, in our thesis work we suggested a new approach based on Greedy Graph partition for adaptive mobile applications, in which an application's objects will be split dynamically between the mobile device and fixed host according to the mobile device and fixed host's available resources and wireless network state.

This approach requires special infrastructure and tools rather than a specific application design. Two issues are important for realizing application adaptation. The first is that the operating system must support a mechanism of notifying applications of changes in the mobile environment. The second is to provide a systematic way to build adaptive applications embodied in frameworks and toolkits. Thus we designed and developed a mobile object toolkit that run on WindowsCE platform that run JVM. With this toolkit we combine JVMs on both the proxy server and the mobile device as one virtual machine from the application point of view to dynamically split applications object between JVMs.

Mobile applications, especially ones that do intensive computation and communication, can be divided dynamically as a client and server between the wired network and the mobile device according to the mobile environment and to the availability of the resources on both the mobile device and the wired network. With our approach, we allow more windows of adaptability to the mobile environment. In addition,

it allows the applications to have dynamic access to faster machines through faster servers. This will increase the performance of applications and reduce the power consumption on mobile devices since offloading computation to the wired network will reduce the CPU cycles and memory required to achieve certain tasks at mobile devices.

Although Java as is our primary developing language for applications as well as for implementing our toolkit, Java Virtual Machines are in early stages of development, particularly those for the WindowsCE platform. They need to be extended to export the mobile computing environment variables, such as available bandwidth, battery lifetime and power available at the mobile host as well as performance parameters such as CPU utilization. These extensions require the use of native interfaces, which if not standardized, will prevent the mobile adaptive application from being portable.

We suggested a modified Greedy Graph Partitioning algorithm to group objects for Load Sharing purposes. As proof of concept, we implemented an MP3 Player in Java. We measured the CPU time and data volume traffic of its object graph that was obtained by special toolkits based on instrumented Java Virtual Machines.

To demonstrate the feasibility of the dynamic load balancing approach, we use the MP3 player object graph as input to the Greedy Graph Partition to obtain clusters that contain strongly related objects. Through calculation models which are based on the available bandwidth and the relative CPU speeds to estimate power consumption costs and performance costs metrics, we determine which of the clusters should be moved to improve one or both of the metrics. The results showed that it is possible to

simultaneously improve both metrics by dynamically shipping the entire MP3 decoder to the proxy server.

8.2 Future Work

We highly suggest the improvement of the mobile object toolkit to help facilitate the implementation of the adaptive mobile application for PDAs in particular. The main improvement on this toolkit would be improving the implementation of the RMI protocol, which is based on serialized object commands between the object servers on both Java Virtual Machines. Currently, we manually write proxy objects; however, we suggest developing tools to automate this process and integrate it with the toolkit.

Determining the clustering level of object graph of an application with more robust algorithm is another avenue of the future research. Currently we run the algorithm with all possible clustering sizes. Other possible ways could use an estimate to level of clustering. Ultimately, we require an algorithm that would react to the rapid changes of the environment, and we need to investigate how to reduce the impact of rapid change in the environment.

The MP3 player we implemented does not react to the changes in bandwidth. We fixed the output playing rate and the sampling size. Further study is required to show how application adaptation policies affect and interact with the automated adaptation by our toolkit.

References

[1] T. Imielinkski and B. R. Bardinate, *Data Management for Mobile Computing*, SIGMOD Record, 22(1): 34-39, March 1993.

[2] Meier-Hellstern, K.S, Alonson, E. and Oniel, D., *The use of SS7 and GSM to support high-density personal communications*, In proceedings of the Third Winlab Workshop on the third generation of wireless information networks.

[3] J. Ioannisdis, D. Duchamp, and G. Magurier Jr. *IP-based Protocols for Mobile Internetworking*, In the Proceedings of SIGCOMM'91 Symposium, pages 253-245, Sept 1991.

[4] M. Weiser, *Some Computer Science Issues in Ubiquitous Computing*, Communications of ACM 36(7): 75-84, July 1993.

[5] G. Forman and J. Zahorjan, *The Challenges of Mobile Computing*, Computer, April 1994.

[6] C. Horstman and G. Cornell, *Core Java 1.1*, Volume 1, Sun Microsystems Press, A Prentice Hall title.

[7] J. Baugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*. ISBN: 0136298419. Publisher. Prentice Hall.

[8] The Java(tm) Language Environment: A White paper, http://ftp.unican.es/manuales/java/white-paper/java-whitepaper-6.html

[9] J. Gosling, B. Joy, G. Steele, The Java Language Specification, http://java.sun.com/docs/books/jls/html/5.doc.html#20232

[10] Security Code Guidelines, http://java.sun.com/security/seccodeguide.html#gcg

[11] C. Mangione , *Just In Time for Java vs. C++*, http://www.ncworldmag.com/ncworld/ncw-01-1998/ncw-01-jperf.html

[12] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, Hideaki Komatsu, and T. Nakatani, *Overview of the IBM Java Just-In-Time Compile*, IBM Systems Journals, Java Performance Issue, Vol. 39, No. 1, 2000.

[13] S. Kalyanaraman, Jini Overview, http://www.cse.ucsc.edu/~shankari/jiniwriteup/ucscjini.html

[14] *WindowsCE Microsoft home page*, http://www.microsoft.com/WindowsCE/

[15] L. Peterson, B. Davie, *Computer Networks: A Systems Approach*, Morgan Kaufmann, 1996, (ISBN 1-55860-368-9)

[16] Object Management Group, http://www.omg.org.

[17] *Object and Components*, http://a-sync.com/chapter1.htm

[18] J.Inscore, *Java IDL*, http://www.uni-koblenz.de/~admin/Doku/Java /Tutorial/idl/intro/intro.html

[19] *The Common Object Request Broker: Architecture and Specification*, Revision 2.2, February 1998, http://www.ti5.tu-harburg.de/Manual/OMG/CORBA/1

[20] *JDBC APIs*, http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/index.html.

[21] Java Remote Method Invocation: Distributed Computing fort Java, http://cermics.enpc.fr/doc/java/rmi.html

[22] W. Edwards, X. Parc, *Core Jini*, Published June 1999 by Prentice Hall PTR (ECS Professional)

[23] K. Arnold, Addison-Wesley, Hardcover, *The Jini Specification*, Published June 1999 ISBN 0201616343

[24] Voyager Mobile Code toolkit, http://www.objectspace.com/products/vgrorb.asp

[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison Wesley Professional Computing Series, ISBN 0-201-63361-2.

[26] *Distributed Component Object Model Protocol,* http://www.microsoft.com/oledev/olemkt/oledcom/dcomspec.txt

[27] WaveRider NCL Series, http://www.ttiwireless.com/products/ncl/ncl.html

[28] *Wrireless Lans*, http://www.proxim.com/products/rl2/index.shtml

[29] Nortel Networks Wireless products, http://www.nortelnetworks.com/products/wireless

[30] The CDPD System, http://www.lucent.com/wirelessnet/products/networks/cdpd_howworks.html

[31] American Mobile's ARDIS Network, http://www.ardis.com/communication/fr_ardisnet.html

[32] B. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, K. Walker, *Agile Application-Aware Adaptation for Mobility*, Proceedings of the 16th ACM Symposium on Operating System Principles, October 1997, St. Malo, France

[33] M. Satyanarayanan, *Mobile Information Access* http://www.cs.cmu.edu/afs/cs.cmu.edu/project/coda//Web/docdir/ieeepcs95.pdf

[34] P. Ward, J Black, T. Kunz, M. Nidd, M. Lioy, B. Elphick, and M. Ostrowski, *Comma, A Communication Manager for Mobile Applications* In Wireless 98. TR Labs, TRIO, IEEE Canada July 1998.

[35] *XDR: External Data Representation Standard*, RFC1832.

[36] J. Case, M. Fedor, M. Schoffstal,, J. Davin, *A Simple Network Management Protocol: SNMP*, May 1990, RFC1157

[37] A. Joseph and M. Kaashoek , *Building Reliable Mobile-Aware Applications using the Rover Toolkit*, M.I.T.Laboratory for Computer Science Cambridge, MA 02139, U.S.A.

[38] L. Ranganathan, A. Acharya, S. Sharma and J.Saltz, *Network-aware Mobile Programs*, Department of Computer Science University of Maryland College Park, MD 20740

[39] O. Angin, A.T Campbell., M.E Kounavis, and R Liao., *The Mobiware Toolkit: Programmable Support for Adaptive Mobile Netwokin, IEEE Personal* Communications Magazine, Special Issue on Adapting to Network and Client Variability, invited paper, August 1998,

[40] A. Lazar, *Programming Telecommunication Networks*, IEEE Network, October 1997.

[41] S. Sheng, A. Chandrakasan, R. Brodersen, *A Portable Multimedia Terminal*, IEEE Comunications Magazine, Dec. 1992.

[42] H. Balakrishanan, S. Seshan, E. Amir, and, R. Katz. *Improving TCP/IP performance over wireless network*, In Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM), Nov. 1995.

[43] R. Kravets, K. Calvert, and K. Schwan, *Payoff adaptation of communication for distributed interactive applications*, Journal of High Speed Networking, *Special Issue on Multimedia Communications*, 1998.

[44] B. Narendran, J. Sienicki, S. Yajnik, and P. Agrawal. *Evaluation of an adaptive power and error control algorithm for wireless system*, In Proceedings of the IEEE International Conference on Communications (ICC'97), 1997.

[45] F. Douglis, P. Krishan, and B. Bershad, *Adaptive disk spin down policies for mobile computers*, In Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing, April 1995.

[46] I. Chlamatc, C. Petrioli, and J. Redi, *Energy conservation in access protocols for mobile computing and communication*, Microprocessors and Microsystems Journal, 1998.

[47] F. Douglis, P. Krishna, and B. Marsh, *Thwarting the power hungry disk*, In Proceedings of the 1994 Winter USENIX Conference, Jan. 1994.

[48] K. Govil, E. Vhan, and H. Wasserman, *Comparing algorithms for dynamic speed-setting of a low-power CPU*, In Proceedings of the First ACM International Conference on Mobile Computing and Networking (MOBICOM), 1995.

[49] M. Othman and S. Hailes, *Power Conservation Strategy for Mobile Computers Using Load Sharing*, Mobile Computing and Communication Review, Vol.2, No. 1, Pages 44-51.

[50] T. Waston, *Application Design for Wireless Computing,* Proceedings of the IEEE Workshop on Mobile Computing, Dec. 1994.

[51] B. Badrinath, A. Acharya, T. Imielinski, *Impact of Mobile Computing,* Operating Systems Review, Vol.27, No.2, April 1993.

[52] Intel Corporation, Microsoft Corporation, and Toshiba Corporation., *Advanced Configuration and Power Interface Specification V1.2*, July 1998. www.intel.com/IAL.powermgm.

[53] Object Serialization, http://java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html

[54] ObjectSpace,Inc.:http://www.objectspace.com/developers/voyager

[55] *Revolutionary RMI: Dynamic class loading and behavior objects,* http://www.javaworld.com/javaworld/jw-12-1998/jw-12-enterprise.html

[56] R. Kravets and P. Krishnan, *Power Management Techniques for Mobile Communication*, MOBICOM 98, P 157-168

[57] S. Omar and T. Kunz, *Reducing power consumption and increasing application Performance for PDAs through mobile code*, in Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications, Vol. II, Las Vegas, Nevada, USA, pp. 1005-1011, June 1999.

[58] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. *Efficient and Language Independent Mobile Programs*, In Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation, pages 127--36, May 1996.

[59] N. Borenstein, *Email With a Mind of its Own: The Safe-TCL Language for Enabled Mail*, In Proceedings of IFIP Working Group 6.5 International Conference, pages 389-402, Jun 1994.

[60] J. Gosling and H. McGilton, *The Java Language Environment*, 1995, http://osm-www.informatik.uni-hamburg.de/osm-www/resources/local Documents.html

[61] S. Clamen, L. Leibengood, S. Nettles, and J. Wing, *Reliable Distributed Computing with Avalon/Common Lisp*, In Proceedings of the International Conference on Computer Languages, pages 169--79, 1990.

[62] J. Falcone., *A Programmable Interface Language for Heterogeneous Systems*, ACM Transactions on Computer Systems, 5(4): 330--51, Nov. 1987.

[63] J. Stamos and D. Gifford, *Implementing Remote Evaluation*, IEEE Transactions on Software Engineering, 16(7):710--722, July 1990.

[64] L. Cardelli, *A Language With Distributed Scope*, In Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, Jan. 1995.

[65] R. Gray, D. Kotz, S. Nog, D. Rus, G. Cybenko, *Mobile Agent for Mobile Computing*, Department of Computer Science, Dartmouth College, Tech. Report, PCS-TR96-285.

[66] A. Bakre, and B. Badrinath, *I-TCPL Indirect TCP for Mobile Hosts*, In the Proceedings of the 15th International Conference on Distributed Computing Systems.

[67] C. Walshaw, M. Cross, and M. Everett. *Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes*. J. Par. Dist. Comput., 47(2):102-108, 1997.

[68] Y.T Wang, R.I.J Morris, *Load Sharing in Distributed Systems*, IEEE Transactions on Computers, IEEE Workgroup, Vol. C-34. No. 3 March 1985.

[69] *Proxy Pattern*, http://www.media.mit.edu/~tpminka/patterns/Proxy.html.

[70] M. Dahm, *Byte Code Engineering*, in Proceedings of JIT'99.

[71] A. Oppenheim and R. Schafer, *Discrete Time Signals Processing*, (Englewood, NJ: Prentice-Hall, 1989: 80-87.

[72] K. Pohlman, *Principles of Digital Audio* (Indianapolis, IN: Howard W. Sams and Co., 1989).

[73] J. Tobias, *Foundation of Modern Auditory Theory* (New York and London: Academic Press, 1970): 159-202.

[74] Davis Pan, *A Tutorial on MPEG/Audio Compression, IEEE MultiMedia,* Vol. 2, No. 2, Summer 1995: 60-73

[75] K. Brandenburg, J. Herre, J. D. Johnston, Y. Mahieux, and E. Schreder, ASPEC: *Adaptive Spectral Perceptual Entropy Coding of High Quality Music Signals,* Preprint 3011, 90th Audio Engineering Society Convention, Paris (1991).

[76] D. Huffman, *A method for the construction of Minimum Redundancy Codes* Proceedings of the IRE, vol. 40, 1962: 1098-1101.

[77] *Java profiling toolkit: JProbe,* http://services.klgroup.com

[78] Wiese and G. Stoll, *Bit rate Reduction of High Quality Audio Signals by Modeling the Ear's Masking Thresholds,* Preprint 2970,89th Audio Engineering Society Convention, Los Angeles (1990).

[79] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte, *Clustering in Object base,* Technical report 6.92, Fakulat fur Informatik, University Karlsruhe, D-7500 Karlsruhe, Jun 1992.

[80] W. Jianwen , *A Proxy server infrastructure for adaptive mobile applications*, 1967- Ottawa. 1999, Thesis (M.C.S.) - Carleton University, 1999. CALL NUMBER: M.C.S. 1999.W35

Appendix A

Java Native Interface for WindowsCE platforms

- **Introduction**

The Raw Native Interface (RNI) is a set of functions and structures defined in native.h and nativcom.h. RNI provides an efficient way of both traversing between Java and native code, and manipulating objects while in native code.

Using RNI requires knowledge of garbage collection, pointer locking, and other issues. Dynamic-link libraries (DLLs) you call through RNI need to be specially written to work with RNI. You can not use RNI to call arbitrary DLLs, although you can write wrapper DLLs for them.

To work with RNI, native code must follow naming conventions, adapt to Java data representations, and deal with the garbage collection process of the Java environment. For example, if the native code is going to be in a long loop, the garbage collector should be called periodically. Similarly, if the code is going to block on user

input or call a thread that will block, it should call the garbage collector around the code that can block. Furthermore, objects need to be protected by garbage collection frames when calling back into Java.

All this increases programming complexity. In exchange for this complexity, you get greater speed and full access to all objects' fields. A good use of RNI is systems-level programming tasks, such as extending the capabilities of the Microsoft virtual machine (Microsoft VM). The example describes the mechanism of calling between Java and native code (typically C or C++) using the Microsoft VM for WindowsCE platform.

For further information of how to write DLL, please consult the following web page. ***http://msdn.microsoft.com/library/devprods/vs6/visualc/vccore/_core_dlls.3a_.overview.htm***

- **Example: Wave Output extension for JVM in Windows CE**

To generate, compile and integrate the native code, stub files and header files of any native classes please read the release notes of the JVM of WindowsCE version 1.1 The following example shows how you could extend the Java Virtual Machine such that you would be able to play PCM or wave file formats through Java applications.

The example contains the following files.

- ceWaveOutDevice.java : A java file that contains the native declaration of the wave output device of the WindowsCE.

- ceWaveOutDeviceDLLMain.cpp: A C++ file that contains the starting point of the DLL entry point and the necessary related initialization
- WaveOutDevice.cpp : A C++ file that contains the implementation of the wave output device for WindowsCE.
- WaveOutDevice.h: The header file of the WaveOutDevice.cpp
- CeWaveOutDeviceDLL.def : The required definitions for the compiler to compile the DLL.

```
package javacex.ceWaveOutDevice;

import java.io.*;

public class ceWaveOutDevice
{
        static{

                System.loadLibrary("ceWaveOutDeviceDLL");
        }

        public ceWaveOutDevice( int nBitsPerSample, int nFreq, int nChannels)
        {
                this.nBitsPerSample = nBitsPerSample;
                this.nFreq = nFreq;
                this.nChannels = nChannels;
                this.totalBytesPerSec = nFreq*(nBitsPerSample/8)*nChannels;
        }


        private int nBitsPerSample;
        private int nFreq;
        private int nChannels;
        private int totalBytesPerSec;
        private int exceptionNumber;


        private synchronized native int     ceWaveOutDeviceOpen( int nBitsPerSample, int nFreq, int
                                             nChannels);
        private synchronized native int     ceWaveOutDeviceClose();
        private synchronized native int     ceWaveOutDeviceWrite(byte[] data, int size);
        private synchronized native boolean ceWaveOutDeviceStillPlaying();
        private synchronized native int     ceWaveOutDeviceSetVolume(int l_volum, int r_volum);
        private synchronized native int     ceWaveOutDeviceGetVolume();
        private synchronized native int     ceWaveOutDevicePause();
```

```
        private synchronized native int         ceWaveOutDeviceRestart();
        private synchronized native int         ceWaveOutDeviceMute();

        // open the device

        public synchronized void open()
        {
                ceWaveOutDeviceOpen( nBitsPerSample, nFreq, nChannels );
        }
        public synchronized void pause()
        {
                ceWaveOutDevicePause();
        }
        public synchronized void restart()
        {
                ceWaveOutDeviceRestart();
        }
        public synchronized void close()
        {

                ceWaveOutDeviceClose();
        }

        public synchronized void write(byte[] data, int size)
        {
                 ceWaveOutDeviceWrite( data, size );
        }

        public synchronized boolean  stillPlaying()
        {
             return ceWaveOutDeviceStillPlaying();
        }

        public synchronized void  setVolume(int l_volum,  int r_volum)
        {
                ceWaveOutDeviceSetVolume(l_volum, r_volum);
        }

        public synchronized void mute()
        {
                ceWaveOutDeviceMute();
        }

        public synchronized int getRVolume( int value )
        {
                return ( (int) (( value & (int)0x0000FF00) >> 8));
        }
        public synchronized int getLVolume( int value )
        {
                return ( (int) ( value & (int)0x000000FF) );
        }
        public synchronized int getVolume()
        {
                return ceWaveOutDeviceGetVolume();
        }
}
```

```
//
// ceWaveOutDeviceDLLMain.cpp
//

#include<windows.h>
#include<windowsx.h>
#include<native.h>
#include"WaveOutDevice.h"
#include<ce_javacex_ceWaveOutDevice_ceWaveOutDevice.h>

WCHAR *szAppName = L"RawSoundDevice";
WCHAR *szTitle  = L"RawSoundDevice";

HINSTANCE       g_hinst      = NULL;   // instance of library
HANDLE          ghLibHeap    = NULL;   // global heap used for library
HWND            hWnd;
DWORD           exceptionNumber;


BOOL InitApplication (HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow);
BOOL DllIntialization( HINSTANCE hInstance,int nCmdShow);

LRESULT CALLBACK WndProc (    HWND hWnd,UINT message,WPARAM uParam,LPARAM lParam);

void onDESTROY_Msg( HWND hWnd,          UINT message,WPARAM uParam,     LPARAM lParam);
void onPAINT_Msg( HWND hWnd,    UINT message,WPARAM uParam,      LPARAM lParam);

WaveOutDevice *wod = NULL;

void setExceptionNumber( WaveOutDeviceError *wode)
{
  if( wode->errorCataogary )
        {
          exceptionNumber = 500;
        }
        else
          exceptionNumber = 501;
}

__declspec(dllexport) DWORD          __cdecl RNIGetCompatibleVersion()
{
        return RNIVER;
}

__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceMute( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis )
{
 long result = 0L;
 WaveOutDeviceError wode;

 if( wod != NULL )
 {
        Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
        GCFrame gcf;
```

```
        GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));
        phThisSafe = hThis;
        ObjectMonitorEnter(phThisSafe);

        result = wod->mute();
        if( wod->getError( &wode ) )
        {
                setExceptionNumber(&wode);
                if( phThisSafe != NULL )
                  ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)->exceptionNumber =
                   exceptionNumber;
        }

        ObjectMonitorExit(phThisSafe);
        GCFramePop(&gcf);
        }

 return result;
}

__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceRestart( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis )

{

 long result = 0L;
 WaveOutDeviceError wode;
 {
        if( wod != NULL )
        {

                Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;

                GCFrame gcf;

                GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));

                phThisSafe = hThis;

                ObjectMonitorEnter(phThisSafe);

                result = wod->restartPlaying();

                if( wod->getError( &wode ) )
                {
                        setExceptionNumber(&wode);
                        if( phThisSafe != NULL )
                          ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)-
                           >exceptionNumber = exceptionNumber;

                }
                ObjectMonitorExit(phThisSafe);
                GCFramePop(&gcf);
        }
 }
 return result;
```

```
}
__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDevicePause( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis )

{
 long result = 0L;
 WaveOutDeviceError wode;
 {
        if( wod != NULL )
        {
                Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
                GCFrame gcf;
                GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));
                phThisSafe = hThis;
                ObjectMonitorEnter(phThisSafe);
                result = wod->pausePlaying();


                if( wod->getError( &wode ) )
                {
                        setExceptionNumber(&wode);
                        if( phThisSafe != NULL )
                          ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)-
                        >exceptionNumber = exceptionNumber;

                }
                ObjectMonitorExit(phThisSafe);
                GCFramePop(&gcf);
        }
 }
 return result;
}
__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceSetVolume(
struct Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis, long long1, long long2 )
{
 long result = 0L;
 {
        if( wod != NULL )
        {
                Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
                WaveOutDeviceError wode;

                GCFrame gcf;
                GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));

                phThisSafe = hThis;
                ObjectMonitorEnter(phThisSafe);
                result = wod->setVolume( long1, long2);
                if( wod->getError( &wode ) )
                {
                        setExceptionNumber(&wode);
                        if( phThisSafe != NULL )
                    ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)->exceptionNumber =
                    exceptionNumber;
                }
                ObjectMonitorExit(phThisSafe);
```

```
                    GCFramePop(&gcf);
            }
 }
 return result;
}

__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceGetVolume(
struct Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis )

{

 long result = 0L;
 DWORD value;
 WaveOutDeviceError wode;
 {
          if( wod != NULL )
          {
                    Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;

                              GCFrame gcf;
                              GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));
                              phThisSafe = hThis;
                              ObjectMonitorEnter(phThisSafe);
                              result = wod->getVolume( &value);
                              if( wod->getError( &wode ) )
                              {
                                        setExceptionNumber(&wode);
                                        if( phThisSafe != NULL )
                                        ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)-
>exceptionNumber = exceptionNumber;

                              }

                              ObjectMonitorExit(phThisSafe);
                              GCFramePop(&gcf);

          }
  //UnprepareThreadForJava(&threadEntryFrame);
 }
 return value;

}
__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceWrite( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis, struct HArrayOfByte* obj1, long long1 )
{
  long result = 0L;
  char *data;
  WaveOutDeviceError wode;

  struct {
        Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
              HArrayOfByte* pobj1Safe;
  } gc;

   GCFrame gcf;
```

```
	{
		if( wod != NULL )
		{
			GCFramePush(&gcf, &gc, sizeof(gc));

			gc.phThisSafe = hThis;
			gc.pobj1Safe = obj1;
			ObjectMonitorEnter(gc.phThisSafe);
			data =  ct_unhand(ArrayOfByte,gc.pobj1Safe)->body;
			result = wod->playSound( data, long1);
			if( wod->getError( &wode ) )
			{
				setExceptionNumber(&wode);
				if( gc.phThisSafe !=  NULL )
				ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,gc.phThisSafe)-
>exceptionNumber = exceptionNumber;

			}

			ObjectMonitorExit(gc.phThisSafe);
			GCFramePop(&gcf);


		}

	}
	return result;
}
__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceClose( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis )

{

	long result = 0L;
	//ThreadEntryFrame threadEntryFrame;


	//if(	PrepareThreadForJava(&threadEntryFrame) )
	{
	 if( wod != NULL )
	 {
			Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
			GCFrame gcf;
			GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));
			phThisSafe = hThis;
			ObjectMonitorEnter(phThisSafe);
			if( wod != NULL )
			{
				delete wod;
				wod = NULL;
			}
			ObjectMonitorExit(phThisSafe);
			GCFramePop(&gcf);

	 }
	 //UnprepareThreadForJava(&threadEntryFrame);
```

```
        }

        return result;
}

__declspec(dllexport) long __cdecl javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceOpen( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis, long long1, long long2, long long3 )
{

 long result = 0L;
 WaveOutDeviceError wode;
 //ThreadEntryFrame  threadEntryFrame;

 //if( PrepareThreadForJava(&threadEntryFrame) )
 {

        if( wod == NULL )
        {

                Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
                GCFrame gcf;
                GCFramePush(&gcf. &phThisSafe, sizeof(phThisSafe));
                phThisSafe = hThis;
                ObjectMonitorEnter(phThisSafe);
                wod = new WaveOutDevice(hWnd,( WORD) long1,long2,(WORD) long3 );
                if( wod != NULL )
                {
                        if( wod->getError( &wode ) )
                        {
                          setExceptionNumber(&wode);
                          if( phThisSafe != NULL )
                            ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)-
                           >exceptionNumber = exceptionNumber;

                        }
                }
                else
                {
                        // Another level off error.
                }

                ObjectMonitorExit(phThisSafe);
                GCFramePop(&gcf);
        }
        // UnprepareThreadForJava(&threadEntryFrame);
 }

 return result;
}

__declspec(dllexport) /* boolean */ long __cdecl
javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceStillPlaying( struct
Hjavacex_ceWaveOutDevice_ceWaveOutDevice* hThis )
{
```

```
        long result = 0L;
        WaveOutDeviceError wode;
        Hjavacex_ceWaveOutDevice_ceWaveOutDevice *phThisSafe;
        GCFrame gcf;
        //ThreadEntryFrame  threadEntryFrame;

  //if( PrepareThreadForJava(&threadEntryFrame) )
        {

                if( wod != NULL )
                {

                        GCFramePush(&gcf, &phThisSafe, sizeof(phThisSafe));
                        phThisSafe = hThis;
                        ObjectMonitorEnter(phThisSafe);
                        result = wod->stillPlaying();
                        if( wod->getError( &wode ) )
                        {
                          setExceptionNumber(&wode);
                                if( phThisSafe !=  NULL )
                    ct_unhand(javacex_ceWaveOutDevice_ceWaveOutDevice,phThisSafe)->exceptionNumber =
exceptionNumber;

                        }

                        ObjectMonitorExit(phThisSafe);
                        GCFramePop(&gcf);
                }
        //      UnprepareThreadForJava(&threadEntryFrame);
        }
        return result;
}

BOOL  CALLBACK  DllMain(HINSTANCE hinst, DWORD dwReason, LPVOID lpv)
{
  switch (dwReason)
        {

                case DLL_PROCESS_ATTACH:

                                if( DllIntialization(hinst, SW_HIDE) == FALSE )
                                        return FALSE;
                                sndPlaySound(L"\\Windows\\Alarm1.wav",SND_SYNC);
                                if (!g_hinst)
                                        g_hinst = hinst;
                                if ((ghLibHeap = HeapCreate(0,2048,0))==NULL)
                                {
                                                return FALSE;
                                }
                                break;

                case DLL_PROCESS_DETACH:

                                DestroyWindow(hWnd);
                                sndPlaySound(L"\\Windows\\Alarm2.wav",SND_SYNC);
                                if (ghLibHeap)
```

```
                                        HeapDestroy( ghLibHeap);
                                    break;
    }
    return TRUE;
}
/*************************************************************
*
*       LRESULT CALLBACK WndProc (    HWND hWnd,UINT message,WPARAM uParam,LPARAM lParam)
*
*************************************************************/

LRESULT CALLBACK WndProc (    HWND hWnd,UINT message,WPARAM uParam,LPARAM lParam)
{


    switch (message)
        {
            case WM_PAINT:
                                        onPAINT_Msg(hWnd,message,uParam,lParam);
            break;
            case WM_DESTROY:
                                        onDESTROY_Msg(hWnd,message,uParam,lParam);
                                        break;

            default:
          return (DefWindowProc(hWnd, message, uParam, lParam));
    }

    return (0);
}

/*************************************************************************

void onPAINT_Msg( HWND hWnd,    UINT message,WPARAM uParam,     LPARAM lParam)

*************************************************************************/

void onPAINT_Msg( HWND hWnd,    UINT message,WPARAM uParam,     LPARAM lParam)
{

        HDC hdc;
    PAINTSTRUCT ps;
        hdc = BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);

}

/*************************************************************************

void onDESTROY_Msg( HWND hWnd,          UINT message,WPARAM uParam,     LPARAM lParam)

*************************************************************************/

void onDESTROY_Msg( HWND hWnd,          UINT message,WPARAM uParam,     LPARAM lParam)
{
```

```
        if( wod != NULL )
        {
                delete wod;
                wod = NULL;
        }

        PostQuitMessage(0);
}


/*************************************************************
*
*       BOOL InitApplication (HINSTANCE hInstance)
*
*************************************************************/

BOOL InitApplication (HINSTANCE hInstance)
{
  WNDCLASSW wc;

  wc.style = 0 ;
  wc.lpfnWndProc = (WNDPROC) WndProc;
  wc.cbClsExtra = 0;
  wc.cbWndExtra = 0;
  wc.hInstance = hInstance;
  wc.hIcon = NULL;
  wc.hCursor = NULL;
  wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = NULL;
  wc.lpszClassName = szAppName;

  BOOL f = (RegisterClass(&wc));
  return f;
}

/*****************************************************************************************

                BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)

*****************************************************************************************/

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{

        // Use the default window settings.
        hWnd = CreateWindow(szAppName,szTitle,WS_VISIBLE | WS_SYSMENU,
                        0,0,CW_USEDEFAULT,CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL);

         if (hWnd == 0)   // Check whether values returned by CreateWindow() are valid.
                return (FALSE);
         if (IsWindow(hWnd) != TRUE)
           return (FALSE);
         ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  return(TRUE);                 // Window handle hWnd is valid.
```

```
}


BOOL DllIntialization( HINSTANCE hInstance, int nCmdShow )
{

                if (InitApplication(hInstance) == FALSE)
                        return(FALSE);

        if (InitInstance(hInstance, nCmdShow) == FALSE)
                return(FALSE);

                return TRUE;

}
```

```
// WaveOutDevice.cpp: implementation of the WaveOutDevice class.
//
//////////////////////////////////////////////////////////////////////

#include<windows.h>
#include<mmsystem.h>
#include"WaveOutDevice.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////


void copyMemory( PVOID dest, const void *src, DWORD len);
void zeroMemory( PVOID dest,DWORD len);

void CALLBACK WaveOutDeviceCallBackStub(HWAVEOUT hwo,UINT uMsg,DWORD dwUser,DWORD
dwParam1,DWORD dwParam2);

void CALLBACK WaveOutDeviceCallBackStub(HWAVEOUT hwo,UINT uMsg,DWORD dwUser,DWORD
dwParam1,DWORD dwParam2)
{
class WaveOutDevice *wod;
 wod = ( class WaveOutDevice *) dwUser;
 wod->WaveOutDeviceCallBack(hwo, uMsg, dwUser, dwParam1,dwParam2);
}

void WaveOutDevice::setErrorFields( DWORD gen, DWORD spec, DWORD line)
{
errorLineNumber = line;
 errorCataogary = gen;
 errorSpecefic = spec;
}


WaveOutDevice::WaveOutDevice(HWND hWnd,WORD wBitsPerSample,DWORD nSamplesPerSec,WORD
nChannels)
{


        WCHAR  *msg1 = L"WOD[Constructor[waveOutOpen1]]";
        WCHAR  *msg2 = L"WOD[Constructor[waveOutOpen2]]";
        WCHAR  *msg3 = L"WOD[Constructor[isCapable]]";

        this->hWnd = hWnd;
        possibleToGo = true;
        counterIn = counterOut  = 0L;



  InitializeCriticalSection(&cs);

  setWaveFormatXHeader(wBitsPerSample,nSamplesPerSec,nChannels);
  nWaveOutDevices = waveOutGetNumDevs();

  result =  waveOutOpen( &hwo, nWaveOutDevices-1 ,&wfx, 0L,0L, WAVE_FORMAT_QUERY);
```

```
        if( cheakWaveOutDeviceResult( result , msg1,__LINE__) )
        {

          result =  waveOutOpen( &hwo, nWaveOutDevices-1 ,&wfx,( DWORD) ::WaveOutDeviceCallBackStub
          ,(DWORD) this, CALLBACK_FUNCTION | WAVE_FORMAT_DIRECT );
           if( cheakWaveOutDeviceResult( result , msg2,__LINE__ ) )
                {
                        setErrorFields( WAVEOUTDEVICE_GENERAL_NOERR, 0L, __LINE__ );
                }
                else
                {
                        possibleToGo = false;
                        setErrorFields( WAVEOUTDEVICE_GENERAL_OPEN, result, __LINE__ );

                }
        }
        else
        {
                        possibleToGo = false;
                        setErrorFields( WAVEOUTDEVICE_GENERAL_OPEN, result, __LINE__ );
        }
         if( !isWaveOutDeviceCapable() )
        {
                if( cheakGeneralError( 1, msg3) )
                {
                }
                else
                {
                        possibleToGo = false;
                }
        }
}
int WaveOutDevice::mute( void )
{

  WCHAR  *msg1 = L"WOD[mute[pause]]";
        WCHAR  *msg2 = L"WOD[mute[reset]]";
        WCHAR  *msg3 = L"WOD[mute[restart]]";


        if( !possibleToGo )
        {
                return 0L;
        }
        EnterCriticalSection(&cs);

        result = waveOutPause(hwo);
        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
        {

                result = waveOutReset(hwo);
                if( cheakWaveOutDeviceResult( result , msg2,__LINE__) )
                {

                        result = waveOutRestart(hwo);
                        if ( cheakWaveOutDeviceResult(result,msg3,__LINE__) )
```

```
                        {
                                // ok we set the value;
                        }
                        else
                        {
                                possibleToGo = false;
                                setErrorFields( WAVEOUTDEVICE_GENERAL_RESTART, result,
__LINE__ );
                        }
                }
                else
                {
                        possibleToGo = false;
                        setErrorFields( WAVEOUTDEVICE_GENERAL_RESET, result, __LINE__ );

                }
        }
        else
        {
                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_PAUSE, result, __LINE__ );
        }
  LeaveCriticalSection(&cs);
        return result;
}


WaveOutDevice::~WaveOutDevice()
{

        WCHAR *  msg1 = L"WaveOutDevice[deconstructor[waveOutReset]]";
        WCHAR *  msg2 = L"WaveOutDevice[deconstructor[waveOutClose]]";

        // This test needs to be checked
        // to consider if the error has happened during playing.

        if( !possibleToGo )
        {
                DeleteCriticalSection(&cs);
                return;
        }

        result = waveOutReset(hwo);
        if( cheakWaveOutDeviceResult( result , msg2,__LINE__) )
        {
                result = waveOutClose(hwo);
                if( cheakWaveOutDeviceResult( result , msg2,__LINE__) )
                {
                }
                else
                {
                        possibleToGo = false;
                        setErrorFields( WAVEOUTDEVICE_GENERAL_CLOSE, result, __LINE__ );

                }
        }
```

```
        else
        {
          possibleToGo = false;
          setErrorFields( WAVEOUTDEVICE_GENERAL_RESET, result, __LINE__ );
        }

        DeleteCriticalSection(&cs);
}

void WaveOutDevice::setWaveFormatXHeader(WORD wBitsPerSample,DWORD nSamplesPerSec,WORD
nChannels)
{

        if( !possibleToGo ) return;

        wfx.wFormatTag = WAVE_FORMAT_PCM;
        wfx.nChannels = nChannels;
        wfx.wBitsPerSample = wBitsPerSample;
        wfx.nSamplesPerSec = nSamplesPerSec;
        wfx.nBlockAlign = (wfx.nChannels *  wfx.wBitsPerSample)/8;
        wfx.nAvgBytesPerSec =  wfx.nSamplesPerSec  *  wfx.nBlockAlign;
        wfx.cbSize = 0;
}


BOOL WaveOutDevice::isWaveOutDeviceCapable( void )
{


 WCHAR *msg1 = L"WaveOutDevice[IsWaveOutDeviceCapable[waveOutGet]]";
 WCHAR *msg2 = L"WaveOutDevice[IsWaveOutDeviceCapable[waveOutGetDevCaps]]";

 UINT  waveDeviceID;
 BOOL  returnValue = false;


 if( ! possibleToGo ) return (! possibleToGo);

 result = waveOutGetID(hwo,&waveDeviceID);

 if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
 {
        result =  waveOutGetDevCaps( waveDeviceID ,&woc, sizeof(WAVEOUTCAPS));
        if ( cheakWaveOutDeviceResult(result,msg2,__LINE__) )
        {

                // We need to compare if the device is ok to handle.
                // data that are coming.

                return true;

        }
        else
        {
                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_GETDEVCAPS, result, __LINE__ );
```

```
        }
}
else
{

        possibleToGo = false;
        setErrorFields( WAVEOUTDEVICE_GENERAL_GETID, result, __LINE__ );
}

return returnValue;
}

BOOL WaveOutDevice::cheakGeneralError( WORD errCode, WCHAR * msgBoxTitle)
{
        WCHAR *errMsg[] =
        {

        L"1:WaveDeviceOut:Device is not capable to satisfy req.",
        L"2:WaveDeviceOut:Cound Not Free Up memeory lpData",
        L"3:WaveDeviceOut:Cound Not Free Up memeory pwh",
        L"4:WaveDeviceOut:Cound Not Alloc Up memeory lpData",
        L"5:WaveDeviceOut:Cound Not Alloc Up memeory pwh",
        L"6:WaveDeviceOut:Data Size is negative"
        };

        BOOL returnValue = true;
        if(  (errCode >=1) && (errCode <= sizeof(errMsg)/(sizeof( WCHAR *)) ) )
        {
                MessageBox( hWnd, errMsg[ errCode-1 ] , msgBoxTitle, MB_OK);
                returnValue = false;
        }
        else
        {
                MessageBox( hWnd, L"Out of range", msgBoxTitle, MB_OK);
                returnValue = false;
        }
        return returnValue;
}

BOOL WaveOutDevice::cheakWaveOutDeviceResult( MMRESULT errCode, WCHAR * msgBoxTitle, DWORD
line)
{
        WCHAR errMsg[265];
        BOOL returnValue;
        DWORD result;
        WCHAR * msg1 = L"WOD[cheakWaveoutDeviceResult[waveOutGetErrorText]]";
        WCHAR * c_msg1 = L" Bad Error Number";
        WCHAR * c_msg2 = L" No driver ";
        WCHAR * c_msg3 = L" No memeory avaiable";

        switch ( errCode )
        {
                case MMSYSERR_NOERROR:

                                returnValue = true;
                                break;
```

```
                default:
                        result  = waveOutGetErrorText(errCode, errMsg, sizeof( errMsg ));
                        switch( result )
                        {
                                case MMSYSERR_BADERRNUM:
                                        wsprintf(errMsg,L"%s :[Line
                                          %d]",c_msg1,__LINE__);
                                        MessageBox( hWnd, errMsg , msg1, MB_OK);
                                returnValue = false;
                                break;
                                case MMSYSERR_NODRIVER:
                                  wsprintf(errMsg,L"%s :[Line %d]",c_msg2,__LINE__);
                                        MessageBox( hWnd, errMsg , msg1, MB_OK);
                                returnValue = false;
                                break;
                                case MMSYSERR_NOMEM:
                                        wsprintf(errMsg,L"%s :[Line %d]",c_msg1,__LINE__);
                                        MessageBox( hWnd, errMsg , c_msg3, MB_OK);
                                        returnValue = false;
                                        break;
                                default:
                                        wsprintf(errMsg,L"%s :[Line %d]", errMsg,line);
                                        MessageBox( hWnd, errMsg , msgBoxTitle,
                                         MB_OK);
                                        returnValue = false;
                                        break;
                        }

        }
        return returnValue;
}

BOOL WaveOutDevice::stillPlaying( void )
{
        EnterCriticalSection(&cs);

        if( (counterIn - counterOut) != 0)
        {
         LeaveCriticalSection(&cs);
         return true;
        }

  LeaveCriticalSection(&cs);
        return false;
}

int WaveOutDevice::setVolume( long lv, long rv )
{

        DWORD  value = 0L;

        WCHAR * msg1 = L"WaveOutDevice[setVolume[waveOutSetVolume]]";

        if( !possibleToGo )
        {
```

```
                return 0L;
        }


        EnterCriticalSection(&cs);

        lv &= 0x000000FF; // to make sure that lv is ok
        value = ( DWORD ) ( ( rv << 8 ) + lv);

        result = waveOutSetVolume(hwo, value);

        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
        {
                        }
        else
        {
                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_SETVOLUME, result, __LINE__ );
        }

        LeaveCriticalSection(&cs);
        return 0L;
}
int WaveOutDevice::getVolume( DWORD *value )
{
        WCHAR * msg1 = L"WaveOutDevice[setVolume[waveOutGetVolume]]";

        if( !possibleToGo )
        {
                return 0L;
        }

        EnterCriticalSection(&cs);

        result = waveOutGetVolume(hwo,(DWORD *)value);
        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
        {
                        }
        else
        {
                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_GETVOLUME, result, __LINE__ );
        }
        LeaveCriticalSection(&cs);
        return 0L;
}


int  WaveOutDevice::restartPlaying( void )
{

        WCHAR * msg1 = L"WaveOutDevice[restartPlaying[waveOutRestart]]";

        if( !possibleToGo )
        {
                return 0L;
```

```
        }

        EnterCriticalSection(&cs);

        result = waveOutRestart(hwo);
        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
        {
                // ok we set the value;

        }
        else
        {

                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_RESTART, result, __LINE__ );
        }
        LeaveCriticalSection(&cs);
        return 0L;

}

int  WaveOutDevice::pausePlaying( void )
{

        WCHAR * msg1 = L"WaveOutDevice[pausePlaying[waveOutPause]]";
        if( !possibleToGo )
        {
                return 0L;
        }

        EnterCriticalSection(&cs);

        result = waveOutPause(hwo);
        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
        {
                // ok we set the value;

        }
        else
        {
                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_PAUSE, result, __LINE__ );

        }
        LeaveCriticalSection(&cs);
        return 0L;

}

BOOL WaveOutDevice::playSound(LPSTR receivedData, DWORD receivedDataSize)
{

        WCHAR * msg1 = L"WaveOutDevice[playSound[waveOutPrepareHeader]]";
        WCHAR * msg2 = L"WaveOutDevice[playSound[waveOutWrite]]";
        WCHAR * msg3 = L"WaveOutDevice[WaveOutDeviceCallBack[LocalAlloc1]]";
        WCHAR * msg4 = L"WaveOutDevice[WaveOutDeviceCallBack[LocalAlloc2]]";
```

```
WCHAR * msg5 = L"WaveOutDevice[WaveOutDeviceCallBack[DataSize]]";

BOOL returnValue = true;
EnterCriticalSection(&cs);

if( receivedDataSize > 0L)
{

 if( !possibleToGo )
 {
        LeaveCriticalSection(&cs);
        return !possibleToGo;
 }


 pwh = (LPWAVEHDR) LocalAlloc(LMEM_FIXED,sizeof(WAVEHDR));

 if( pwh == NULL )
 {
        if( cheakGeneralError( 5, msg3) )
        {
        }
        else
        {
        }

       LeaveCriticalSection(&cs);
       return false;
 }
 zeroMemory((LPVOID )pwh,(DWORD) sizeof( WAVEHDR ));
 pwh->lpData = (LPSTR) LocalAlloc(LMEM_FIXED,receivedDataSize);

 if( pwh->lpData == NULL )
 {
        if( cheakGeneralError( 5, msg3) )
        {
        }
        else
        {
        }

        // make sure that it returns null i.e that is pwh is freed.
        LocalFree(pwh);
        pwh = NULL;
        LeaveCriticalSection(&cs);
        return false;
 }
 copyMemory(pwh->lpData,receivedData, receivedDataSize);

 pwh->dwUser = ( DWORD ) this;
 pwh->dwBufferLength = receivedDataSize;
 pwh->dwFlags = WHDR_BEGINLOOP | WHDR_ENDLOOP;
 pwh->dwLoops = 1;

 result = waveOutPrepareHeader(hwo,pwh,sizeof( WAVEHDR ) );
```

```
        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
        {
                result = waveOutWrite(hwo,pwh,sizeof( WAVEHDR ) ) ;
                if ( cheakWaveOutDeviceResult(result,msg2,__LINE__) )
                {
                        counterIn++;
                }
                else
                {
                        returnValue = false;
                        possibleToGo = false;
                        setErrorFields( WAVEOUTDEVICE_GENERAL_WRITE, result, __LINE__ );



                }

        }
        else
        {
                returnValue = false;
                possibleToGo = false;
                setErrorFields( WAVEOUTDEVICE_GENERAL_PREPAREHEADER, result, __LINE__ );
        }

       }
       else
       {
         cheakGeneralError(6,msg5);
       }

       LeaveCriticalSection(&cs);
       return returnValue;
}
BOOL WaveOutDevice::getError( struct WaveOutDeviceError * wode)
{

       if( errorCataogary == WAVEOUTDEVICE_GENERAL_NOERR )
       {
                return false;
       }
       else
       {
                wode->errorCataogary  = errorCataogary;
                wode->errorSpecefic   = errorSpecefic;
                wode->errorLineNumber = errorLineNumber;
                return true;
       }
}

void WaveOutDevice::WaveOutDeviceCallBack(HWAVEOUT hwo,UINT uMsg,DWORD dwUser,DWORD
dwParam1,DWORD dwParam2)
{

  WCHAR * msg1 = L"WaveOutDevice[WaveOutDeviceCallBack[waveOutUnprepareHeader]]";
       WCHAR * msg2 = L"WaveOutDevice[WaveOutDeviceCallBack[LocalFree1]]";
```

```
WCHAR * msg3 = L"WaveOutDevice[WaveOutDeviceCallBack[LocalFree2]]";

LPWAVEHDR pwh;
switch( uMsg )
{

case WOM_OPEN:
        break;
case WOM_DONE:

                        pwh = ( LPWAVEHDR ) dwParam1;
                        result = waveOutUnprepareHeader( hwo,pwh,sizeof(WAVEHDR));
                        if ( cheakWaveOutDeviceResult(result,msg1,__LINE__) )
                        {
                                if( LocalFree(pwh->lpData ) == NULL )
                                {
                                        if( LocalFree(pwh) == NULL )
                                        {

                                                // ok
                                                EnterCriticalSection(&cs);
                                                counterOut++;
                                                LeaveCriticalSection(&cs);
                                                pwh=NULL;
                                        }
                                        else
                                        {
                                                if( cheakGeneralError( 2, msg3) )
                                                {
                                                }
                                                else
                                                {
                                                }
                                        }
                                }
                                else
                                {
                                        if( cheakGeneralError( 3, msg2) )
                                        {
                                        }
                                        else
                                        {
                                        }
                                }

                        }
                        else
                        {

                                setErrorFields(
WAVEOUTDEVICE_GENERAL_UNPREPAREHAERDER, result, __LINE__ );

                        }
                        break;
        case WOM_CLOSE:
                break;
```

```
        }
}

void copyMemory( PVOID dest, const void *src, DWORD len)
{
register DWORD i = len;

if( len <= 0L ) return;
do {
                --i;
                *( ((unsigned char *)(dest))+i ) = *( ((unsigned char *)(src))+i );
        } while( i > 0L );
}

void zeroMemory( PVOID dest,DWORD len)
{
        register DWORD i = len;
        if( len <= 0L ) return;
        do {
                --i;
                *( ((unsigned char *)(dest))+i ) = 0;

        } while( i > 0L );

}
```

```
// WaveOutDevice.h: interface for the WaveOutDevice class.
//

#if !defined(AFX_WAVEOUTDEVICE_H__D5C67602_D3F4_11D2_A2CD_541C05C10000__INCLUDED_)
#define AFX_WAVEOUTDEVICE_H__D5C67602_D3F4_11D2_A2CD_541C05C10000__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#define MMRESULT DWORD

// #define BOOL bool

class WaveOutDevice;
struct WaveOutDeviceError
{

        DWORD       errorCataogary;
        DWORD       errorSpecefic;
        DWORD       errorLineNumber;

};

class WaveOutDevice
{


public:
        WaveOutDevice(HWND hWnd,WORD wBitsPerSample,DWORD nSamplesPerSec,WORD  nChannels);
        virtual ~WaveOutDevice();
        int pausePlaying( void );
        int restartPlaying( void );
        int playSound(LPSTR  receivedData, DWORD receivedDataSize);
        int stillPlaying( void );
        int setVolume( long , long );
        int getVolume( DWORD * value );
        int mute( void );
        BOOL getError( struct WaveOutDeviceError * wode);

        friend void CALLBACK WaveOutDeviceCallBackStub(HWAVEOUT hwo,UINT uMsg,DWORD
dwUser,DWORD dwParam1,DWORD dwParam2);


private:

                BOOL cheakGeneralError( WORD errCode, WCHAR *msgBoxTitle);
                BOOL cheakWaveOutDeviceResult( MMRESULT result, WCHAR *msgBoxTitle, DWORD line);
                BOOL isWaveOutDeviceCapable( void );

                void WaveOutDeviceCallBack(HWAVEOUT hwo,UINT uMsg,DWORD dwUser,DWORD
dwParam1,DWORD dwParam2);
```

```
                void setWaveFormatXHeader(WORD wBitsPerSample,DWORD nSamplesPerSec,WORD
nChannels);
                void setErrorFields( DWORD gen, DWORD spec ,DWORD line);


private:

        WORD                    nWaveOutDevices;
        HWAVEOUT                hwo;
        HWND                    hWnd;
        BOOL                    possibleToGo;
        MMRESULT                result;
        WAVEOUTCAPS             woc;
        WAVEFORMATEX            wfx;
        LPWAVEHDR               pwh;
        CRITICAL_SECTION        cs;
        DWORD                   counterIn;
        DWORD                   counterOut;
        DWORD                   errorCataogary;
        DWORD                   errorSpecefic;
        DWORD                   errorLineNumber;


};

#define         WAVEOUTDEVICE_GENERAL_NOERR                     1000
#define         WAVEOUTDEVICE_GENERAL_OPEN                      1001
#define         WAVEOUTDEVICE_GENERAL_CLOSE                     1002
#define         WAVEOUTDEVICE_GENERAL_RESET                     1003
#define         WAVEOUTDEVICE_GENERAL_GETID                     1004
#define         WAVEOUTDEVICE_GENERAL_GETDEVCAPS                1005
#define         WAVEOUTDEVICE_GENERAL_SETVOLUME                 1006
#define         WAVEOUTDEVICE_GENERAL_GETVOLUME                 1007
#define         WAVEOUTDEVICE_GENERAL_RESTART                   1008
#define         WAVEOUTDEVICE_GENERAL_PAUSE                     1009
#define         WAVEOUTDEVICE_GENERAL_PREPAREHEADER             1010
#define         WAVEOUTDEVICE_GENERAL_WRITE                     1011
#define         WAVEOUTDEVICE_GENERAL_UNPREPAREHAERDER          1012


#endif // !defined(AFX_WAVEOUTDEVICE_H__D5C67602_D3F4_11D2_A2CD_541C05C10000__INCLUDED_)
```

```
; Def file use in Visual C++ V 5.0
LIBRARY         ceWaveOutDeviceDLL
DESCRIPTION     'Native DLL for Playing Sounds on CEJVM written by Salim H. Omar'

EXPORTS
        RNIGetCompatibleVersion
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceRestart
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDevicePause
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceWrite
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceStillPlaying
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceSetVolume
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceClose
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceOpen
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceMute
        javacex_ceWaveOutDevice_ceWaveOutDevice_ceWaveOutDeviceGetVolume
```

Appendix B

Example of Using Object Mobility Toolkit

The following example demonstrates how the object mobility toolkit APIs can be used. We demonstrate the client side only since the server side is very similar Note that to understand the mechanism behind the achieving such distributed toolkit, a basic knowledge about Java and, particularly, Java Reflection APIs is a must.

The following code represents the client side, mobile device side. By running this code the necessary objects will be created and initialized to start the object server on the mobile side. The object server starts as a thread and keeps listening for the commands from the other object server remotely. The graphic user interface used here just to redirect the output consoles such as err and out of Java Virtual Machines. Not all implementation of Java Virtual Machines or operating systems provide a consol screen. For example WindowsCE does not provide a consol to be used as an output stream. The program demonstrates the use of APIs that is responsible for initialization process, creating objects and moving objects between client and the server Java Virtual Machines.

```
package ceDemo;

//
//
// MainProgram   Client side
//
//

import socketstream.*;
import ceuserinterface.*;
import objectmobility.*;
import shared.*;
import java.io.*;


public class MainProgram
{
  public static void main( String[] argv )
  {
        //
        // Create the GUI for displaying output of the program.
        //

        GUI  Gui = new GUI();
        Gui.setVisible( true );
        GUIService GuiService = Gui;

        //
        // Redirect the output to the GUI created before.
        //

        System.setOut( GuiService.getPrintStream() );
        System.setErr( GuiService.getPrintStream() );

        //
        // Leave error output to the default consol.
        //

        System.err.println("\nStarting Initialization");

        //
        // Create the action perform class that will implement
        // the required functionality of the demo
        //

        ActionPerform   actionperform = new ActionPerform();
        GuiService.setGUIActionPerform( actionperform );

    }
}


class ActionPerform  implements GUIActionPerform
{
```

```
//
// Get the run time class
//

Runtime          rt = Runtime.getRuntime();

Mobility                 mobilityService = null;     // mobility service
CommunicationLayer             cmml  = null;         // communication layer
RemoteObjectService              ros  = null;        // remote object service
ObjectA                         obja = null;        // the object that will be moved
                                                    // and being invoked locally
                                                    // or remotely.

InputStream                     spins  = null;      // Input stream from the socket
OutputStream                    spouts = null;      // Output stream from the socket

ActionPerform()
{
        System.out.println("1 Free  memory =  " + rt.freeMemory());
        SocketStream ss = null;

        try
        {
          // Create socket streams that the entire communications of the
          // the distributed toolkit will go through.

        ss  =  new SocketStream("134.117.57.167", 8000);

        //
        // Setting the delay between data chunks to simulate slow wireless channels.
        //
        ss.setDelay( 1 );

        //
        // The chunk size of data that will be shipped at once.
        //
        ss.setChunkSize( 1024000 );

        //
        // Create communication layer
        //

        cmml = new CommunicationLayer();

         //
         // Set the Input and the Output streams of this layer...
         //

          cmml.setINOUTStreams( ss.getInputStream(), ss.getOutputStream());

          // Create the Object Server that keeps listing to the coming commands from
          // the other remote servers.

          ros = new RemoteObjectService( cmml );
```

```
            //
            // Create the Mobility service that is responsible for moving objects
            // between servers.
            //
            mobilityService = new Mobility( ros );

            //
            // Initialize and start the Object server.
            //

            ros.setPriority( Thread.MAX_PRIORITY );
            ros.start();

          }
          catch( Exception e)
          {
              System.out.println("ActionPerform Exception :\n" + e );
              return;
          }

}

public void CreateObject()
{

            //
            // Create an object Called obja, which is a proxy to the real object
            // ObjectA.

            obja = ( ObjectA )AssigneProxy.assigne(obja,new ObjectA());
}

public void MoveObject()
{

          // We try to move the ObjectA created before
          // to the remote site and  get it back as well.
          long t1 = System.currentTimeMillis();

          if( mobilityService != null )
          {
                    if( obja != null )
                    {

                              switch( obja.getAssociatedObjectLocation() )
                              {

                                        case ProxyObject.REMOTE:
                                                  mobilityService.moveObjectTo( obja,
                                                  ProxyObject.LOCAL );
                                                  break;

                                        case ProxyObject.LOCAL:
                                                  mobilityService.moveObjectTo( obja,
                                                  ProxyObject.REMOTE );
                                                  break;
```

```
                    }

                }
                else
                  System.out.println("Please create the objectA first");
              }
              else
                System.out.println("Wait for the server to connect");

              long t2 = System.currentTimeMillis();
        }

        public void ExecuteMethod()
        {

              if( (obja != null) )
              {
                    try
                    {
                          long stime;
                          long etime;
                          int ncalls = 100;
                          double avg = ( double ) 0.0;
                          int result = 0;
                          stime = System.currentTimeMillis();
                          for( int i=0; i < ncalls ; ++i )
                          {
                           stime = System.currentTimeMillis();
                           result = obja.function1( i );
                           etime = System.currentTimeMillis();
                           avg += ( etime - stime );

                           System.out.println("sample time = " + ( etime - stime ));
                          }
                          System.out.println("Time/call in millis = " + ((( avg )/(ncalls))));

                }
                catch( Throwable e )
                {
                      System.out.println( "Exception in Execute method " + e );
                }

        }
        else
           System.out.println("Create object A first");
}

public void NullMethod()
{
              //
              // Here we test how the garbage collector will be invoked....
              // By trying to invoke the System.gc().

              obja = ( ObjectA )AssigneProxy.assigne(obja, null );
              System.gc();
}
```

```
        public void PrintROSTables()
        {

                // This mainly is used for debugging tables and reference counters
                // of the local object server

                if( ros != null )
                {
                        System.out.println("<<Objecta>> = " + obja );
                        ros.printTables();
                }

        }
}
```