KMOT : A MOBILE CODE TOOLKIT FOR RESOURCE-CONSTRAINED PORTABLE DEVICES

By

Yang Wang

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

Master of Computer Science

Ottawa-Carleton Institute of Computer Science
(OCICS)

Carleton University
Ottawa, Ontario
April 2001

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

KMOT : A MOBILE CODE TOOLKIT FOR RESOURCE-CONSTRAINED PORTABLE DEVICES

Submitted by Yang Wang B.S

In partial fulfillment of the requirement for
The degree of Master of Computer Science

Dr. Thomas Kunz, Thesis Supervisor

Dr. Frank Dehne, Director of School of Computer Science

Carleton University
April 23, 2001

Abstract

With the convergence of two technological developments, wireless communication and portable information appliances, a new paradigm of computing called mobile computing is becoming a reality. However, due to the intrinsic constraints of mobility such as small, slow, battery-powered portable devices, and variable low-bandwidth communication links, the design and deployment of non-trivial mobile applications are complicated. How to cope with these constraints is a hot research area as well as a demand of the PDA market, especially with the advent of the PalmPilot. One promising technique to address this problem is mobile code. Code mobility can make mobile applications adapt to the context changes and hence improve its performance on mobile devices with the aid of a proxy server.

In this thesis, we present our experiences from porting an existing mobile code toolkit for Windows CE (DMOT) to a new kind of emerging resource-constrained portable device, Palm IIIc. The new version of DOMT for this environment is called KMOT: KVM-based Mobile Object Toolkit. KMOT is designed as a platform for mobile code applications on WinCE and PalmOS. Its performance has been evaluated by several benchmarks, and hence we can conclude that, under certain conditions, mobile code is a feasible artifact to overcome the constraints in mobile computing, even for resource-constrained portable devices, and KMOT is a useful toolkit to realize the code mobility.

Acknowledgements

My graduate study would not have been possible without the technical and moral support of many people. First and foremost, I would like to sincerely thank my thesis supervisor, Dr. Thomas Kunz, for providing immeasurable guidance and enthusiastic encouragement over the course of my thesis research. He introduced me the field of mobile computing, and provided me with many opportunities for both personal and professional growth: reading and writing research proposals; reviewing technical papers; writing and presenting conference papers; and programming experiments to validate ideas. These opportunities opened a door for me in the research community, and make me perceive that academic research is an enjoyable thing in life. I consider myself fortunate that I had access to his valuable supervision.

This work was financially supported by the Bell Mobility, which I gratefully acknowledge to provide me with the funds with which to complete my graduate study. I am also grateful to the staff members of the Department of Systems and Computer Engineering who gave me the administrative and technical support to help finish my thesis. In particular, I wish to thank Salim Omar for freely sharing with me his considerable experience in Java and DOMT, and Thomas Barry for patiently setting up the environment. Their work made my research start up quickly and proceed smoothly.

I will have fond memories of my days in Carleton University due to not only its beautiful campus but also my good friends. They always bring me happiness in life, and teach me how to take care of myself when I live alone. The great time together with them

let me know in addition to storm, life still has sunshine. I will remember their name no matter where I go: Shixin Wei, Lei Tan, Ying Wang, Hua Guo, Yan Cui, Ping Li, Angle Tao, Chuncai Yang, Maoyu Wang, Liang Qin, Xuejun Xu, Chen Huang, Xuqing Wu, and Zhaoshu Zeng. Thank you.

Last but not least, I wish to thank my family for their constant moral support. My mother is always confident of her son, and keeps me motivated to achieve new goals of life continuously. Her support has been a source of encouragement to me. My wife did the best for me only in a short period during my graduate study due to her health. I will treasure her love. This thesis is also dedicated to the memories of my father whose optimistic and positive attitudes to life always inspire me to face the difficulty confidently.

Contents

List of Tables

List of Figures

Chapters

List of Tables

List of Figures

Chapter 1

Introduction

With the convergence of two technological developments, wireless communication facilities and portable information appliances, a new paradigm of computing called mobile computing is made a reality. Mobile computing is distinguished from classical, fixed-connection computing by the characteristics of many resource constraints such as small, slow, battery-powered portable devices, and variable low-bandwidth communication links. These constraints are not artifacts of current technology, but are intrinsic to mobility. They complicate the design of mobile information systems and require rethinking traditional approaches to information access and application design. In order to overcome these challenges, various computing paradigms and techniques are proposed and investigated. In this chapter, we will introduce some basic concepts and issues pertaining to mobile computing and define the goal of the thesis.

1.1 Wireless Data Networking

Wireless networking technologies allow the users carrying portable computers to access the capabilities of the global network at anytime without regard to their location or mobility. These technologies are highly application-oriented. Circuit and packet switched

communications are the two networking alternatives. Each one of them is provided through a variety of technology offerings.

1.1.1 Wireless Network for Circuit-Switched Data

Cellular networks [39] and cordless telephony [39] are two main current systems utilizing circuit switching technology. Although voice was the first major application for these systems, they have evolved to provide data circuits through the wireless infrastructure. Cellular networks such as GSM [34], TDMA, and CDMA [10] are suitable for wide-area mobility which features slow and unreliable wireless data transmission, whereas cordless systems, for instance CT2 [39], or DECT [7] can be applied to local-area mobility such as conference rooms, university campuses, etc. with relatively high-speed data transmission. Circuit switching technology is optimized for isochronous data traffic transmission and hence, suitable for applications with a large amount of data to be transferred smoothly.

1.1.2 Wireless Network for Packet-Switched Data

Circuit-switched data is a somewhat selfish way of using limited resources. On the contrary, a wireless packet-switched data service is designed for sharing both radio resources and network resources. There are mainly two alternatives: mobile data network [43] and wireless local area networks (WLANs) [39]. They both employ packet switching technology and are ideal for asynchronous data traffic transmission, which is the characteristic of applications with short, bursty data transmission patterns like in web surfing.

Mobile data network can be generally characterized as providing high mobility, wide ranging communication to the users. The typical systems are ARDIS [43], CDPD [21], and the emerging GPRS [43]. These technologies provide economical means for the

realization of mobile computing. However, due to physical layer constraints, these systems suffer from low-speed data transmission, typically on the order of 9600 b/s. In order to overcome this barrier, the combination of GPRS and EDGE (enhanced data rates for global evolution) promises to improve the utilization of the radio network and provide the potential for a whole range of mobile multimedia services in such fields as Internet and Intranet.

Wireless local-area networks (WLANs) can be viewed as providing low-mobility high-speed data communications within a confined region, e.g., a campus or large building. Although WLANs have been evolving for a few years, the success of efforts to standardize them is very limited. The market is active with various products whose data rate range from hundreds of kb/s to more than 10 MB/s depending on link technologies. Roughly speaking, these technologies can be divided into two categories: using radio frequency or using infrared. Radio frequency technology recently witnessed a new advance called Bluetooth. There are two overall network architectures for designing of WLANs. One is a centrally coordinated and controlled network in which base stations exercise overall control over channel access. The other type is the self organizing and distributed controlled network where all mobile devices have the same function and networks are formed ad-hoc by communication exchanges among mobile devices. Typical WLAN systems are Motorola's Altair, NCR's WaveLAN, Proxim's RangeLAN, WinDATA's Freeport, and Cabletron's Freelink.

1.2 Architecture of General Mobile Environment

The wireless architecture partially inherits the cellular concept, adopting fixed backbone networks extended with a number of mobile hosts (MH). The fixed backbone connects fixed hosts called Mobile Support Stations (MSS), or Base Stations (BS). The radio coverage area of an individual base station is called a cell which could be a real cell as in cellular communication networks or a wireless local area network. Figure 1.1 displays the architecture of a wireless network [22].

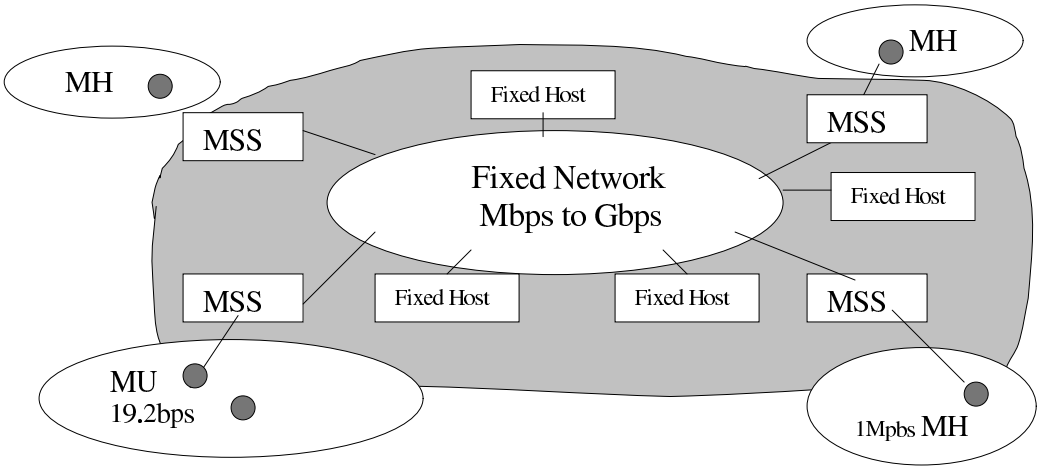


Figure 1.1 Architecture of a Wireless Network

Wireless networking allows the roaming mobile hosts to communicate with other units, mobile or fixed, only through the base station of the cell in which it resides. As a mobile host moves, it may cross the boundary of its current cell and enter a new cell covered by a different base station. This process is called handoff, which can transfer a call in progress on a radio channel automatically to a new radio channel without interruption to the call. However, management of the handoff process in data transmission is non-trivial due to the limited tolerance to information loss.

1.3 Personal Digital Assistants

Since the invention of computers, their capability, size and form factor experienced dramatic changes from mainframe to desktop to palmtop, accompanied by increasing performance and decreasing cost. Personal digital assistants (PDAs) are one of the great successes along these lines. A PDA is effectively a handheld PC, capable of handling all the normal tasks of its leather-bound ancestors—address book, notepad, appointments diary, phone list as well as other applications such as spreadsheet, word processor, database, financial management software, clock, calculator and games. With equipment for wireless connectivity, PDAs have achieved another giant leap in their capacity since it is no longer necessary for them to connect to the PC or laptop through cables and cradles. Synchronization can be achieved through an infrared port or new radio technologies such as Bluetooth. This capacity can be also extended to communicate with WLANs, inducing an affordable reality to many powerful applications represented by PalmPilot in the mobile application market.

PalmPilot enables mobile users to manage their critical personal and business information on their desktop and remotely. They obtained market leadership with their distinguished features including shirt-pocket size, an elegant graphical user interface, and an innovative desktop docking cradle which facilitates two-way synchronization between the PC and the PDA. With the launch of the Palm VII devices, which have the capability of wirelessly accessing the Internet, great interest in mobile computing in the context of PalmPilot has arisen in both the industrial and the academic community. Currently, several well-known projects such as TACOMA [24] and Grasshopper [18] have been migrated or are in the process of being migrated to the PalmPilot platform.

1.4 The Challenges of Mobile Computing

Wireless networking technologies, together with portable devices set up the fundamental structure to mobile computing, but also pose new challenges. The principal challenges faced by a mobile application stem from three essential properties of mobile computing.

- Wireless Communication

The common property of wireless technologies we discussed above is that the signal or data has to go through an air space with various barriers, which may interact with the signal, block the signal path and introduce noise and echoes. These factors may result in weak connectivity characterized by lower bandwidths or high bandwidth variability, higher error rates, and more frequent spurious disconnections. As a consequence, the quality and performance of wireless connections is much worse than that of wired connections due to the techniques adopted to overcome these difficulties such as retransmissions and error control protocols.

- Portability

In addition to the weight and limited input systems, portable devices, especially like the PalmPilot, still have many other constraints: low power, hightened risk of data loss, small user-interfaces, and limited on-board storage. For example, the Palms V weights 115g at a size of 115mm x 77mm x 10mm and at the same time is equipped with a 160x160 pixel backlit screen. These limitations are inherent properties of portable devices, and add a new dimension to the design of applications for mobile devices.

- Mobility

Mobility is a unique feature in mobile computing. This advantage introduces other challenges: the whole configuration of the system, including fixed and mobile hosts,

changes dynamically. Assigning a fixed topology is no longer valid in mobile computing. Location management adds extra cost to locate mobile elements when computing is in progress. Connectivity becomes highly variant in performance and reliability, the number of devices in a network cell changes with time, and so do both the load at the base station and bandwidth availability.

These challenges require mobile application designers to rethink the traditional design approach to reflect the new constraints. For example, mobile applications should minimize dependence upon data obtained over such limited, unreliable connections and data stored in limited, unreliable resources on a mobile host. Finding approaches to overcome these challenges and to improve application performance is a vital and interesting problem. The techniques proposed range from system level to application level.

Data hoarding [23] and lazy write-back are simple system-level support for disconnected operation. The principle of this technique is that when a network disconnection is anticipated, data items and computation are moved to the mobile client to allow its autonomous operation during disconnection. Upon reconnection, updates performed at the mobile hosts are reintegrated with updates performed at other sites. Update reintegration is usually performed by re-executing the log at the fixed host. Data hoarding indicates that the more autonomous a mobile client, the better it can tolerate network disconnection. A critical issue in this technique is how to anticipate the future needs of the mobile unit for data. One approach is to allow users to explicitly specify which data items to hoard. Another approach is to use information about the past history of data access to predict the future needs for data. Another technique to hide both round-

trip latency and short disconnection is operating asynchronously. In asynchronous operation a client sends multiple requests before asking for any acknowledgment. These techniques have the potential to mask some network failures. Low bandwidth can be increased effectively by certain software techniques. For example, filtering and compressing the data stream between a client application on a portable device and a server executing on a stationary host sometimes almost double the throughput [2, 13, 25, 55]. This filtering and compression are generally realized at the system-level by employing a "split-TCP" approach based on the client-proxy-server model which will be discussed later, and hence is transparent to the application-layer protocol (such as HTTP). However, this technique also suffers from an important flaw. The mobile client has to recover the transformed data stream by consuming some resources. For example uncompressing the data stream will consume CPU cycles and battery power, which will degrade the performance and undermine portability—users may have to recharge frequently. Another set of system-level techniques provides an end-to-end approach, where "Wireless Logical Link Control (WLLC) [51] " is designed to deal with the specificities of the wireless link. WLLC resides directly below IP with the aim at reducing the error rates on the wireless link. It has been shown [14] that the interaction of those mechanisms in the link layer (here, WLLC) and the transport layer (here, TCP) becomes beneficial if the error rate of the network exceeds a given threshold. The advantage of the end-to-end approach is that it does not imply any change in semantics of any of the protocols in the TCP/IP protocol suite. However, it must be also recognized that the use of error correction in the link layer will introduce variable delays, which will be observable by the application. High bandwidth variability can be approached at the

application level by adapting to the currently available resources, providing the users with a variable level of quality of service.

Power consumption is another main concern in mobile computing since the battery size and weight have to be limited to some small range for easy portability. There are several important approaches from hardware to software to save power [17]. (1) Battery size can be reduced by greater levels of VLSI integration and multichip module technology. (2) Voltage can be reduced by redesigning chips to operate at lower voltages. (3) Clock frequency can be reduced by trading off computational speed for power saving. Other power-saving methods come from management software which can power down individual components when they are idle, for example, spinning down the internal disk or turning off screen lighting, even leading the computer into a doze mode in which clock speed is reduced and no user computation is active. Doze mode will return to normal operation upon receipt of a message.

Coping with limited storage is not a new problem in mobile computing. Solutions include compressing file systems, accessing remote storage over the network, sharing code libraries, and compressing virtual memory pages [16]. However, due to network disconnections, these network-dependent technologies are less appropriate for mobile computers. A novel approach to reducing program code size is to interpret script languages, instead of executing compiled object code, which are typically many times the size of the source code. The typical examples are General Magic's Telescript and Apple Technology Group's Dylan and NewtonScript. An equally important goal of such language is to enhance portability by supporting a common programming model across different platforms.

Although these techniques attack some inherent challenges of mobile computing, they do not address the problem of deploying a non-trivial application on the resource-constrained mobile devices. Here, non-trivial applications refer to those which have a lot of CPU-bound work as well as require a large address space to execute. These applications are typically power consuming, such as a MP3 player. In order to achieve this goal, offloading computationally intensive application components from portable devices to more powerful servers in the access network is a promising approach. This kind of computation migration results in an adaptive application whose efficiency has been demonstrated in [1, 25, 30].

1.5 The Adaptive Approach

In mobile computing, adaptation means mobile applications need to take advantage of the changing availability of resources in the mobile environment to adjust their behavior, so as to maintain the semantics of the application for the user, and at same time achieving better overall performance. Due to the different nature between the wired and wireless network, an adaptive application is usually built based on a new kind of computing paradigm known as Client-Proxy-Server model to keep the application server in the access network unchanged.

1.5.1 The Paradigms of Mobile Computing

The Client-Proxy-Server Model is considered commonplace in today's heterogenous network environments. This model is generated by adding another component to the traditional Client-Server Model. It takes the form of an intermediary placed between two communicating end-points such as a client and server. The purpose of the intermediary is

to improve the quality of the network as perceived by the client along some dimension which is constrained and blur the boundary of functionality between client and server. Hence, this architecture somewhat alleviates the impact of the limited bandwidth and the poor reliability of the wireless link. The typical functionality at the proxy includes support for messaging and queuing for communication between the mobile client and the proxy, moving some responsibilities from the client to the proxy. The proxy can also cache data to minimize long round trips on the network and thus reduce application response time.

However, this model also has an important drawback, the communication between a mobile host and a correspondent host in the access network is no longer "direct contact", since they are interacting through a "relay" (i.e. the proxy host). This invalidates the end-to-end semantics of a transport protocol, and may jeopardize some application layer protocols like ftp or rlogin, which assume end-to-end reliability from TCP. In addition to this, security is another problem to proxies, unless authoring mechanisms are introduced.

1.5.2 DOMT: Our Motivation and Approach

Adaptation has different targets ranging from those attempting to alleviate the effects of congestion and weakly connected operations to User Interface Management System which attempt to adapt to the display capabilities through battery management systems within the operating system. Adaptation can be realized in different components of the systems with different technologies. The rise of middleware has generated interest in providing a generic adaptation architecture and mobile code techniques provide a feasible approach to make it a reality. DOMT [30] is a mobile code toolkit implemented as a middleware based on Java technology for WinCE to reflect the confluence of these two

technologies. It adopts the application-aware strategy to handle the mobile characteristics of the environment, and employs the mobile code technique to support the design of adaptive applications. The fundamental mechanisms provided by this toolkit are dynamic object migration and remote method invocation. The key task of the programmer in building such applications with DOMT is to define movable objects. The toolkit runtime system can dynamically divide the program into portions that run on the mobile client and portions that run on the proxy server according to the current environment parameters. The two parts communicate by means of remote method invocations.

DOMT is a promising approach to support the building of adaptive applications. However it does not run on the resource-constrained portable devices like PalmPilot. We noticed that this phenomenon also happens in many projects like Coda, Odyssey, Rover and BARWAN which are designed for supporting adaptive applications on various mobile platforms, including PDAs. Only very few studies [24] focused on building applications on PalmPilot and identify the benefits and tradeoffs involved. As a consequence, a wider acceptance of adaptive application based on code mobility is presently hampered by the fact that the soundness of the abstractions and mechanisms proposed is not verified by quantitative evaluations and experimental evidence. Hence, our motivation is twofold. First, we build a mobile code toolkit called KMOT (KVM-based Mobile Object Toolkit) for these portable devices. Second, based on this toolkit, we try to evaluate the efficiency of code mobility of an application between a resource-constrained portable device and its powerful proxy servers. Our approach is to extend DOMT to PalmPilot devices with KVM as a porting platform. KVM is a scaled-down version of Java Virtual Machine for resource-constrained portable devices. However, due

to the limitations of the KVM technology, some fundamental functions and mechanisms used by DOMT are not provided, and the architecture adopted by DOMT cannot be reused directly under the scarce memory of PalmPilot device. Therefore, KMOT is far from a simple compile-and-run portability exercise.

1.6 Contributions of the Thesis

The contributions of the thesis can be summarized as follows:

- Extending DOMT to PalmPilot platform as KMOT, in which the mechanism to support object migration, construction of distributed object graph and remote method invocation are provided. This mechanism is based on Java Serialization and Class Reflection which are not present in KVM.
- The architecture of DOMT is adjusted for KMOT. A new implementation method, i.e., Distributed Recursive Method, for Nested Method Invocation is realized which can simplify this kind of method invocation on resource-constrained mobile devices.
- A performance evaluation of dynamic object migration between resource-constrained portable devices and the proxy server is presented.
- Early insights "KMOT: A Mobile Code Toolkit for Resource-constrained Portable Devices" are published in Proceedings of the Symposium on Software Mobility and Adaptive Behavior, pp 89-97, York, United Kingdom, March 2001.

1.7 Thesis Road Map

There are 7 chapters including this one in the thesis. They are organized as follows. In Chapter 2 we survey technologies to support adaptive application design. One promising technique is mobile code, which can make mobile applications adapt to the context changes dynamically and hence improve the application performance on mobile devices with the aid of a proxy server. In Chapter 3, we give a brief overview of the DOMT Toolkit's abstractions, architecture, and its fundamental functionality. DOMT is our starting point to realize KMOT. Chapter 4 outlines the design and implementation of KMOT including its fundamental support and its architecture as well as its limitations. Chapter 5 provides an evaluation of KMOT. Experimental setup and results are presented. Chapter 6 discusses the related work including some well-known mobile-aware adaptive systems such as Coda, Odyssey and Rover as well as other mobile code systems. Finally, Chapter 7 draws our conclusion and raises some problems for future work. Appendix A describes our further research on the strategies of scheduling objects between the mobile client and its proxy server so as to improve the application performance.

Chapter 2

Background

2.1 Introduction

Due to the well-known characteristics of mobile computing, we need the applications to adapt to the changing constraints of the resources available in the environment in a quick manner. In order to achieve this goal, one of the approaches requires that the computation and communication between the client and its proxy server can be traded off in the progress of computing. By moving computational components from the mobile device to its powerful server, or vice versa, the communication cost such as the bandwidth requirement, and the battery drain can be reduced dramatically. A component in an object-oriented program can be a group of objects to accomplish some task. Component movement can be done in an offline or online manner, depending on the application properties. The offline approach requires the application designer to figure out the program components to reside at different sites prior to their execution. This approach will naturally result in a distributed object computing system. CORBA [48], DCOM [6] and Java RMI [54] are typical object-based distributed systems to realize this approach. The online approach is more flexible than its offline counterpart since it can ship the components during the program execution, and hence can adapt to the mobile

environment more gracefully. The effect of this approach is well-exemplified in mobile code techniques.

In this chapter, we discuss some background knowledge to realize adaptive application in the mobile environment.

2.2 Distributed Systems

Performing a computation on top of networked computers is called a distributed computation, since they are not physically resident in the same host and have autonomy. This distribution enjoys many advantages such as collaboration through connectivity and networking, performance improvement through parallel processing and extensibility through dynamic configuration and reconfiguration. During the last decade, object technology gained wide acceptance, favoring its characteristics of abstract data typing, inheritance and polymorphism. The composition of these two techniques created a new research area called Distributed Object Computing (DOC) [15] based on a client-server architecture. These techniques made life easier for platform developers and application developers. In this subsection, we first discuss the common characteristics of object-based distributed system and then look at some concrete systems including the Common Object Request Broker Architecture (CORBA) defined by Object Management Group (OMG), and Java based solutions such as JavaRMI. Finally, we briefly introduce the Open Group's Distributed Computing Environment (DCE) model[38], which is deemed a competitor to CORBA.

2.2.1 Distributed Objects

An object-oriented program consists of a collection of interacting objects which are entities encapsulating data and methods to process them. Objects communicate with each other by sending and receiving messages. Each object has an identifier in the underlying object system; for example, the value of a Java object reference is an object identifier or OID. A class describes a potentially infinite set of similar objects and can be used as types for defining parameters and results in signatures. Multiple inheritance allows a class to make use of the code of several other classes. Java does not provide multiple inheritance. But it does allow classes to implement several *interfaces* as well as inheriting from one class. In Java, an interface is an abstract definition of the signatures of a set of methods. The method executed is chosen according to the class of the recipient of the message. This is called *dynamic binding*.

Objects in distributed systems are called distributed objects, which have some unique features. First, the distribution of objects in different address spaces or physical sites will result in remote method invocations (RMI), provided that the class implementation is available. Second, an object may be accessed concurrently by more than one remote or local objects, and hence the possibility of conflicting accesses must be addressed. However, the fact that the data of an object is accessed only by its own methods allows objects to provide methods for protecting themselves against incorrect accesses. For example, they may use synchronization primitives such as condition variables to protect access to instance variables. The difficulty of the first problem is that RMI should be transparent. That is, an object should be able to send a message to another object and to receive a reply without being aware of whether the receiver is local or remote. This

transparency may be achieved by providing a local proxy for each remote object that can be invoked by a local object. The role of a proxy is to behave like a local object towards the message sender, but instead of executing the message, it forwards the message to the remote object. The remote object performs the message and replies without being aware that its reply is sent back to a sender on a remote site. A remote object has a 'skeleton' object whose class has the server stub procedures as its methods. The classes for the proxy and the skeleton used in RMI are generated automatically by an interface compiler like the client and server stub procedures in RPC [5].

Distributed object systems may adopt the client-server architecture. In this architecture, objects are managed by servers and their clients invoke their methods through proxies. Objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance. The Clouds system and Shapiro's Larchant system [46] use distributed shared memory to replicate objects at the point of use, whereas the Emerald [32] system experimented with object migration with a view to enhancing the performance and availability of objects.

2.2.2 CORBA

The CORBA specification defines an abstract object model similar to the one described in the previous subsection. It includes an IDL (interface definition language) to provide facilities to define interfaces, types, attributes and method signatures. Each remote object has an IDL interface specifying the methods that may be requested by clients. An IDL interface compiler generates client stubs in the language of the client and IDL skeletons in the language of the server. The CORBA architecture is designed to support an Object Request Broker (ORB) that enables clients to invoke methods in remote objects that have

been implemented in a variety of languages. These invocations are realized by the cooperation between the client stubs and server skeletons for marshaling, transmitting, and unmarshaling the parameters and the results, and hence the language-neutral property is achieved. A server in CORBA is a process that is executing the implementation of one or more remote objects. These objects are allowed to become clients of other remote objects, thus enabling clients to perform invocations that cause chains of related actions on distributed objects. The Object Adapter deals with everything that a client needs at run time in order to invoke a method in a remote object such as the registering object implementation in an implementation repository, activating an object implementation in a server process whenever a client needs it, and registering the servers currently offering activated objects and accessing them as needed by clients. CORBA extensions are also enriched to provide transactions or some form of concurrency control and recovery. It also provides some form of object replication, which facilitates the effectiveness of interactive programs.

CORBA does not state anything about the semantics of remote objects or how they are implemented because it was designed to provide services based on existing software. Client software can be designed with caches, but the design of a cache is difficult when the semantics of server objects are unspecified. CORBA objects are generally fairly large-grain objects due to the considerable performance overhead for RMI, and therefore, object migration and object to be passed by value are not supported by CORBA. On the other hand, the language-neutral nature limits the kinds of data to be transmitted to the basic data types that can be represented in all the target languages. Furthermore, in object-oriented terms, no polymorphism is allowed—the transmitted object's type (or its

reference type) cannot be a subtype of the type expected by the skeleton. This requires that the receiving process know exactly what the sending process places on the wire.

Although CORBA provides distributed applications on heterogeneous systems, its current technology is not designed for use in a mobile computing environment. First, its specification, including Inter-ORB protocol (IIOP) does not indicate whereabouts to address the mobility of the clients or the servers. Second, its implementations are not typically built upon micro-kernel architecture. This makes it difficult to modify the CORBA runtime down to its bare essentials, which is important for embedded system development. Finally, its implementations do not give developers low-level control over the management of system resources like heap allocation. All these limitations prevent CORBA from deploying on mobile devices to support adaptive applications.

To better meet the need of adaptive applications and the mobile system programmers, CORBA specifications and implementations need to be redesigned.

2.2.3 JavaRMI

Although on the surface, the RMI system is just another RPC mechanism, much like CORBA, it represents a very different design philosophy, one that results in a system that differs not just in detail but in the very set of assumptions made about the distributed systems in which it operates. These differences lead to differences in the programming model, capabilities, and way the mechanisms interact with the code that implements and builds the distributed systems.

As we described above, CORBA was built on assumptions of heterogeneity. These mechanisms assume that the distributed system contains machines that might be different, running different operating systems. However, the Java RMI system is built on an

entirely different set of assumptions. Heterogeneity is not the major problem since the client and the server are both Java classes running in the Java virtual machine, which makes the network a homogeneous collection of machines. With this single-implementation language assumption, the RMI system does not need a language-neutral IDL and any Java object is allowed to pass as a parameter or return value in a remote call. Remote objects pass by reference, in effect, by passing a copy of the object's stub code. Non-remote objects are passed by value, creating a copy of the object in the destination. The objects that pass are real objects, not just the data that makes up an object's state. Unlike CORBA, this distinction realizes the polymorphism, which means the skeleton can receive a subtype of the type it expects, and hence it is not necessary for the receiving process know exactly what the sending process places on the wire in advance. This property provides flexibility to design of more powerful applications. RMI uses a variant of the Java Object Serialization protocol to marshal and reconstruct objects. Unlike CORBA whose stub code is linked ahead of time into the client, Java stubs for remote objects originates with the object and are loaded at runtime when needed. This approach allows programmer using the system to build a variety of "smart" proxies. Such smart proxies can cache certain values in the stub, avoiding the need to make remote calls in certain cases, or can be extended with more logic to move its associated remote object to the point of use, enhancing the performance. This is a desired property to realize adaptive applications.

2.2.4 DCE

DCE, short for Distributed Computing Environment, is a technology that is a bit older than CORBA, but more stable and scalable. Not only does it offer the basic RPC

mechanisms and IDL stub compilers needed to create distributed applications, but it also offers security and authentication through Kerberos (an area that still exhibits weakness in CORBA). Unlike CORBA, it is not a ground-up redesign of the principles of distributed computing, but is rather a tightly integrated package of existing technologies. The DCE architecture is a layered model that integrates a set of fundamental technologies bottom up from the most basic supplier of services (e.g., the operating system) to the highest level consumers of services (e.g., applications). To applications, DCE appears as a single logical system, which can be organized into two broad categories of services: the DCE Secure Core and DCE Data Sharing Services. The DCE Secure Core services give software developers the tools such as multithreading, RPC to create end-user applications and system software products for distributed computing. DCE Data Sharing services facilitate better use of shared information and require no programming by the end-user. The main component is the DFS distributed file system, which is a high-performance, scaleable, secure method for sharing remote files. DFS appears to the user as a local file system, providing access to files from anywhere in the network for any user, with the same file name used by all (i.e., uniform file access).

Although DCE can provide similar functions to the CORBA, the fundamental difference between them is that DCE was designed to support procedural programming, while CORBA was designed to support object-oriented programming.

2.3 Mobile Code System

We are concerned with the techniques to support adaptive applications. The distributed systems like CORBA or JavaRMI achieve this goal by balancing the workload among

machines in the systems statically prior to execution. They focus on static distributed objects rather than dynamic mobile objects. Therefore, this approach does not ensure the degree of flexibility, customizability, and reconfigurability needed to cope with the challenge of adaptive applications. A different promising approach is to exploit the notion of mobile code. Code mobility can be defined informally as the capability to dynamically change the bindings between code fragments and the location where they are executed [9]. The ability to relocate code is a powerful concept since it implies that computation migration becomes a reality. Computation migration provides us with the opportunity to trade off the cost between the communication and computation, and hence allows adaptive applications to react to the changes of the mobile environment dynamically and gracefully.

2.3.1 Execution Model

The execution model for code mobility, well defined by Gian Pietro Picco [9], can be abstracted into several components. The first is the Computational Environment (CE) which resides upon the network operating system. The purpose of the CE is to provide applications with the capability to dynamically relocate their components on different hosts. The components may be executing units (EU) or resources. Executing units represent the computational elements of the model and may be modeled as the composition of a code segment and a state. Single-threaded processes or individual threads of a multithreaded process are typical examples of EU. Resources are passive entities representing data, such as a file in a file system, an object shared by threads in a multithreaded object-oriented language, or an operating system variable. The code segment provides the static description for the behavior of a computation, and a state can

be decomposed into a data space and an execution state. The data space is the set of references to resources that can be accessed by the EU. The execution state stores private data that cannot be shared, as well as control information related to the EU state, such as the call stack and the instruction pointer.

This model gives us a basis to identify the various mobile code mechanisms and classify the existing mobile code systems.

2.3.2 Mobile Code Mechanisms

In conventional systems, each EU is bound to a single CE for its entire lifetime. Moreover, the binding between the EU and its code segment is generally static. Even in environments that support dynamic linking, the code linked belongs to the local CE. However, in mobile code systems, the code segment, the execution state, and the data space of an EU can be relocated to a different CE. These components can be moved independently. There exist two forms of mobility, characterized by the EU constituents that can be migrated. Strong mobility is the ability of a mobile code system to allow EUs to move their code and execution state to a different CE. Weak mobility is the ability of a mobile code system to allow code transfer across different CEs; code may be accompanied by some initialization data, but no migration of execution state is involved.

Strong mobility is supported by two mechanisms: migration and remote cloning. The migration mechanism suspends an EU, transmits it to the destination CE, and then resume it. This mechanism can be either proactive or reactive, depending on whether the migration is determined autonomously by the migrating EU or not. The remote cloning mechanism creates a copy of an EU at a remote CE, but the original EU is not detached from its current CE. As in migration, remote cloning can be either proactive or reactive.

Mechanisms supporting weak mobility provide the capability to transfer code across CEs and either link it dynamically to a running EU or use it as the code segment for a new EU. This mechanism can be either synchronous or asynchronous, depending on whether the EU requesting the transfer suspends or not until the code is executed.

In both strong and weak mobile code systems, upon migration of an EU to a new CE, its data space, i.e. the set of bindings to resources accessible by the EU, must be rearranged. Two classes of strategies are possible: replication strategies and sharing strategies. Replication strategies can be further divided in static replication strategy and dynamic replication strategies. In static replication strategy, some resources such as system variables can be statically replicated in all CEs. The original bindings to such resource are deleted and new default bindings are established with the local instances on the destination CE. Dynamic replication strategies allow copies of the bound resources to be established dynamically in the destination CE, the original bindings are deleted, and new bindings are established with the copied resources. Sharing strategy implies that the original binding is kept and therefore inter-CE references to remote resources must be generated. Mobile code system may exploit different strategies for different resources, depending on their properties.

2.3.3 Mobile Code Paradigms

Carzaniga, Picco, and Vigna [9] decomposed distributed applications into several components and classified the interactions between the components to provide a description of mobile code design paradigms. These paradigms are categorized into Client/Server (CS), Remote Evaluation (REV), Code on Demand (COD), and Mobile Agent (MA) paradigms. The components used here consist of resource components such

as code and data, computation components such as thread of execution. The interactions indicate the event and information passing between two or more components. In these paradigms, distribution execution can be modeled as primitives operating in one of the above mobile code scenarios.

The CS paradigm is well-known and wildly used. In this paradigm, the computation components as well as their resources such as code and some data are kept fixed at the client and server sites respectively. The client computation component requests the execution of a service with an interaction with the server's corresponding part. This interaction may ship some data as input parameters, and as a response, the server performs the requested service and produces some sort of result that will be delivered back to the client with an additional interaction. This is the usual RPC style of programming.

The REV paradigm can be viewed as an extension of the CS paradigm by not only shipping the data but also shipping the code to the remote server for executing. Although they are very similar in nature, the additional shipping of code will pose more requirements on the runtime system of procedural languages, which are always used in CS programming.

The COD paradigm is an inversion of REV. Instead of sending code and data to the remote server for execution, the client requests code and data from the server, and executes it locally. An example of this form of code mobility is the applet service in Java programming language.

Finally, MA is the most dynamic and autonomous of the above paradigms. Unlike CS, REV, and COD, whose code and data component can be transferred from one site to

another, but computation component remains fixed to their original sites, MA can move not only its resource components, but also its entire computation component along with its state.

The mobile code paradigms define a number of abstractions for representing the bindings among components, locations, and code, and their dynamic reconfiguration. Unlike the traditional computing paradigms, these bindings are dynamic in nature. This property suggests that these abstractions are effective in the design of adaptive mobile applications, which require a mechanism to react to the changes of the mobile environment. Mobile code paradigms can achieve this goal by providing component mobility. By changing their location, components may change dynamically the quality of interaction, reducing interaction costs. But it is not clear when and how these paradigms should be used in mobile computing. Obviously, CS paradigm as we mentioned in the previous chapter, is directed at traditional fixed connection computing. Mobile Agent can migrate, at times of its own choosing, from machine to machine in a heterogeneous network, and therefore have the potential to provide a convenient, efficient and robust paradigm for mobile applications, especially for disconnected operations. However, due to its autonomous operations, agents always have a wealth of features, thus becoming very heavyweight. They, hence, are usually not amenable to the resource-constrained portable devices in mobile computing. We can use the REV paradigm to cause the execution of code on a remote site to save the battery and CPU cycles of the local site. In addition, the COD paradigm enables computational components to retrieve code from other remote components. The union of these two paradigms provides a flexible way to extend dynamically their behavior and the types of interaction they support, and hence is

a good candidate to realize our mobile code toolkit. Unlike mobile agent, the unit of mobile code can be an object or a group of objects decided by the adaptive mechanism in response to current environmental changes. This fine-granularity mobility is beneficial to the resource-constrained portable devices involved in mobile computing, and considered in our toolkit design also.

2.3.4 Java Language: A Case Study

Mobility has a strong impact on programming language features such as its data size and type system, the scope of an identifier, the name resolution as well as its linking mechanism [11]. In this subsection, we analyze Java [50] from these perspectives to demonstrate that Java has triggered most of the attention and meet the expectations for code mobility.

In many other languages, such as C, C++. The data type formats and sizes may depend on the language implementation for some specific computer architecture to obtain optimal performance. Java data type formats and sizes are clearly specified. For instance, the Java compiler translates Java source programs into an intermediate, platform independent language, called Java Byte Code, which is a common format to the Java virtual machine on all architectures. This feature is important to achieve mobility.

The language type system is generally categorized into typeless system and typeful system. In typeless language, all data belong to an untyped universe and can be interpreted as values of different types when manipulated by different operators. This flexibility is desired by code mobility, but counterbalanced by the impossibility of protecting data against erroneous manipulations. The typeful system has an extreme case that is the strong type system, in which the language definition ensures that the absence

of type errors can be guaranteed for a program statically. This restriction prohibits code mobility with the reason that code can be downloaded from a remote site and linked dynamically to a running program, and type correctness of the program cannot be verified statically. Therefore, these two extreme cases always compromise with each other. Java language reflects this by weakening the type system through complementing compile-time type checking with several type checks performed at runtime. For instance, during the translation of Java source into bytecodes, static type checking is accomplished. Java Byte Code can be dynamically loaded into and executed by the Java Virtual Machine. Type checking is performed at the runtime to ensure that this dynamically loaded and linked code obeys the language type rule.

The scope of an identifier is the range of instructions over which the identifier is known. Static scoping means that the scope of a variable name is determined by the lexical structure of the program such as function scope, class scope, and file scope. However, in dynamical scoping, the scope of a name can be modified at run time by the programmer such as how function calls are nested. For example, we have a segment of Java code:

```
public class Scope
{
   static int x = 1;
   public static void print()
   {
      System.out.println("x = "+x);
   }
   public static void main(String[] args)
   {
     boolean x = true;
     print();
   }
}
```

Under static scooping the *x* written in *print* is the lexically closest declaration of *x*, which is an int. Hence the output of this segment is *x=1*. However, under dynamic scooping,

since *print* has no local declaration of *x*, *print*'s caller is examined. Since *main* calls *print*, and it has a declaration of *x* as *boolean*, that declaration is used, and output under this condition is *x* = *true*. Compared with static scoping, dynamic scoping makes type checking and variable access harder and more costly due to its dynamic nature. However, it is closely related to the virtual function concept used in C++, which implements polymorphism, an essential feature of an object-oriented programming language to support code mobility. Java adopts static scoping and realizes the polymorphism, which make it amenable to mobility.

Name resolution rules determine which computational entity is bound to each identifier in any point of a given program. It is critical in mobile code language. During the execution of a mobile code application, the names that appear in the code may be bound to entities that may be located on remote computational environments. Name resolution can either be performed automatically by the runtime support or hooks in the language runtime support can be provided to allow programmers to define their own name resolution rules. Java does not offer particular features to perform automatic name resolution for remote resources. In Java, name resolution for a local resource is performed statically and remote resources have to be accessed explicitly. For example, a Java program can download and link code from the network but the name resolution of dynamically downloaded classes has to be explicitly programmed. A Java programmer must be aware of the location of the classes and write his own class loader that resolves the names of the classes to be downloaded from the network.

Dynamic linking defers the code linkage to the program till the runtime. This code may reside on the local site or somewhere on the remote side. Remote code dynamic

linking allows programmers to implement a mobile code application based on the COD paradigm. Java exploits remote code dynamic linking extensively to enable the implementation of scalable and dynamically configurable applications. The loading and linking of the different classes that compose a Java application are performed at runtime by the class loader, which is part of the Java virtual machine.

Capturing and restoring the application states is another feature, which supports code mobility. Currently, Java provides a serialization mechanism, which allows the capture and restoration of object's states, and therefore the migration of objects between machines is supported in Java.

Although Java provides the above mechanisms to support code mobility, it still has some deficiencies. Java does not provide any mechanism for capturing and restoring a thread state. The consequence is that most of the mobile code systems implemented on top of Java only provide weak mobility. However, this limitation can be overcome by extending Java virtual machine.

2.4 Thread and Process Migration

The technologies most closely resembling the work on strong mobility is the thread and process migration in Operating Systems. Process migration [48] is the transfer of a sufficient amount of the state of a process from one machine to another for the process to execute on the target machine. Unlike mobile agents, the interest in this concept is from the research on load balancing in distributed systems. Hence, process migration has a different implementation from the mobile agent although they have to show almost the same issues. In general, process migration is usually implemented at a low level, making

no assumptions about the application structure, or even about the programming language. The machine architecture and environment are assumed to be very similar, if not identical, at the two ends of the migration. Ideally, the migration policy and mechanism are only realized in the Operating System and the processes are migrated passively by the Operating System, and transparent to other reference processes as well. However, in reality, it is hard to achieve this goal due to the inherent complexity such as the process structure, process interactions. Therefore, process migration is always under some restriction. For example, to be migratable, a process should not perform I/O on non-NFS files, spawn subprocesses, utilize their pid or any other location-specific information and exploit pipes and sockets. In addition to these restrictions, compiler and the language runtime system are sometime enriched to provide support for process migration.

Moving a thread really amounts to moving a thread state. The thread state is essentially composed of a data component representing the values of local variables in the activation records on the call stack and a code dependent component consisting of a thread's executable code and pointers into this code. Form this perspective, thread migration is similar to the process migration, and the modification of the operating system and compiler can be also used to attack this problem.

Mobile agents generally adopt mobile code languages, which are usually interpretive languages. The interpreters (virtual machine) are always the computational environments to support the execution of the mobile agent program. The time and destination for migration are determined autonomously by the mobile agent, different from the migration of processes, which are scheduled by the operating system. The mobility of an agent is usually reduced to a single instruction like the Telescript [53] *go* instruction, and Obliq

[8] *hop* instruction. These instructions are realized in the corresponding interpreters to capture the current state of the agent and transmit this state to the destination machine, and the interpreters remain fixed. On the other hand, unlike process migration, the location-specific operations in mobile agent are also handled by the agent with the aid of the interpreters. Therefore, the interpreter is always called agency to support the agent mobility. From this perspective, the mobile agent is implemented at a higher level without resorting to special support from the underlying operating system. Compared to the process migration mechanisms, the agent mobility mechanism is easy to extend to the heterogeneous platforms since it adds an extra layer, i.e., the agency, above the operating systems, whereas the computational environments for the process migration are operating systems. Hence, process or thread migration is far more difficult than agent migration. For further information about progress migration, please refer to [57].

2.5 Summary

In this chapter, we surveyed the fundamental technologies to realize adaptive applications in mobile computing. An adaptive application is established on the client-proxy-server model and distributed in nature. Its computational loads can be divided into two parts prior to its execution, and reside on both mobile hosts and its proxy server. This is a static manner for mobile applications to adapt to its slow CPU speed. CORBA, RMI and DCE are good examples to realize this approach. However, they all have limitations with regards to deployment on the resource-constrained portable devices. On the other hand, the mobile environment is a dynamic environment in terms of battery power, wireless bandwidth and other factors. Although the static adaptation can deal with the factors like

slow CPU speed of mobile devices, it can not handle the dynamic changes of battery power and wireless bandwidth. Mobile code is a promising technique to implement a dynamic adaptive application by shipping code during the execution of the program. We outlined this technique in this chapter, including its execution model, mobility mechanism, and the computing paradigm. Based on the characteristics of these paradigms, we prefer the combination of REV and COD as our paradigm to design our mobile code toolkit. A mobile code language has many distinct features compared to the traditional procedure languages. These features, as we summarized in this chapter, are centered on the code mobility. Java is a good example language to implement a mobile code system. We analyzed this language and pointed out its advantage and deficiency to support code mobility. We also choose it as our primary developing language due to its popularity and portability. Finally, we gave a brief introduction of thread and process migration, and then compared them to mobile agents. Due to the different implementation layer, thread and process migration has a different mobility model from the mobile code system, and hence it is not a mainstream in the research on mobile code.

Our mobile code toolkit is targeted toward resource-constrained portable devices, and the union of REV and COD is our paradigm to realize adaptive applications. We use Java to implement this toolkit. The function and architecture of this toolkit are partially borrowed from DOMT, a mobile code toolkit for Handheld PC with WinCE, which will be discussed in the next chapter.

Chapter 3

DOMT Overview

A mobile application is a composition of objects encapsulating functionality, consisting of two disjoint sets of objects, one residing at the portable device, and the other residing at the proxy server. These sets should change dynamically for performance improvement according to the mobile environment. DOMT is a dynamic object migration toolkit developed at Carleton University whose goal is to realize this functionality by extending Java Virtual Machine with a middleware that facilitates the mobility of objects between portable hosts and proxy servers in a dynamic manner, transparent to the application developers and users. On the other hand, to enhance the flexibility, DOMT also provides a set of mobile-aware application programming interfaces, which allow a mobile-aware application to ship its objects explicitly. Hence, DOMT offers applications a dynamic distributed object system based on a client/server architecture. Clients are DOMT applications that typically run on mobile hosts, but could run on stationary hosts as well. Servers typically run on stationary hosts and hold the long-term state of the system for making dynamic object migration decisions. DOMT consists of a class library linked into all applications and runtime objects on client and server machines. DOMT applications actively cooperate with the runtime system to import objects into the local machine or export objects to the remote machine, and invoke well-defined methods on those objects

regardless of their locations. The key task of the programmer in building an application with DOMT is to define *dynamic movable objects* (DMO), the runtime system can divide implicitly these objects into portions that run on the client and portions that run on the server. The two parts communicate by means of remote method invocation (RMI).

This chapter briefly describes DOMT from two aspects based on its fundamental support for dynamic object migration. First, I describe the structure of the toolkit, and then discuss the functionality of the toolkit in managing object migration and remote method invocation. For further insight into the DOMT Toolkit, see [37].

3.1 DOMT Architecture

DOMT consists of a class library linked into all applications and runtime modules for client and server machines. Currently, it can run on desktop computers with Windows NT and Handheld PCs with Windows CE. DOMT adopts a symmetric architecture between the portable side and proxy side, running the same program on both sides. Its runtime module is structured as three protocol layers and consists of four components. At the top is the application, linked with the DOMT library. Below the application is the Proxy Layer and encompassing the code storage. Finally, the object server is the Reference Layer for the system sitting between the Proxy Layer and Transport Layer. Their relationships are depicted in Figure 3.1.

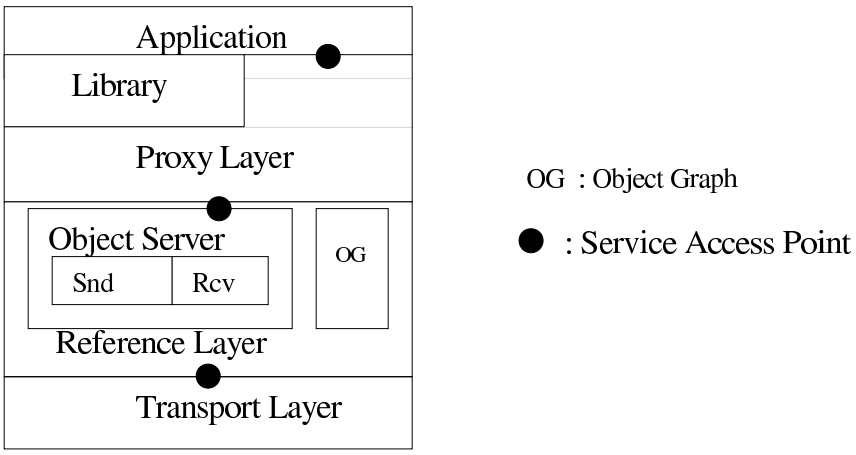


Figure 3.1 DOMT architecture

The function of each component is discussed below.

3.1.1 Library

Each DOMT application is linked with the *DOMT library*. This library defines the DOMT application programming interface and manages communication between the client application and the DOMT runtime system. DOMT applications are typically structured in an event driven manner to allow the DOMT library to handle messages from the DOMT runtime system as they arrive. The typical reaction is to ship objects between client and server. However, this structure is not strictly necessary.

3.1.2 Proxy Layer

The Proxy Layer consists of proxy objects for some specific application and code storage. The function of this layer is to provide applications with transparent access to objects locally or remotely.

- Proxy Object: Each dynamically movable object of an application needs to be associated with a proxy object that has the same interface. Other objects will not reference application objects directly, but they reference them through their proxies.

Hence, the proxy object acts as a mediator between the caller and the real object realizing transparent remote or local method invocation. In Figure 3.2, Object Px is the proxy object of Object x. The real Object B accesses real Object A through its Proxy Object P_A. This will facilitate moving objects without worrying about changing reference of other objects to it.

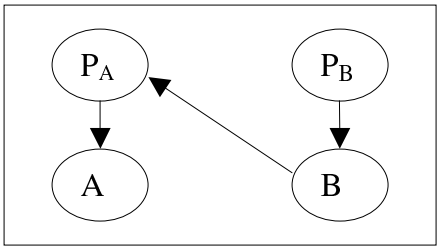
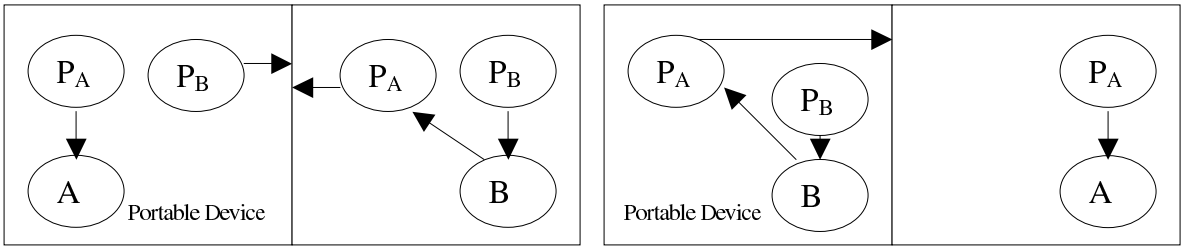


Figure 3.2 Proxy Objects with their associated Objects

There are two scenarios regarding object migration, calling object migration and called object migration. In Figure 3.2 Object B is the calling object and Object A is the called object. Figure 3.3(a) demonstrates moving the calling object, Object B from the portable device to the proxy server. Moving a calling object does not require moving its relevant called objects. At the proxy server, however, the proxies of the relevant called objects



(a) calling object migration (b) called object migration

Figure 3.3 DOMT Object Migration

must be created to forward the calls from the calling object to the corresponding real called objects residing at the portable device. In our scenario, moving Object B will not require moving Object A. At the proxy server, only a proxy of Object A, P_A must be

created to forward the calls to Object A at the portable device. A new proxy of B will be instantiated at the proxy server to allow local objects there to reference Object B. The other scenario, moving a called object has a slight difference from moving a calling object in that it is moved to the remote side without creating any other proxy objects to forward remote method invocations. In Figure 3.3(b), migrating A does not require changing the reference to A and B since B references only the proxy of A. Any calls from B to A will be forwarded remotely through the proxy to the portable device. The proxy of A at the proxy server again allows other objects to reference A without affecting the flexibility of moving A again to the portable device. These descriptions reveal that the cost of moving a calling object is higher than moving a called object since calling objects have high couplings with their relevant objects.

Services in Proxy Layer are available at its SAP (Service Access Point) called *invokeMethod* which is defined in some class to receive the method invocation parameters of some real object such as method name and its corresponding arguments. Proxy Layer is responsible for sending this data to the remote site by encapsulating it into a unit datagram and forwarding this datagram to the underlying Reference Layer through its SAP. On the other hand, *invokeMethod* can also be used to receive the message sent from its underlying layer.

- Code Storage: The code storage contains the validated classes files (bytecode) at the portable device. It is managed by DOMT runtime system. At the request of the proxy device, the code will be transferred to the proxy. On the other hand, imported class files from the remote server also reside in the code storage. The code storage is the static part of DOMT system and can survive between application sessions.

3.1.3 Reference Layer

The semantics of object migration and remote method invocation are realized in the Reference Layer. The Reference Layer combines JVMs on both the proxy server and the portable device as one virtual machine from the application point of view. The components to accomplish this function are Object References and Profiling and Object Server.

- Object References and Profiling (Object Graph): This component contains the representation of the application's objects along with profiling information about these objects. This information will be sent to the proxy server to be analyzed and the proxy server will decide which objects must be shipped to its side according to the execution environment. In DOMT, this information is collected a priori and provided to the toolkit.
- Object Server: Object Server is the main core of the toolkit. Its functionality can be divided into two parts. One is the *sender* which can be called by its upper Proxy Layer and receives the upper layer requests. When the *sender* receives a request, it will forward it to its underlying Transport Layer by accessing its SAP. The *sender* acts as the SAP for Reference Layer. However, it does not receive any message from the Transport Layer. The message from the Transport Layer is received by the *receiver* in the Reference Layer. Hence, the *receiver* interacts with the SAP in the underlying Transport Layer directly. It runs a thread that listens continuously to all the commands from a remote object server. Commands can be related to moving objects or related to the remote invocation of a method on a remote object.

3.1.4 Transport Layer

The Transport Layer is at the bottom of DOMT architecture. Its function includes marshaling and unmarshing method's parameters and simulating wireless links in terms of low bandwidth. In DOMT, marshaling and unmarshaling are realized by the serialization mechanism provided by JVM. Simulating wireless link, in DOMT, is to chunk the data streams being sent through the Transport Layer into pre-determined sizes based on empirical tests. DOMT introduces a controllable amount of delay between data chunks, which allows us to control the throughput dynamically at run time for testing purposes. The following equation is used in DOMT to determine the simulated throughput. Changing both the delay and the chunk size changes the throughput.

Throughput(Kb/s)= (10(ms)x838(Kb/s)*Chunksize(bytes))/(Delay(ms)*1024(bytes))

The SAP in this layer is typically the *ObjectInputStream* and *ObjectOutputStream* provided by JVM.

This three-layer architecture is realized on both the mobile client and its proxy server. Hence one instance of the toolkit executes on both sides. We refer to this architecture as symmetric architecture.

3.2 RMI Protocol

When the toolkit is initialized at the portable device and the proxy server, a socket connection is established in the Transport Layer 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 de-serialized 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.

When a method is invoked on a proxy object, the proxy will know whether the associated object is local or remote. Hence, it decides Local Method Invocation (LMI) or Remote Method Invocation (RMI). In DOMT, every proxy object and its associated object have a unique entrance point. If a method invocation occurs through this proxy, its entrance point will receive the corresponding method name and its parameters and check whether it is LMI or RMI. If the associated object is local, this entrance point will communicate with its counterpart in the local associated object and forward the call to the associated object through its entrance point. The entrance point of the real object will

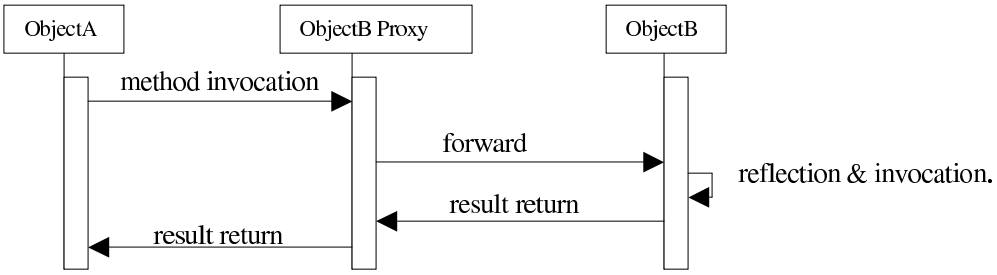


Figure 3.4 DOMT LMI Operation

invoke the corresponding method based on the reflection mechanism. Figure 3.4 shows the LMI scenario. In this scenario, Object A invokes a method in Object B. Object A and Object B coreside at the same physical site.

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 specific 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 corresponding proxy object. If the proxy object is found, then the server will ask the proxy to invoke the 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 a waiting state until the exceptions or results are returned.

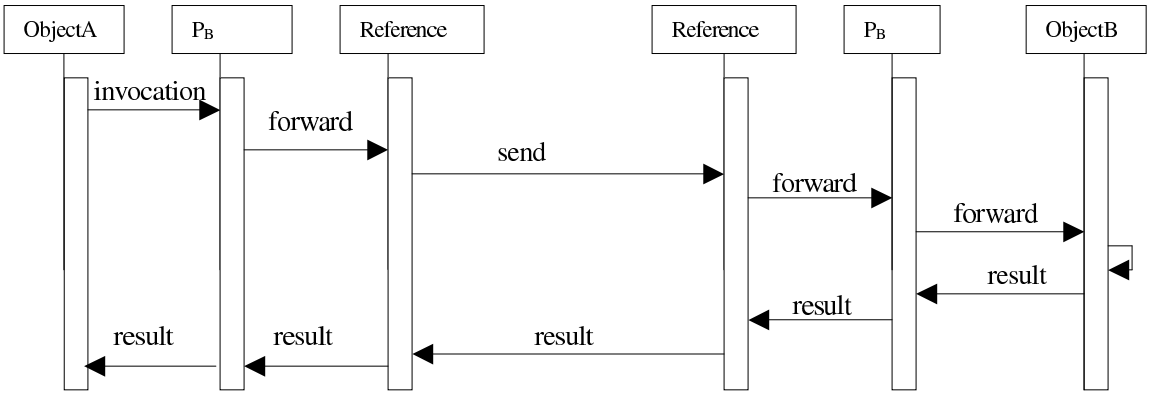


Figure 3.5 DOMT RMI Operation

Figure 3.5 shows the RMI scenario. In this scenario, Object A invokes a method in Object B. Object A and Object B reside at different physical sites.

3.3 Distributed Garbage Collection

Determining when objects are no longer in use is a problem in distributed computing toolkits. In a naive approach, Java's garbage collector can reclaim objects, however, consider Figure 3.2, where object A and a proxy to A exist on the proxy server. Since no other references to these objects exist locally, the local garbage collector will reclaim these objects, even though object B on the portable device still has a remote reference to object A. To deal with this, every object created in DOMT is assigned a local and remote reference counter. DOMT creates references to each local object and manages them via these reference counters to avoid the premature finalization of objects. These counters are updated whenever a proxy object is referenced locally or remotely. This requires sending messages between the portable device and the proxy server to keep the object reference counters updated. If the associated object of this proxy is local, then the 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 garbage collects the object if there are no further references locally or remotely to the associated object.

3.4 Discussion

DOMT is an adaptation approach for portable devices to reduce power consumption and to improve application performance, in which a part of an application will be encapsulated in a mobile component and potentially shipped for execution to proxy servers, according to the portable device and fixed host's available resource and wireless

network state. The platform for which DOMT is designed is Handheld PC with WinCE as its operating system. It adopts Java to hide the irregular architectures of underlying system and hardware to improve its portability. With this toolkit one can combine JVMs on both the proxy server and the portable device into one virtual machine from the application perspective. Experimental results show that it is possible to simultaneously improve application performance and reduce power consumption by migrating some CPU bound objects to the proxy server in the case of a slow portable device and sufficient wireless bandwidth.

Although DOMT is a promising approach to support adaptive application on portable devices, there are still some problems to be addressed, especially when considering its use on resource-constrained portable devices like PalmPilot, which provide a more limited JVM than Handheld PCs. We classify the problems into two categories: one is those which are inherent to DOMT itself, while the other are those related to its portability to the resource-constrained portable devices. One of the problems in the first category is the automation of the dynamic application partitioning process. DOMT results indicate which parameters are important, based on a predefined object graph, which is obtained by tracing a few executions with JProbe [56] or other profilers. It lacks a monitor, however, to detect the changes in the live application environment as well as the applications themselves, to obtain these parameters on the fly for constructing an object graph with minimal overhead and making partitioning decisions to achieve high performance. These changes may result from user movements (moving into an area of substantially different network connectivity), the actions of other users (as more users start offloading code to a shared proxy server, the relative CPU difference will change),

the link state change, such as the forthcoming disconnection of wireless links which limit the capacity of the link from one time to another, or other factors. Intuitively, one would expect the runtime system to rebalance the application accordingly. However, shipping objects at runtime is not cheap. Therefore, we need to explore how to balance the resulting overhead with the anticipated performance gains and power reduction, in particular in execution environments that change rapidly. One possibility is to allow the mobile user to select preferences that prioritize the movable objects.

PalmPilot as we described, is a new resource-constrained portable device, which is anticipated to dominate the palm-size segment of the PDA market. Although DOMT simplifies the design of code mobility mechanism to fit PDAs compared to other systems, it does not execute on the PalmPilot. Problems in extending DOMT to PalmPilot stem from two aspects. First, the symmetric architecture of DOMT is not appropriate for PalmPilot since the design assumes that the mobile client and its proxy server have almost equivalent computational power and memory storage. Second, DOMT is based upon the powerful JVM, whereas PalmPilot is only equipped with KVM, a compact, portable Java Virtual Machine intended for small, resource-constrained devices. The fundamental mechanisms of object serialization and class reflection are not provided in this scaled-down version of a virtual machine.

All in all, these two categories of problems motivated us to redesign a new version of a mobile code toolkit for PalmPilot as well as WinCE based on the framework of DOMT. The new toolkit will be KVM-based, adding some new modules to deal with the problems in the first category, redesigning the architecture and providing the necessary fundamental support such as object serialization and class reflection to attack the

problems in the second category. The new toolkit is called KMOT, short for KVM-based Mobile Object Toolkit.

Chapter 4

Design and Implementation of KMOT

4.1 Introduction

KMOT is a PalmPilot oriented mobile code toolkit, which provides palm application developers with a set of tools to isolate applications from the limitations of mobile environments. KMOT provides the same functions as DOMT, such as object migration and remote method invocation to support adaptive application, and realizes a uniform distributed object system based on a client/server architecture. Since our research on KMOT borrows from early work on DOMT, features of DMOT are also mentioned from time to time to illustrate the rationale behind our design.

4.2 Fundamental Support

In order to realize code mobility in KMOT, some supporting mechanisms must be provided. In this subsection, we will discuss the implementation of object serialization, distributed object graph and the construction of proxy objects.

4.2.1 Object Serialization

Object serialization is the act of writing a data object in a serial, byte-at-a-time manner to files or communication channels [12], and thus is fundamental to object migration

between a mobile host and its proxy server. However, object serialization always incurs large runtime overhead and degrades the overall performance of RMI (Remote Method Invocation). Many publications [29, 40] addressed this problem with or without native code. For KVM, due to the lack of JNI (Java Native Interface) support, we prefer methods without any native code and design a separate module for this purpose. Our approach is to adopt the externalization protocol described in [40], asking the movable object to serialize itself. In order to achieve this successfully, a serialization protocol must be set up. This includes three steps:

(1) Defining the serializable interface which contains only two methods for writing objects and reading objects.

(2) Serializing the system core and application classes. Application classes can be serialized by implementing the serializable interface directly. System classes that come with the KVM are integrated in the executable, and can not implement this serializable interface directly. Hence they can be serialized by wrapping them in a thin class designed to provide the methods of the serializable interface. On the other hand, each application and thin wrapper class should contain a default constructor. This default constructor will be used to construct a movable object in a remote site.

(3) Defining our own ObjectOutputstream and ObjectInputStream classes. Since we have no reflection mechanism, these two classes can not detect the class structure of the object to be moved on the fly, they will require the serializable object to take part in this process.

In order to maintain the transparency to the application, we define various conversion functions to use in object migration. The consistency of the Java type system is ensured

by defining a new root object with serializable interface. It must be inherited by any movable object. The following example demonstrates our serialization protocol:

Serializable Interface :

```
public interface K_Serializable {
    public void writeObject(ObjectOutputStream out);
    public void readObject(ObjectInputStream in);
 }
```

Root Object Class:

```
public class K_Object extends Object implements K_Serializable  {
  public void writeObject(ObjectOutputStream out);
  public void readObject(ObjectInputStream in);
}
```

Application Class:

```
public class App extends K_Object implements K_Serializable {
  int i;
  Str s
  public App() {}
  public App(int i) {
      this.i = i;   s = new Str("abc");
  }
  public void writeObject(K_ObjectOutputStream out) {
       out.writeInt(i);
       out.writeObject(s);
  }
  public void readObject(K_ObjectInputStream in) {
       i = in.readInt();
       s= (Str)in.readObject()
  }
}
```

System Class:

```
public class Str extends K_Object implements K_Serializable {
  String str;
  public Str() { str = null}
  public Str(String s) {
    str = s;
  }
  public void writeObject(K_ObjectOutputStream out)
       out.writeUTF(str);
  }
  public void readObject(K_ObjectInputStream in) {
       str = in.readUTF();
  }
}
```

In this example, we give the definition of the serializable interface, the root object class, a system class which is a class wrapping a *String* object as well as an application class. We try to serialize an App object to the remote site. *K_ObjectOutputStream* and *K_ObjectInputStream* classes define the serialization and deserialization process in our toolkit. When an App object *anApp (new App(10))* is serialized, a method *writeObject(anApp)* defined in *K_ObjectOutputStream* is invoked, it writes the class name of the serialized object followed by the fields we chose to write, in the order we chose to write them. Based on the class definition, the primitive type can be written into the stream directly, whereas the object type is written into the stream recursively.

App	1	Str	"abc"	

Figure 4.1 KMOT Stream Format

Figure 4.1 illustrates the stream format. When the receiver deserializes the stream, it first reads the class name, *App* and creates an object based on this name. This is the function of the default constructor. The subsequent fields are read in successively. If the field is object type, the read procedure is recursively, corresponding to its writing counterpart.

In KMOT, object migration is done in what we call a *briefcase*, an instance of a class containing a folder and a label. The folder contents can be arbitrary data or code whose type is indicated by the label. This simple method not only allows us to serialize everything but also provides us with the flexibility to compensate some deficiency in DOMT, which will be discussed next.

4.2.2 Distributed Object Graph

The *object graph* is a run-time structure of an object-oriented program, its nodes represent the objects and edges indicate their reference relationships. In DOMT, the object graph is constructed offline and provided to the toolkit. When the object graph is partitioned for migration, some objects will be shipped to the proxy side. Hence, a centralized object graph residing on a mobile host becomes a decentralized one residing on both the mobile host and its proxy server, which is called a *distributed object graph*. The program execution on this distributed object graph should be consistent with its execution on the centralized object graph and improve its performance, compared to the centralized counterpart. Unfortunately, DOMT will result in more remote method invocations when it moves two related partitions successively and one references the other, since the underlying serialization process in JVM does not remember their relationships as soon as the first one is restored in the remote side. The following is an example to demonstrate this process.

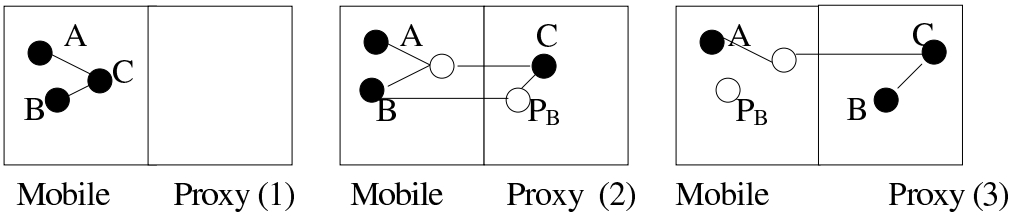


Figure 4.2: Objects Migration : ● real object ○ proxy object

In Figure 4.2, the object graph of an application consists of three objects A, B and C. Their relationships are indicated in Figure 4.2 (1). A invokes a method in C, denoted by C.f( ), while inside C.f( ), some statement invokes a method in B, denoted by B.g ( ). When C moves to the Proxy, it will bring B's proxy object with it to the Proxy since it is a calling object. When A invokes C.f( ), this invocation will be forwarded to C through its proxy object on the Mobile. Based on the same principle, C invokes B.g( ) through B's proxy object in Proxy. When B.g( ) is finished, the result will be returned following the same path but in the reverse direction (Figure 4.2(2)). At this point, everything is fine. Moving an object in DOMT resorts to *writeObject* defined in the *ObjectOutputStream* class in JDK, and accordingly, receiving the movable object in the remote side resorts to *readObject* defined in the *ObjectInputStream* class. We define the time during these pair of invocation as a *migration session.* In another scenario, we move C and B during the same migration session, and the distributed object graph is shown in Figure 4.2(3), A invokes C.f( ) through its proxy object in Mobile, while it invokes B.g( ) locally. In the same migration sessions, the relationships between the relevant movable objects are kept in the Java serialization process. The problem occurs when C and B move to the Proxy in separate migration session. The serialization process in JVM does not remember the

relationship between C and B. Therefore, when B moves to the Proxy after C has been moved in a prior session, its relation with C will be lost (Figure 4.3(1))

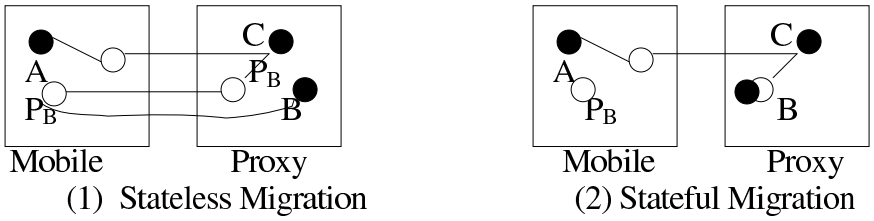
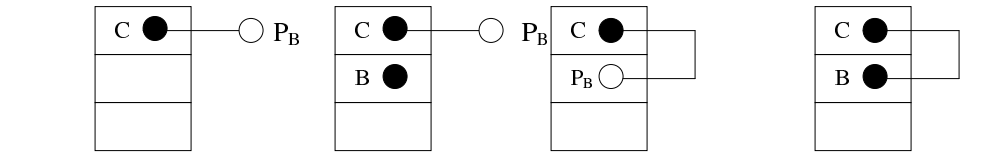


Figure 4.3 Migration Categories

If C wants to reference B at the Proxy, it has to invoke B's proxy object at the Proxy instead of B directly. B's proxy object at Proxy will forward this reference back to the Mobile to invoke B there, but unfortunately, B is on the Proxy, so B's proxy object at Mobile will forward back this request to the Proxy again, B is referenced and the result will follow the same route to return to C. We call this kind of migration *stateless migration*. In stateless migration, the relationships between the relevant objects are not kept in the serialization process but can be recovered through their proxy objects. However this recovery is extremely inefficient. For example, in our previous scenario, when C invokes B.g( ), it needs four additional remote method invocations compared to moving them in the same migration session. This performance degradation resulted from stateless migration is called *performance anomaly*. Such a performance anomaly does not suggest moving relevant partitions successively. Intuitively, relevant partitions should co-reside on the same site. DOMT attempts to address this problem by registering proxy objects in some object cache when they move to the local site from a remote site. The object cache provides stable storage for local copies of proxy objects. These proxy objects will be retrieved when their associated real objects arrive. However, this approach does not solve the problem completely. For example, when C moves



(1) DOMT: C arrives (2) DOMT: B arrives (3) KMOT: C arrives (4) KMOT: B arrives

Figure 4.4 Snapshots for Object Cache

to Proxy, the snapshot of its object cache is shown in Figure 4.4 (1). Suppose in the next migration session, B arrives, DOMT will search its proxy object in the object cache, and can not find it. Then B will register in the object cache again (Figure 4.4(2)). This solution fails because DOMT does not process the referenced proxy like P_B in our example. Although this problem can be solved in DOMT by reflecting C to obtain its referenced proxy object, it is infeasible for KMOT to do so since we have no reflection mechanism.

In design of KMOT, what we want is a kind of *stateful migration* which can keep these relationships in the serialization process even when the object migration happens in different migration session (Figure 4.3 (2)). In our scenario, when C moves to the Proxy, it, together with its associate proxy object P_B, will register in the object cache (Figure 4.4 (3)), as soon as B arrives, it can easily find its proxy object there. We can realize this mechanism by properly design the serialization protocol and proxy objects.

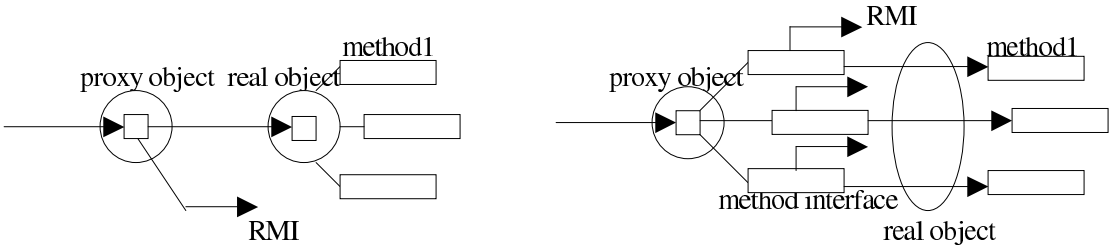
In order to recover the relationship between movable objects in different migration session, we will follow two steps. First, we assign a unique number to each real object as its ID and register every proxy object as well as its associated real object in the local object cache when they are created. Due to the recursive creation, the proxy objects referenced by the associated object are created and registered in the object cache too. Finally, the partial object graph is set up in this object cache. This object graph is useful

when some objects that once moved to the remote side move back. Second, when an object moves to the remote side, we introduce some extra logic into the serialization process to overcome the constraint of object serialization in DOMT, and hence to avoid the performance anomaly. Since the proxy object and its associated real object have different semantics, we distinguish them when we serialize them to the remote site. In the remote site, the serialized object will be deserialized and its type is checked to see whether it is a proxy object or a real object. If it is a proxy object, the deserialization process will search the object cache by using its associated object's ID to see if this proxy object has been registered. If the proxy object is not registered, the proxy object will be constructed, its fields are read in from the input stream, and then it is registered in the object cache. Otherwise, it will be retrieved from the object cache, and the following real object can find its correspondent proxy object, and the distributed object graph is constructed. Hence, the relationships between the calling object and called object are recovered in separate migration sessions. In our case, this is shown in Figure 4.4 (4). If it is a real object, the standard procedure is followed.

The distributed Object Graph is a very important concept in the design of KMOT. It allows us to realize the stateful migration which is fundamental to establish dynamic object migration. Another benefit from realizing stateful migration is the construction of a partial object graph in the object cache. The object graph can provide us with partitioning information to make scheduling strategies for shipping an object from the portable devices to their proxy, or vice visa. We will discuss this problem in designing the Monitor for KMOT.

4.2.3 Proxy Objects and Class Reflection

The semantics of proxy objects in KMOT is the same as those in DOMT, acting as a mediator between the calling object and the called object to handle object mobility and method invocations. DOMT relies on class reflection provided in JVM to realize the remote method invocation. Class reflection supports introspection about the classes and objects in the current JVM. Due to the limitations of KVM, this mechanism is not provided. Hence, we have to rethink the design of proxy objects. As we described in Chapter 3, when a method invocation occurs in a proxy object (Figure 4.5 (1)), its entry point method will be invoked to check whether this invocation is RMI or LMI. If it is RMI, the entry point method will forward this invocation to the remote site directly. Otherwise, it will forward it to its real object. The real object entry point method receives this invocation and reflects the corresponding method to be invoked. The advantage of



(1) DOMT Method Invocation (2) KMOT: Method Invocation

Figure 4.5 Method Invocation

this solution is that the proxy object does not need to know the exact information of its associated object, even information about the associated object's class. Hence, the proxy object constructed in DOMT is very general, independent of any specific object class. All work is done in its associated object. However, in KMOT we can not use this approach. Actually, the generality of the proxy object in DOMT is not necessary. A proxy object can aim at a specific class, which means that any number of instances of this class can

share the same proxy object. Therefore, this kind of proxy object can contain the class information of its associated object. This idea is very useful to compensate for the deficiency of KVM. If we reflect a class file in advance and put the reflection information into a proxy object, which is created later, then the lack of reflection mechanism can be overcome. As soon as the proxy object receives some parameters such as method name and its arguments, it will know immediately which segment of code should be executed based on this information, since every method interface is hard-coded in the proxy object. Here, method interface means a method in the proxy object, which has the same signature but with different content from the corresponding method in the real object. The semantics of the content in the method interface are to invoke the corresponding method in the real object or to forward the invocation to a remote site. This procedure is described in Figure 4.5 (2).

Based on the principle described, we simulate class reflection in KMOT by defining an abstract *ProxyObject* class inherited by every proxy object. This abstract class contains an abstract method, which provides a unified interface for various remote method invocations, whereas the concrete implementation resides in the proxy objects. Since the proxy object knows the structure of its associated object, this implementation can handle all method invocations on this object. Following is the Java-like pseudo-code in our simulation (please refer to the previous example in page 50):

- **Abstract ProxyObject Class**

```
public abstract class ProxyObject extends Obj implements Serializable
{
   ……
   public abstract Obj exec(boolean returnFlag,  // true, if the method has a return value
                            String methodName,  // indicates the method name to be
                                                //invoked
                            Obj parameters,        //  the parameters of this method
                            boolean parametersPrimitiveFlags  // the parameter's type,
```

```
                                                        //primitive or class
                        } throws Throwable
    ……
  }
```

- **An Proxy Class of App**

```
public class App extends ProxyObject implements Serializable
{
    public String method1(…) {…}
    ……

    public Obj exec(boolean returnFlag,
                    String methodName,
                    Obj parameters,
                    Boolean parametersPrimitiveFlags
                    ) throws Throwable
     {
          if(methodName.equals("method1"))
          {
               String result = method1(parmeters);
               return new Str(result);
           }
          …
      }
  }
```

Figure 4.5(2) depicts the remote method invocation in KMOT without class reflection mechanism. For example when a remote method invocation occurs to *anApp.method1(…),* KMOT analyzes this request to obtain the information about *returnFlag, methodName, parameters* and *parametersPrimitiveFlags,* and packs it, together with the called object name *anApp* to send to the remote side, where this package is resolved, and the unified interface is called. The segment code like

```
ProxyObject obj = (ProxyObject)podb.getObject(anApp); //podb is the object cache.
ResultObj=obj.exec(returnFlag,methodName,parameters, parametersPrimitiveFlags);
```

AnApp's proxy object is retrieved from the object cache *podb*. In DOMT, the subsequent procedure will need a reflection mechanism, whereas in KMOT, it is replaced by the abstract method invocation *obj.exec(…).* The disadvantage of this approach is that extra logic has to be introduced to the proxy object. However, it avoids the expensive class reflection. We will evaluate these performance differences in Chapter 5.

In DOMT, the proxy pattern can be seen as a remote proxy that provides a local representative for a movable object in a different address space. When a movable object needs to be created locally, its proxy object is created first, followed by itself. We define this kind of binding between a proxy object and its associated real object as *early binding*. However, movable objects are always expensive with respect to CPU or memory. Hence, this creation is generally expensive as well. Thus, we should avoid early binding to create all "expensive" objects at once when the program is initialized. The proxy creates its associated object only when it is referenced. This kind of binding is defined as *late binding*. Late binding is adopted in KMOT. In KMOT, the real object is created on demand only when its method is invoked. Therefore, the proxy pattern in KMOT can be seen as a composition of remote proxy and virtual proxy.

The construction of proxy objects can be online or offline. DOMT programmers generate the proxy files manually, and hence it is an offline method. In order to ease the burden of application developers, we realize a proxy compiler *pcomp* to generate the proxy file automatically. *Pcomp* reads in a Java class file, reflects its information, and outputs this information as well as other common information related to a proxy object to a proxy file. This proxy file is compiled by *javac* to generate the proxy class file loaded by the classloader of virtual machine, JVM or KVM. Since *pcomp* resorts to reflection mechanism, it only runs on JVM. This compiler method is also an offline method. The alternative online method can be realized in the classloader. An example is the Voyager toolkit, which will be investigated in Chapter 6.

4.3 KMOT Architecture

Our goal in designing KMOT is to have a thin, yet powerful client with functions comparable to its proxy server. Like DOMT, KMOT also contains a library with the same application programming interface and functionality. It is linked with KMOT applications to provide basic services of the toolkit. DOMT adopts a symmetric architecture with the implicit assumption that the mobile host and proxy host have approximately the same computational power. The *Object Server* is the core component in this architecture, which realizes the client and server function to accomplish the management of object migration and remote method invocation at the same site. Hence, the communication between the mobile client and the proxy server is through two symmetric connections which are set up during the toolkit initialization (Figure 4.6).

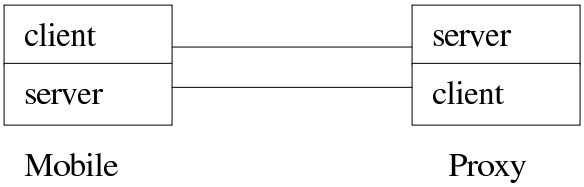


Figure 4.6 DOMT Communication Setup

Although this architecture simplifies the deployment of DOMT on various devices, it is not suitable for the PalmPilot due to the following reasons, which are also our motivation to redesign the structure of KMOT.

(1) We think that realizing a server on the PalmPilot is not advisable since the Palm does not serve the outside world constantly. Also, implementation of a server in Palm will require a lot of resources.

(2) PalmPilot can access network services via the third party software installed on its desktop machine. The advantage of this access manner is that it is not necessary for

the PalmPilot to have its own IP address, it can use the IP address of its desktop machine through PPP protocol to access the network service. However, this access manner is unidirectional, the outside world can not access the service installed on the PalmPilot via some software on its desktop machine. Therefore, unless the PalmPilot has its own IP address, and connects to the network directly, DOMT's architecture is not suitable for the PalmPilot.

(3) If the Proxy serves many mobile hosts, each mobile host should have its own client part in the Proxy. Hence, scalability of the Proxy is a problem. Actually, this problem has no impact on the PalmPilot. However, if we change or add some logic in the Proxy to manage the multiple clients, the symmetric architecture will be broken.

4.3.1 Three Layer Structure

KMOT, other than the object migration, also needs the capability to accept the requirement of remote method invocations. This requirement is a natural result of object mobility and fulfilling this requirement will induce an equivalence of functionality between the mobile client and its proxy server. Our design is based on this principle by simplifying the DOMT client/server-server/client architecture to a client-server architecture. Some symmetry in components, especially the Object Server, is broken up to reduce the toolkit footprint on mobile devices (Figure 4.7). The equivalence of functionality is addressed by *distributed recursive method* (DRM) on both sides, which will be discussed in the later subsection. Borrowed from the framework of DOMT, the

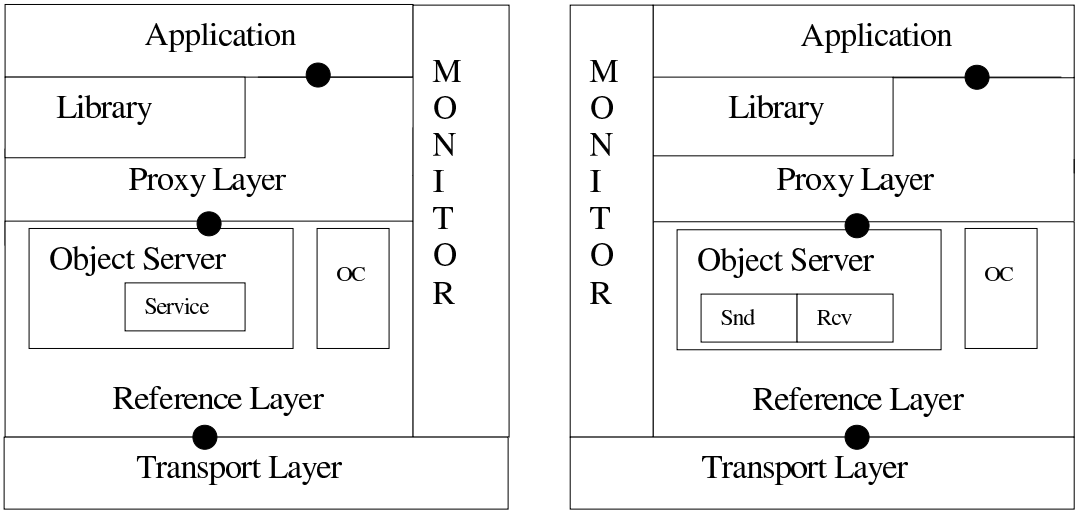


Figure 4.7 KMOT architecture OC: Object Cache

architecture of KMOT also consists of three layers: Proxy Layer, Reference Layer and Transport Layer, and adds a new Monitor component.. The semantics of these three layers is identical to those in DOMT. The Proxy Layer consists of proxy objects for some specific application and code storage. The only difference is the content in proxy objects which includes extra logic to compensate for the lack of class reflection as described above. The semantics of object migration and remote method invocation are realized in the Reference Layer as in DOMT. Unlike DOMT, the object graph is partially constructed in the object cache dynamically. The object cache provides a stable storage for locally created objects as well as imported objects from the remote site. Applications do not usually directly interact with the object cache. It can be seen as a system resource managed by KMOT itself. The contents in the object graph are shared by the Object Server, the serialization protocol in the Transport Layer and the Monitor to ensure the consistency of the semantics of the application execution. The Object Server is the main core of the Reference Layer that mediates all interactions between mobile client applications and the proxy server. Hence, it allows the Reference Layer to provide

applications with a consistent communication interface. A client implicitly uses the Object Server to import objects from a remote site, and caches them locally or exports objects to the remote site. Remote method invocations are also managed by the Object Server. In KMOT, the symmetric structure of the Object Server is broken. On the mobile client side, the "rcv", which is the server in DOMT, is discarded, and the function of "snd" is enriched by DRM (Distributed Recursive Method) to accomplish the nested method invocations. In the Transport Layer, we omit the function to simulate wireless links in terms of low bandwidth since this function can be realized by third party software. The semantics of the Transport Layer are slightly different from the counterpart in DOMT, where the Transport Layer is responsible for low-level communication without any knowledge about the data semantics. In KMOT, some data semantics are known by the Transport Layer to enable different processes, including object cache access. These processes are described in more detail in the class reflection and object serialization sections of this chapter.

4.3.2 Monitor

In order to enrich the functions of KMOT and hence enhance its flexibility, we add one more component to the toolkit: the Monitor. Its function is to monitor the changes in the mobile environment such as the bandwidth, the power status as well as the current object graph. The state of changes can be delivered as events by the Monitor to the interested application components. The Monitor is independent of the Proxy Layer and the Reference Layer. It only resorts to the Transport Layer to communicate with its counterpart on the other side. The communication between Monitors uses a different connection from the Reference Layers. Hence object migration and remote method

invocation can happen at the same time. The function or structure of the Monitors in both sides is not symmetric. The decision to migrate objects in the background is made in the proxy side since it consumes CPU cycles and other resources. The decision strategy is discussed in section 4.5. The Monitor cooperates with the Object Server through the object cache to support adaptive application.

4.4 Distributed Recursive Method

Methods in different objects can be invoked in a nested manner. Because of the object mobility, the nested method invocation may happen on the mobile host and proxy host alternatively. Here, it is worth noting that what we described includes a direct or indirect recursive invocation as a special case of the nested ones. For example: two objects A and B reside on the proxy host and the mobile host respectively (Figure 4.8). Suppose we have the following execution:

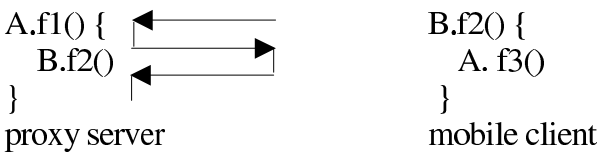


Figure 4.8: Nested Method Invocation

In object A, f1() will invoke f2() in object B where f2() in object B invokes f3() in object A. This sequence of invocations will result in heavy traffic between the mobile client and the proxy server. Unfortunately, unless all relevant objects reside on the same site, this phenomenon cannot be totally avoided. In order to relieve the burden of highly nested invocations, we adopt a single thread method called *distributed recursive method* (DRM), which is different from the methods used by DOMT that resort to the server service and

multithread function. The idea of DRM is very simple since nested method calls are inherently ordered. The only difference between DRM and other regular nested method invocations is that the nested methods are distributed on different hosts. DRM simulates the process of a function call in a single host but on two stacks, one for a mobile client thread, and the other for a proxy server thread. Referring to the example above, suppose a program P on the mobile side invokes the remote method A.f1(), it sends the request to the proxy server and waits for the result. On the proxy side, A.f1() calls B.f2() on the mobile side, and sends this request to the program P, then A.f1() waits for the result. On the mobile side, P invokes B.f2() and transfers the control to it. When B.f2() invokes A.f3(), it sends a request to A.f1() which is waiting for a result. A.f3() is invoked and obtains control. This process can continue until a result is returned from a method. The result will follow the reverse direction to the initial invocation. From this description, one can see that if a method needs to invoke a remote method, it sends a request and waits for the response. If the response is the result it needs, the method will continue or exit. If the response is another request to execute a method, it will invoke that method immediately. For the new method, it will repeat the same procedure. We abstract this mechanism into the reference layer, and with the aid of the proxy object concept, DRM is transparent to the applications.

DRM reduces the complexity of the control logic to execute the nested method distributed on different machines. Otherwise, multithreaded execution should be employed, as in DOMT, where every proxy object is implemented as a separate thread, and ordering of certain events for executing distributed nested methods between these threads is realized by synchronization primitives such as wait() and notify(). This

approach likely will be erroneous or inefficient if implemented by novices in concurrent programming. Another advantage of DRM is that it is very easy to extend DRM to any number of machines, each being part of the nested invocations. However, since DRM use a single thread for nested methods, the thread stack may overflow, especially in resource-constrained portable devices.

4.5 Scheduling Strategy

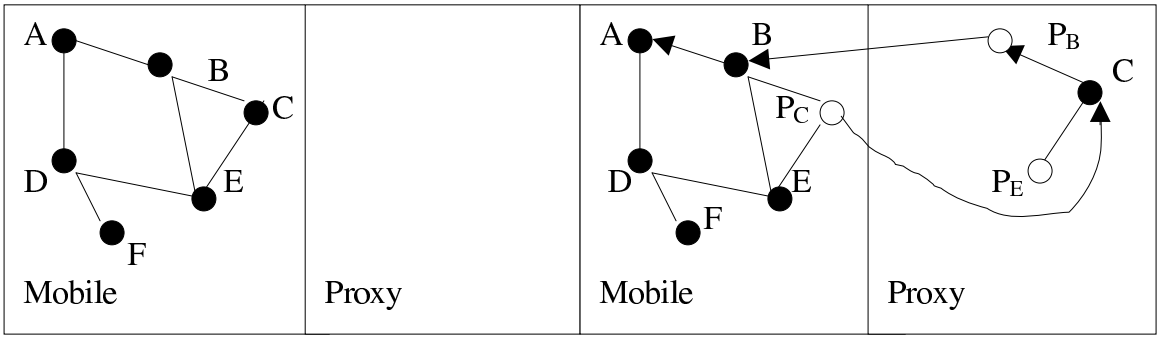
The goal of object migration is to improve the overall performance of the application program. Ideally, the entire object graph can be shipped to the remote fast proxy server. However, due to the link state, the workload on the proxy server as well as other factors, only a subset of this object graph may be shipped to the proxy server.

Partitioning an object graph for shipping can be static, partially dynamic or fully dynamic. In static partitioning, the object configuration is determined at compile time and cannot be changed. In partially dynamic partitioning, the location of an object is determined dynamically during application initialization, but cannot change during the application session. This kind of partition is sufficient for adapting to some static device configuration such as terminal types. The most general concept is fully dynamic partitioning, which allows objects to be moved at any time during the application session. This is useful when bandwidth or other dynamic resource such as memory changes radically, and the change is expected to last for some time. We adopt the fully dynamic partitioning strategy and realize a simple algorithm to schedule objects for mobility in the monitor on the proxy side. The objects scheduled to the remote site make up a partition of the object graph.

A good scheduling algorithm needs to consider many factors, which involve the mobile environment parameters and application profiles. The mobile environment parameters include the available Quality-of-Service (QoS) along the communication paths, workload of the proxy host and power level in the mobile host. The application profile may list the objects in the application, and a set of possible configurations for them. Algorithm designers can choose and decide the information in the configurations that will highly affect algorithm performance. In KMOT, we only consider the application profile and a simple configuration. The application profile is an object graph, which only includes the movable objects. The object graph is stored in the object cache. The configuration information for each object is its local and remote reference counts, coded in its proxy object, which indicate how many times this object is referenced locally or remotely. In this sense, these concepts are different from those in distributed garbage collection we discussed in the Chapter 3. Initially, the object graph is constructed on the mobile site, with the object migration, this central object graph becomes a distributed object graph residing on both the mobile and the proxy sides. Stateful migration ensures that this process is feasible.

When a calling object is moved to the proxy server, some proxy objects referenced by this object are also shipped. They are all stored in the object cache as a partial object graph, which is the basis for our scheduling strategy. Hence, when a real object is moved, its proxy object as well as its referenced proxy objects will exist at both sites. When some proxy object is referenced for a remote method invocation at their original site, its local reference count will increase by one and send a request to its remote counterpart; otherwise the local count is kept intact. As soon as the remote proxy object receives this

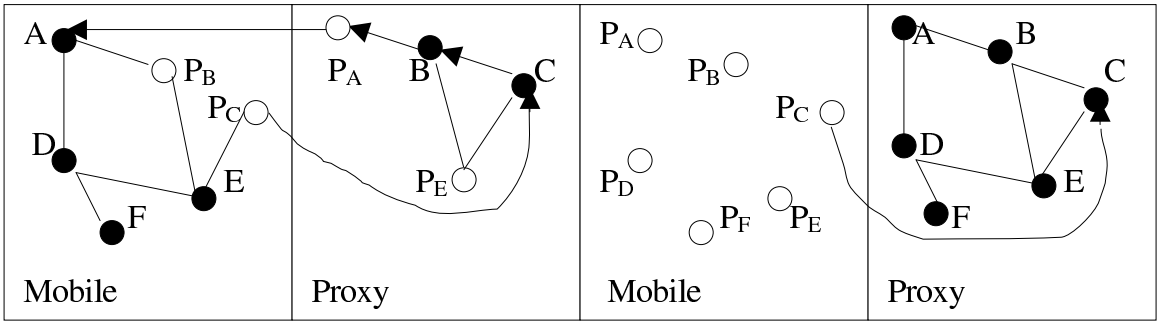
request, its remote reference count will increase by one. In our design, the monitor at the proxy site traverses the object cache periodically to check if there is any proxy object with a remote associated object and a local reference count exceeding a threshold value. If there exist several ones, we assume that they will be referenced locally in the near future. The monitor will select one of them randomly and send a request to the mobile client to move this object to its local site. When the mobile client receives this request, it will retrieve the object from its local object cache, and send it to the proxy server. When the proxy server receives this object, it will reset this proxy object's local and remote count to zero. Based on the same principle, when this object is shipped to the proxy server, some proxy objects referenced by this object are also moved, and we can repeat the procedure. On the other hand, the monitor at the proxy site also checks the proxy object's remote count, which indicates the number of remote references of this object, and schedules this object to the mobile site if its remote reference count exceeds some threshold value. If there exist several ones, the monitor will select one of them randomly to move to the mobile site with the assumption that it will be referenced remotely by the mobile client in the near future. We call this strategy *Random Greedy Strategy* (RGS) since only a random neighbor of an object is selected every time. Before giving the formal description of the algorithm, we first give an example to demonstrate it. For simplicity, we just consider the situation of the proxy object's local count in the proxy server. For the proxy object's remote count, the principle is almost the same.



(1) RGS initial state (2) RGS Object A moves to Proxy

Figure 4.9 RGS Scenario 1

Figure 4.9 (1) shows the initial state, the object graph contains five objects created locally on the mobile side. In Figure 4.9 (2), C, which is a calling object, referencing B and E, is moved to the Proxy. We then reference C in the Proxy side through its proxy object P_C in the Mobile side, C references B in the Mobile side through its proxy object P_B in the Proxy side. B may reference A or E in the Mobile side and the result is returned following the same path but in the reverse direction. After this invocation, from the point of view of Proxy, the remote reference count of P_C (C and P_C are bound together) and the local reference count of P_B should be increased by one. Suppose the local reference count of P_B exceeds the threshold value and this information is detected by the monitor in the



(1) RGS B moves to Proxy (2) RGS Entire Object Graph moves to Proxy

Figure 4.10 RGS Scenario 2

Proxy, then the monitor tries to move B to its local side (see Figure 4.10 (1)). Figure 4.10 (2) shows when the entire object graph is moved to the Proxy side. Naturally, every object can move around between the mobile side and its proxy side based on our RGS algorithm.

Now we give the formal description of Random Greedy Algorithm. The algorithm is distributed in nature. The proxy object and monitors residing on both the mobile client and the proxy server are involved in the algorithm. However, their functions are different, and hence this distributed algorithm is not symmetric.

Proxy Object:

```
if ( it is a local reference ) then
   if (the associated object is remote) then
      local_count = local_count + 1;
      send RMI request to the remote site;
   endif
endif
```

```
if (it is a remote reference) then
   remote_count = remote_count + 1;
endif
```

Monitor (Proxy)

```
for ( each proxy object in Object Graph) do
  if(the associated object is at remote site) then
     if( local_count > threshold ) then
       send (objID) to mobile client for moving this object
       receive(obj) from mobile client;
       pObj = object_cache.getObject(objID);
       pObj.local_count = 0;
       pObj.remote_count = 0;
     endif
  else /* associated object is at local site */
     if( remote_count>threshold) then
        send (its associated object) to mobile client;
     endif
  endif
endfor
```

Monitor (Mobile)

```
receive(objID) from proxy server;

pObj = object_cache.getObject(objID);
pObj.remote_count = 0;
pObj.local_count = 0;


send(pObj.associatedObj) to mobile client;
receive(an associated object) from mobile;
```

This algorithm is realized in Monitors that run concurrently with the Object Server. The problem is how and when to execute this algorithm. There are two possibilities, one is synchronous, denoted by SRGS, which means that invoking remote methods and

migrating objects are executed synchronously. The other possibility is asynchronous, denoted by ARGS, meaning that there is no synchronization point between the execution of remote method invocations and object migration. In SRGS, the KMOT execution time is divided into two parts, which occur alternatively (Figure 4.11). One is the remote method invocation period (RMI period), the other is the object migration period (OM period). This approach almost has no advantage in the design of KMOT since we need an

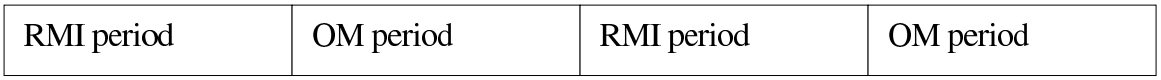


Figure 4.11 KMOT execution time

extra protocol to have the proxy server and the mobile client reach a distributed consensus, indicating that they both should enter RMI or OM period at the same time. This may degrade the system performance and complicate the system design. Therefore, in the design of KMOT we adopt ARGS, both Monitor and Object Server run independently, which one executes is decided by KVM thread scheduling. Furthermore, we require the Monitor sleep to one minute before every execution. However, it may induce problems in the case that some movable objects are currently being migrated, and a reference reaches the destination and at that time no proxy object has been created for that movable object. We call this problem *tracing problem*. The following example illustrates the scenario. D is on the way from the mobile client to the proxy server, but a

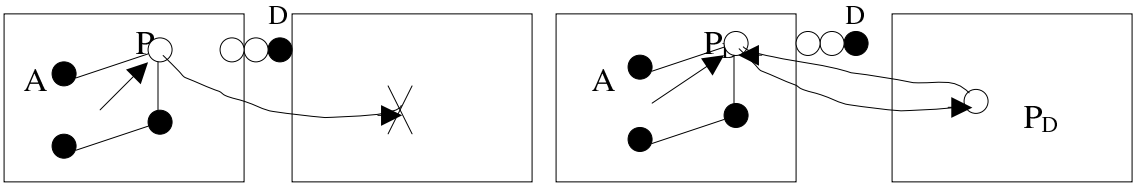


Figure 4.12 Tracing Problem in ARGS

reference to it reaches the proxy first. So far, there is no trivial solution to the *tracing problem* in KMOT. However, if there exists a proxy object for the on-the-way movable object, the execution is safe since the proxy object P_D in the Proxy can return the reference to the counterpart in the Mobile, and P_D in the Mobile forward this reference again. Hence, there is a loop between these two P_Ds until D arrives at the Proxy.

We can prove that RGA will converge at the shipping of the entire object graph under the condition that every object in the graph is a calling object. The proof is straightforward since the object graph is connected and every calling object will take its reference proxy objects, which are its direct neighbors, and this procedure is repeated until no object is left.

Although RGA executes efficiently, the application performance is not guaranteed to be improved and may be even worse than that without object migration because it only considers the reference counts, does not take the features of objects as well as the mobile environment parameter such as the capacity of wireless link into account. For example, in Figure 1.10(2), at the Proxy, if the remote reference count of Pc exceeds the threshold, the monitor will deliver C back to Mobile. This schedule degrades the overall performance since the references to B and E from C will become remote invocations. In order to exploit the advantage of object migration, an optimal decision strategy considering all these factors comprehensively is desired. In this thesis appendix, we investigate this problem in detail, set up a model and propose several algorithms to deal with it.

4. 6 Implementation

KMOT is a research platform to be designed for investigating the efficiency of adaptive application by offloading workload from resource-constrained portable devices to a powerful proxy machine. We have chosen a layered protocol to design it, and implemented it according to object-oriented principles. Our intention is to use this prototype as a testbed for a wide range of ideas to identify the efficiency of code mobility on resource-constrained portable devices.

The client side of KMOT is implemented on a Handheld PC (Hitachi SH3) with WinCE2.0 and a PalmIIIc with PalmOS 3.5. For WinCE, the development environment is IDE VJ++, which generates the KMOT class files and downloads them into WinCE. For PalmOS, we develop KMOT using the Java compiler with KVM1.0 API classes which can be downloaded from Sun Micosystem's web site. Our primary mode of operation is to use these two platforms as mobile clients. However, we also use the Intel PII with WinNT4.0 and Sun SparcStation 10 with Solaris as a platform to implement our toolkit.

The KMOT server is implement on Intel PII running Linux and Sun SparcStation running Solaris. It is developed under the JDK1.0 or above environment.

4.7 Programming Model

KMOT offers applications a distributed object system based on a client/server architecture. Clients are KMOT applications that typically run on resource-constrained portable devices, but could run on stationary hosts as well. Servers typically run on stationary hosts and hold the long-term state of the system. KMOT supports adaptive mobile application through its runtime library, which consists of a set of application

programming interfaces. Therefore, from the perspective of the programmers, there is almost no difference between programming on DOMT and on KMOT. However, KMOT has its own object serialization protocol, which requires the movable object to take part in the serialization process. Moving objects is not totally transparent to the programmer. An example for programming for a movable object is discussed in Subsection 4.2.1. The movable object might be as simple as a thin wrapper object with its associated operations or as complex as a module that encapsulates part of an application. Defining the movable objects is the first step to programming on KMOT. There are several factors that influence this decision. First is the coupling of an object with its direct reference objects. Second is the computational load of the object. Objects with high coupling are not prone to move, but high computational load will make it beneficial to migrate an object. Unfortunately, these two factors usually conflict since high computational load usually results in high coupling. Programmers should strike a balance to improve overall application performance. The final factor is related to the mobile environment; for example, if the bandwidth is high and the Proxy CPU is very fast compared to the Mobile CPU. We may benefit from moving more objects, but not always. KMOT provides programmers with an API *moveObjectTo* to move objects explicitly. When the object is in the local site, this API will move it to the remote side. Otherwise, this API moves the object to the local site. The second step is to generate proxy objects for the movable objects. This step is slightly different from programming DOMT, in which every movable object should have a proxy object. However in KMOT, if a movable object is not referenced on its original site again after it moves away, it dose not need a proxy object since the function of proxy objects is just to provide transparent access to the real

objects. For example, in a MP3 Player running on a mobile device, if its decoder is shipped to the remote proxy server, the player has to send the frame one by one to the remote decoder for decoding. In this situation, a frame is a movable object, which will not be referenced again in the player after it goes to the decoder. Therefore the frame does not need a proxy object and is referenced directly in the decoder. Although this strategy can improve performance, programmers have the responsibility to figure out this kind of objects.

KMOT monitors can make migration decision and migrate objects in the background for the users. This relieves the programmers from the burden of scheduling the moving of objects.

4.8 An Execution Scenario

Figure 4.13 shows the data flow of a remote method invocation accompanying an object migration between KMOT components. This scenario provides a comprehensive picture

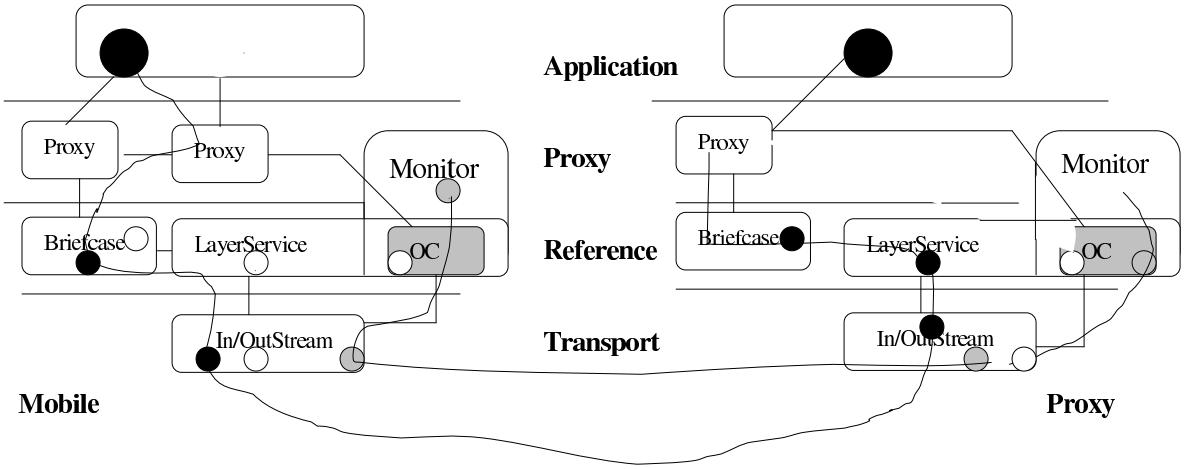


Figure 4.13 KMOT Execution Scenario

of how KMOT works. An object migration request is initiated by the Monitor on the proxy server, and sent to the mobile client through its Transport Layer (The bold line shows this process, while the grey points indicate the moving objects). When the request is received by the Monitor on the mobile client, the Monitor will retrieve the proxy object from its object cache, encapsulate it in a briefcase object to the LayerServer in the Reference Layer, and return this briefcase object to the proxy server through its Transport Layer. The Transport Layer on the proxy server will de-serialize the briefcase object to obtain the proxy object which contains its real object, and put it directly into its object server (The dot line shows this process, and the white points indicate the moving objects). As soon as a remote method invocation of the moved object occurs at the mobile client, this request is encapsulated in a briefcase object and forwarded to the LayerService of the Reference Layer. The LayerService is responsible for resolving the semantics of this invocation and transfers the briefcase object to the proxy side through its Transport Layer. The Transport Layer on the proxy server will first receive this request, and deserialize it. The request is resolved in the Reference Layer's LayerService, and invokes the method of the specific object by simulating the class reflection procedure. The response will be returned along the reverse direction. (The dashed line shows this process, and the black points indicate the moving objects). Object migration and remote method invocation can execute concurrently and asynchronously.

4.9 Limitations

Although KMOT provides support for designing adaptive applications on resource-constrained portable devices, it is still in the infant phase. This is evident by some

limitations that are not overcome yet. First, object serialization is not transparent to the applications, which will require the support from the programmer to serialize each movable object. Also, this serialization protocol is incompatible with the standard one, and therefore prevents integration with a system adopting the standard serialization protocol. So far only one-dimensional arrays can be transferred. Second, for a remote method invocation, returning an array of objects is not available now. Third, for realizing stateful migration, every movable class needs to have at least one non-default constructor for creating a new object. Finally, Proxy Objects cannot be created on the fly, and a tool is designed to create them offline automatically.

Chapter 5

Evaluations and Results

The Earlier chapters on KMOT introduced its architecture and its fundamental functionality to support adaptive applications. Although this architecture possesses the features to fit the constraints of PalmPilot, the performance improvement will depend on the relative cost of moving objects and exploiting the fast proxy processor.

This chapter will give evaluations to validate the benefits for mobile code deployed on resource-constrained portable devices for adaptive purposes. Our approach is from two aspects: functionality and performance.

5.1 Functionality

We compare KMOT with two other mobile code toolkits DOMT and Voyager (which will be discussed in the next chapter) to obtain the functionality comparison. The rational behind this choice is that DOMT is the ancestor of KMOT, and Voyager is a Java agent-enhanced Object Request Broker (ORB), which represents a commercial product in the market. The limitations to these toolkits are that DOMT can only run on Handheld PC or more powerful computer. Voyager requires more resources to execute than DOMT, and is therefore only executable on computers with sufficient resources like workstations, desktops and notebooks. Both DOMT and Voyager are Java-based, depending on the

underlying JVM supports. Unlike these toolkits, the design of KMOT originally focused on the resource constraints and the cross-platform executions. Hence, KMOT can run on a broad range of Java-enabled platforms from workstations to palm-size devices without any modification. Due to its KVM-based property, it does not resort to some standard technologies such as object serialization and class reflection adopted by DOMT and Voyager to realize object migrations and remote method invocations. It has its own serialization protocol and reflection simulation to achieve the same goal. Therefore, KMOT is more autonomous than the other two. This autonomy naturally leads to its platform-independent property.

Another advantage enjoyed by KMOT is its small footprint and its lightweight execution on portable devices with constrained resource such PalmPilot. DOMT is designed for WinCE and has a bigger footprint than KMOT. Voyager has the biggest footprint among the three toolkits due to its flexibility and capacity.

Since DOMT and Voyager rely on the standard object serialization protocol, the performance will degrade dramatically if multiple object subgraphs are migrated to the remote sites in different migration sessions. This performance anomaly is overcome in KMOT by its own serialization protocol.

Comparing DOMT and Voyager, the limitations of KMOT are mainly its limited support for array migration and its application-aware migration as we described in the previous chapter. These limitations do not result from our design or implementation, but are intrinsic to the lack of class reflection in KVM. The functionality comparisons are summarized in the following table.

Table 5.1 Comparison between the three Toolkits

	KMOT	DOMT	VOYAGER
Platform	WorkStation, Desktop, Notebook, Handheld, Palm	Workstation, Desktop, Notebook, Handheld	Workstation, Desktop, Notebook.
Object Migration	Yes	Yes	Yes
RMI	Yes	Yes	Yes
Performance Anomaly	No	Yes	N/A
Array Migration	One Dimension	Multiple Dimensions	Multiple Dimensions
Application-aware	Yes	No	No
Footprint	56K	224K	2620K

5.2 Performance

In order to evaluate the performance of KMOT under the mobile settings quantitatively, we distinguish two kinds of factors, which impact the overall performance. The first is related to the mobile environment, including platforms and wireless bandwidth. We chose platforms from desktop to PalmPilot with different CPU capacities. We also adjust the link bandwidth to various values to simulate a range of wireless environments. The second relates to KMOT itself: its performance relies on its basic mechanism such as object migration and remote method invocation as well as its scheduling strategy. Based on these observations, we conducted experiments to test several benchmark algorithms under various mobile environments.

5.2.1 Experimental Setup

The experiment system configuration (Figure 5.1) consists of two hosts connected via a LAN with 10 Mbps bandwidth and two mobile devices. One of the hosts is a Sun Ultra-1 with 92 MB memory running as a KMOT proxy server. The other is a Pentium II PC running Windows NT4.0 with 130 MB memory at 233 MHz. The function of this

computer is twofold. One is to act as the KMOT mobile client. The other is to act as the desktop for the mobile devices to access the proxy server. The mobile devices in our experiment are either a Handheld PC (Hitachi SH3/16M) running WinCE 2.0 or a PalmIIIc with 8MB RAM running PalmOS 3.5. The Handheld PC and the PalmPilot connect to the desktop computer via the serial cable and the Hotsync cradle respectively. They access the proxy server via the software Mocha Win32 PPP [33] preinstalled on the desktop computer. Mocha Win32 PPP permits configuring the baud rate of the serial link. We use this function to fluctuate link speed to simulate the wireless connections.

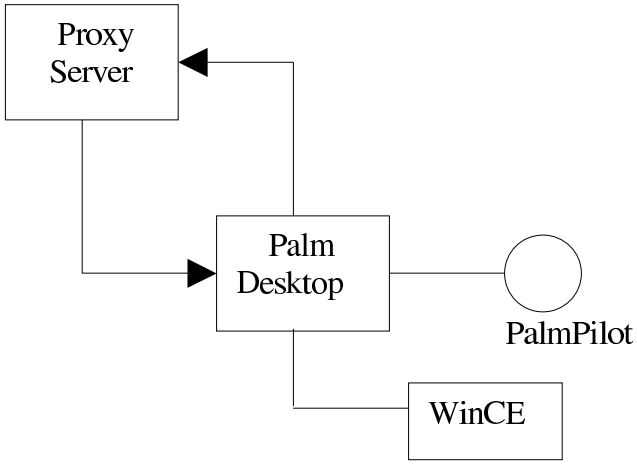


Figure 5.1 Experimental Testbed Configuration

5.2.2 Benchmarks

Ideally, we hope to find some non-trivial KVM-based applications as our benchmarks to evaluate our toolkit. However, these applications are hard to find at this time, partially because the release of KVM is fairly recent, and the limitations of KVM prevent non-trivial applications from being deployed on it easily without extra support mechanisms. This fact causes some difficulties to choose benchmarks for the experiments. An application, which is suited as a benchmark should satisfy at least two conditions. First, it

must run well on the KVM without resorting to other support. Second, it must reflect some common properties of the applications on the KVM. Our benchmarks are two popular algorithms in many applications: the binary search algorithm and the quick sort algorithm. The performance of these two algorithms can properly give us some clues to evaluate an application on KMOT due to their popularity. Another problem is the object size, which, we think, is a relevant parameter of the experiments. Unfortunately, we have no idea about what size objects are amenable to be migrated. We obtain some hints from [32] in which object size is treated as a parameter of the dynamic partitioning of the application between the mobile client and its proxy server. Based on their placement decision, for wireless environments, the authors claimed that only very small objects less than 2KB should be placed on the mobile clients for performance improvements. Therefore, we limit our movable object size to less than or equal to 2KB, which indicates that the level of the binary tree in the search algorithm is at most 6 and the number of the elements in the array in the sort algorithm is around 200.

5.2.3 The Performance of Basic Mechanisms

Our basic mechanisms include object serialization protocol, class reflection and remote method invocation. KMOT is based on these mechanisms. Therefore, their efficiency will naturally have great impacts on KMOT overall performance. To quantify these impacts, we try to minimize the effects of other factors such as network traffic, bandwidth, and processor speeds. We configured the mobile client and its proxy server on the same machine, i.e., the desktop computer in our experimental setup. In addition to our toolkit, the other two toolkits, DOMT and Voyager, are employed for comparison purpose since

they both exploit the standard serialization protocol and class reflection mechanism to realize remote method invocation.

5.2.3.1 Object Serialization and Migration

In order to evaluate our serialization protocol, we decompose it into several phases. The object is first serialized into a byte stream, and then we transfer this stream to a remote side. Finally, the remote side de-serializes the stream and recovers the object. Hence, the efficiency of the serialization and de-serialization as well as the size of the byte stream is critical to our consideration. We first compare our stream size with those produced by the other two toolkits and then the efficiency of serialization/de-serialization between our protocol and the standard protocol are compared by serializing a binary tree object with level from 1 to 10.

Table 5.2 Comparison between Serialization Protocols

Tree Level (sz)	KMOT Serialization (B)	Standard Serialization (B)
1 (32B)	49	107
3 (224B)	241	287
5 (1KB)	1009	1007
6 (2KB)	2033	1967
8 (8KB)	8177	7727
10 (32KB)	32753	30767

Table 5.2 demonstrates the stream sizes after serializing, which indicate that our protocol is more efficient than the standard one under the condition that the serializable objects are small enough. However, with the increasing object size, the efficiency of our protocol is degraded gradually, compared to the standard one. For example, in level 10, to serialize an object with size 32KB, there are about 6.5% efficiency losses in the stream size compared to the standard protocol.

To obtain insight into the performance of our serialization protocol, we measured the

Table 5.3 Tree Moving

Tree Level	KMOT (ms)	DOMT (ms)	VOYAGER (ms)
1 (32B)	600	10	65
3 (224B)	800	15	95
5 (1KB)	2350	40	103
6 (2KB)	4550	70	110
8 (8KB)	16030	205	123
10 (32KB)	60690	613	300

Table 5.4 Array Moving

Array size	KMOT (ms)	DOMT (ms)	VOYAGER (ms)
10	500	35	60
50	650	40	75
100	950	60	88
200	1400	70	100
400	2350	95	118

costs for moving different structures, binary trees with various levels, representing the recursive structure, and arrays with various size, representing the flat aggregate type. These structures are typical since other object types can be reduced to them. Table 5.3 and 5.4 display the time for moving trees and arrays respectively under the three toolkits. We can make several observations from these numbers.

KMOT has the lowest performance. This fact does not surprise us since our protocol is realized outside of KVM, unlike the standard protocol, which resorts to the Java native code techniques. However, the degree of the performance degradation is beyond what we initially expected. For instance, to move a tree with level 10, the stream size produced by our protocol is just 6.5% bigger than that produced by the standard one. However, the overall performance for migrating the tree in KMOT is about 100 times slower than

DOMT, and 200 times slower than Voyager. The same situation also happens when moving the flat structure like an array. There are several reasons for the high overhead. The first is the application-level implementation of our protocol, as we described above. The second is the extra logic in our protocol for constructing the distributed object graph. This logic is executed repeatedly as soon as an object is de-serialized, which is a time consuming task, especially for the recursive or flat aggregate structures. Finally, unlike the standard protocol, our protocol is not optimized for the object to be transferred.

Another observation is that our protocol is amenable to moving very small objects, say the object size is less than or equal to 1KB. It is not only because of the stream size it produced but also because of the overall performance of moving that object. For example, the performance for moving a binary tree with level 5 is less than 60 times slower than DOMT, and around 20 times slower than Voyager, which are much better than moving the binary tree with 10 level as we discussed above. By comparing these two tables, if the bandwidth is always available, sending a flat structure like array can obtain more benefits than sending a recursive structure like a binary tree in KMOT. The reason is that our serialization process is recursive, and hence, only one object is processed in an invocation of *writeObject( )* or *readObject( ),* whereas serializing and de-serializing an array of objects bypass the extra logic to construct the distributed object graph and process the elements in batch in one invocation of corresponding methods.

Finally, no matter what the type of the object structure, the performance of migrating an object in DOMT surpasses that in Voyager only if the object size is less than 2KB. This conclusion is not valid if the object size is bigger than 2KB in our scenarios. This demonstrates that Voyager possibly optimizes its protocol to migrate big objects.

5.2.3.2 Class Reflection

As we discussed in the previous chapter, class reflection is not provided in KVM. However, this mechanism is mandatory in KMOT. We simulate it by adding some logic to each movable object's proxy object, and make this simulated mechanism our basic support to design the toolkit. Since remote method invocation will reference this mechanism in a local manner, we can only investigate the efficiency of local method invocation to evaluate its performance. We inherit the methodology and experimental setup of the previous subsection to do our experiments.

Table 5.5 Local Method Invocation (Searching)

Tree Level	KMOT (ms)	DOMT (ms)	VOYAGER (ms)
1 (32B)	5	10	10
3 (224B)	30	50	75
5 (1KB)	175	200	330
6 (2KB)	340	335	425
8 (8KB)	1332	1171	1064
10 (32KB)	5648	4887	3835

Table 5.6 Local Method Invocation (Sorting)

Array size	KMOT (ms)	DOMT (ms)	VOYAGER (ms)
10	20	50	28
50	100	115	106
100	195	215	203
200	395	430	400
400	781	800	800

The results displayed in Table 5.5 and 5.6 demonstrate the performance of our reflection mechanism. Java reflection resorts to the native code support whereas our method is to hardcode the real object information into its proxy object. Our advantage is to avoid the actual reflection invocation but achieve the same function. However, the simulation is

realized at the application level. Therefore, there must exist a tradeoff. In our test scenario, no matter what type of the object, if its size is less than 2KB, our mechanism outplays the standard reflection adopted by DOMT and Voyager. Standard reflection might optimize method invocation code and outplay our mechanism provided that the method contains big objects as its parameters.

5.2.3.3 Remote Method Invocation

Finally, we consider the remote method invocation under the condition that there is no bandwidth limitation, no network traffic and processor effects. In this situation, the performance of remote method invocation can be viewed as the overall performance of our basic mechanisms since it involves object serialization, class reflection, and local method invocations. Table 5.7 and 5.8 show the performance results of the three toolkits.

Table 5.7 Remote Method Invocation (Searching)

Tree Level	KMOT (ms)	DOMT (ms)	VOYAGER (ms)
1 (32B)	50	140	110
3 (224B)	95	140	130
5 (1KB)	330	170	200
6 (2KB)	521	210	320
8 (8KB)	1833	435	962
10 (32KB)	7331	1031	3725

Table 5.8 Remote Method Invocation (Sorting)

Array size	KMOT (ms)	DOMT (ms)	VOYAGER (ms)
10	50	180	60
50	150	225	95
100	270	302	120
200	460	335	193
400	770	465	310

Unlike the performance of object migration, KMOT does not always have the worst performance among the three. For very small objects, the performance of KMOT is slightly better than the other two. This fact is partially consistent with the local method invocations. But for larger objects, the performance of KMOT degrades dramatically. The reason for this phenomenon is partially from what we discussed in the local method invocations. On the other hand, the low efficiency of our object serialization and an extra dynamic binding cost in the simulation of class reflection also account for the additional overhead in the remote method invocations.

Up to now, we can conclude from the experiments conducted that our basic mechanisms are amenable to migrate small objects and invokes method remotely or locally incurred by these objects. In the following section, we will investigate the impacts of the mobile environment on our toolkit. These impacts include the platform types, and the variation of the bandwidth.

5.2.4 KMOT in Mobile Environment

In this section, we conducted experiments to evaluate our toolkit under a mobile environment characterized by different bandwidths and platform types. The experimental results indicate that the bandwidths have a great impact on object migration, especially for the big objects, and PalmPilot is more sensitive to the changes of bandwidths than Handheld PC.

5.2.4.1 Bandwidth Effects

The purpose of our experiments is to measure the performance of the KMOT basic functionalities under various bandwidths. We still adopt the binary search algorithm and the quick sort algorithm as our benchmarks. We executed the proxy side on the Sun

Ultra-1 station, and installed the mobile side of KMOT on PalmIIIc as described above. In this experiment, the bandwidth is an important factor. We investigate the performance of object migration with various sizes and compare the performance of method invocation between the local site and the

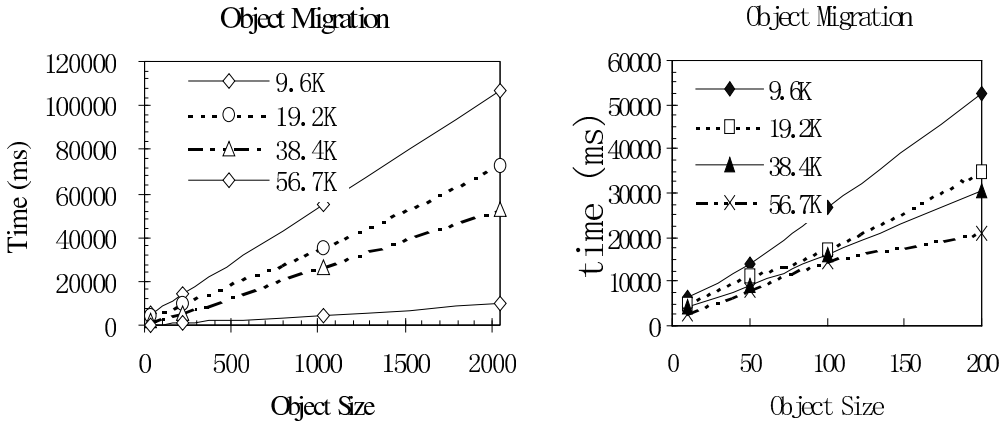


Figure 5.2: KMOT Object Migration

remote site. Figure 5.2 shows the performance of object migration of these two benchmarks. The unit of object size in the left figure is bytes, whereas, in the right figure, we use the array size to indicate the object size. We measure the performance under link baud rates from 9.6 Kbps to 56.7 Kbps. The results show that with the increment of bandwidth, the migration performance improves regardless of the object structures, and for a specific bandwidth, the time consumed to migrate an object is proportional to the object size.

Figure 5.3 shows the performance of RMI in KMOT. The results show that we cannot benefit from RMI unless proxy computation power or available bandwidth exceeds some value specific to the application. The cost of RMI for searching is almost constant with the increment of input size. This is because the proxy server is fast and the amount of

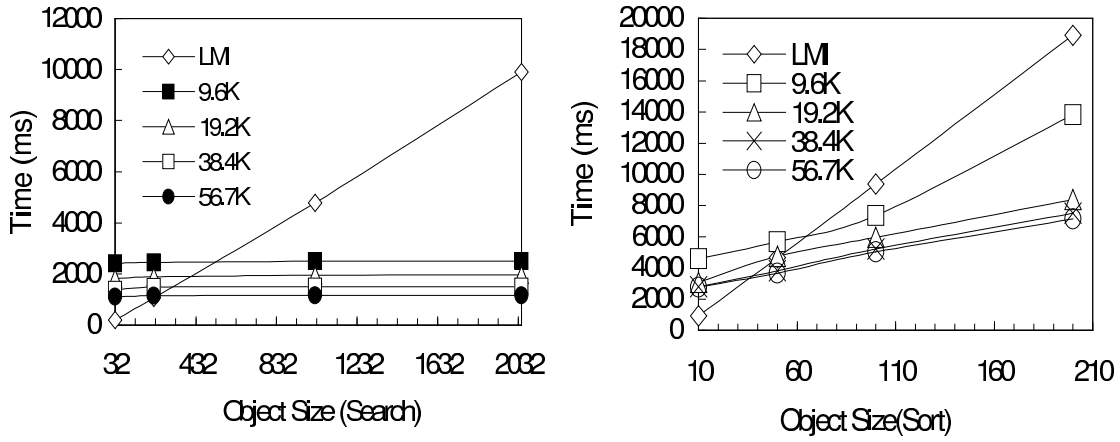


Figure 5.3 RMI Performance

data to be sent back is constant. On the contrary, sorting will return the entire array again, and has worse performance than searching. Figure 5.3 also reveals another fact: with increasing bandwidth, its impact on RMI will be reduced compared to the proxy server speed.

5.2.4.2 Platform Effects

We vary the platforms to execute KMOT to understand the platform effects on our toolkit performance. Our candidates are Handheld PC and PalmIIIc, representative PDAs in the market. In these experiments, we fix our bandwidth as 56.7 Kpbs and 19.2 Kpbs. The benchmark we adopt here is only the binary search algorithm.

Table 5.9 Migration (56.7Kpbs)

Platform	1 (32B)	3 (224B)	5 (1KB)	6 (2KB)
HPC (ms)	2000	3500	7000	12000
PalmIIIc(ms)	1047	3520	13417	26523
Ratio (P/H)	0.5235	1.006	1.917	2.210

Table 5.10 Migration (19.2Kpbs)

Platform	1 (32B)	3 (224B)	5 (1KB)	6 (2KB)
HPC (ms)	2000	4000	8300	14000
PalmIIIc(ms)	4500	9960	34790	72990
Ratio (P/H)	2.25	2.49	4.191	5.213

The results for moving binary trees under different bandwidths are summarized in Tables 5.9 and 5.10 respectively. From the ratio of P/H, one can see that the performance on the Handheld PC surpasses that on the PalmIIIc under almost all the conditions except the movable is extremely small. At high bandwidth, this ratio reaches 2.2, which means the object migration between the Handheld PC and its proxy server (Sun Sparc) is 2.2 times faster than that between the PalmIIIc and the same proxy server. At low bandwidth, this ratio reaches 5.2. This fact illustrates that the Palm is more sensitive to the bandwidth changes than the Handheld PC. In other words, with low-bandwidth networks, the gain from using a Handheld PC becomes much larger. Another observation from these results is that the bandwidth has greater adverse effects on moving bigger objects than moving smaller ones, which is consistent with our intuitions.

Table 5.11 RMI (56.7 Kpbs)

Platform	1 (32B)	3 (224B)	5 (1KB)	6 (2KB)
HPC(ms)	1000	1000	1000	1000
PalmIIIc(ms)	1117	1140	1160	1167
Ratio (P/H)	1.117	1.14	1.16	1.167

Table 5.12 RMI (19.2 Kpbs)

Platform	1 (32B)	3 (224B)	5 (1KB)	6 (2KB)
HPC(ms)	1000	1000	1500	1500
PalmIIIc(ms)	1820	1880	1950	1960
Ratio (P/H)	1.82	1.88	1.95	1.96

The results of RMI under different bandwidths are shown in Table 5.11 and 5.12. The performance ratio between these two platforms ranges from 1.117 to 1.167 under the higher bandwidth (56.7 Kpbs), from 1.82 to 1.96 under the lower bandwidth (19.2 Kpbs). The ratio increases with increasing object size, but this increment is not so noticeable as that in the previous tables. This result demonstrates that if a RMI does not involve big objects to be moved around, it is relatively insensitive to the bandwidth changes compared to the object migration.

5.2.5 Performance of Scheduling Strategy

Another critical factor affecting the performance of KMOT is the strategy for migrating the objects. The purpose of experiment in this subsection is to evaluate RGS adopted in KMOT. The benchmark is a reduced version of an arbitrary application. KMOT creates the partial object graph in its object database. The monitor schedules the objects to the

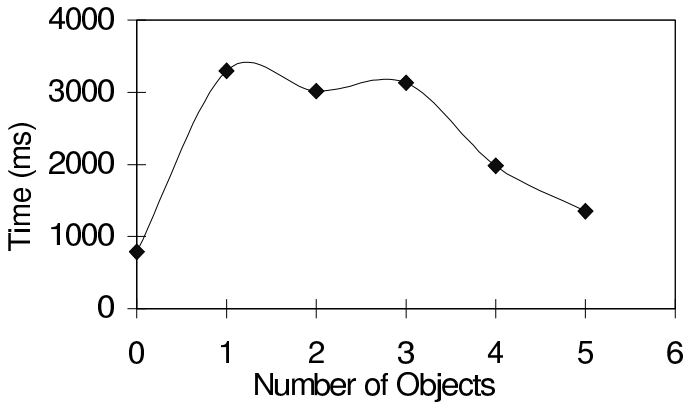


Figure 5.4 RGS Performance

remote proxy server according to RGS. Figure 5.4 shows the performance changes of RMI with the increment of the number of objects to be moved. It reveals that the RMI performance is not guaranteed to improve. This is not surprising since an inappropriate

migration may incur extra remote method invocations and degrade the performance dramatically. When coupling objects coreside on the same site, a related method invocation will have minimal remote accesses. RGS can move all coupling objects to the remote side to reduce the invocation costs. We refer to this kind of migration as *complete migration*, otherwise, we call it a *partial migration*. The results in this experiment also give us some hints that high coupling objects should be grouped together as a unit for shipping. For remote method invocation, complete migration is preferred.

Overall, it is not always beneficial to ship code to a more powerful proxy server to gain performance. The benefits depend on network characteristics, and relative CPU and I/O speed as well as the structure of application programs.

Chapter 6

Related Work

The motivation to design KMOT is to support adaptive application on resource-constrained portable devices. It achieves this goal by adopting the middleware architecture and mobile code techniques to realize both object migration and remote method invocation. KMOT realizes an application-aware adaptation strategy. This strategy permits individual applications to determine how best to adapt, but preserves the ability of the system to monitor resources and to enforce allocation decisions. We choose this design strategy for adaptive applications with an attempt to achieve better overall performance of our toolkit. Many other researchers adopt different strategies and follow different implementation technologies to design their systems for the same purpose. In this chapter, we will survey some recent efforts in this field to give an outline of the background picture to our toolkit. We first examine the mobile-aware adaptation, and then discuss several existing mobile code toolkits. We think this consideration is reasonable in that mobile code toolkits do not lead to the automatic support for adaptive applications if they have not an adaptive mechanism inside them.

6.1 Mobile Aware Adaptation

The computation of clients and their proxies has to be adaptive in response to the changes in the mobile environment [26]. This adaptation can be achieved in different components of the systems with different strategies. The range is delimited by two extremes [44]. At one extreme is Laissez-faire adaptation, which means adaptation is entirely the responsibility of individual applications without any support from the operating system. The other extreme, application-transparent adaptation, places the entire responsibility for adaptation on the system. Between these two extremes lies a spectrum of application-aware adaptation. This approach supports collaborative adaptation between the applications and the underlying system.

6.1.1 Application-Transparent Adaptation

Many traditional client-server applications are based on the assumption that the environment of the client does not change during the computing process. The approach of application-transparent adaptation attempts to facilitate the migration of these applications to the mobile world without any modification. The philosophy of this approach is that any adverse effects of mobile environments can be hidden by some additional components such as proxies, which are independent from the applications. The typical projects adopting this strategy are Coda [45], Little Work [19] and WebExpress [20]. In these projects, a local proxy runs on the mobile host and provides an interface for regular server services to applications. For example, in the Coda system, a file system proxy hides mobile issues from applications and emulates file server services on the mobile computers. The file system proxy is implemented as a user-level process called Venus whose function is to manage a file cache on the local disk on each client.

The applications interact with this proxy through the standard file system API as if they work on the real file system. The proxy handles the requests from the applications by managing the cache and mitigates the adverse effects of mobile environments.

WebExpress adopts a similar approach, in which two components, the Client Side Intercept (CSI) and the Server Side Intercept (SSI) are inserted into the data path between the Web client and the Web server to enable Web browsing applications to function over wireless links without imposing any changes on browsers and servers.

The Mowgli project [28] also supports Web applications over wireless links. A specialized HTTP agent and a specialized HTTP proxy are installed on the WWW client and the WWW server respectively with the aim of reducing unnecessary message exchange as well as the volume of data transmitted over the wireless link, and supporting disconnected operations. With the agent-proxy approach, neither Web clients nor servers need to be modified.

Compared to KMOT, these systems can be viewed as data migration systems with adaptation strategies realized in its proxy process. In these systems, the movable object is the data such as the server files in Coda and the Web pages in WebExpress and Mowgli. Although data mobility can support adaptive application to some extend, it does not allow for computation migration, as we suggest in KMOT. Hence, the cost of communication and computation can not be traded off gracefully in these systems.

6.1.2 Application-Aware Adaptation

Although the application-transparent adaptation has as main advantage that the mobile issues can be handled by the system without imposing any changes on the applications, there are important situations where it is inadequate. For example, a movie player can not

display the images in full-motion color if the bandwidth is below a certain value. Application-transparent adaptation fails to handle this situation because the adaptation mechanism has no knowledge of the data semantics and the application behavior. However, if the application is capable of displaying the image in slow-scan black and white, it could automatically do so when bandwidth falls below a critical threshold. Application-aware adaptations can handle this situation by resorting to the cooperation between the operating system and the application in various ways. The minimum system support should include (a) notifying the application of any relevant environmental changes and (b) providing a central point for resource management. Changes are modeled as asynchronous events, which can be detected either within the kernel or at the user-level. It is the application's responsibility to react to these events. For example, in the Odyssey system [36], the application negotiates and registers a window of tolerance with the system for a particular resource. The resources in question may be generic, such as network bandwidth, cache space, process cycles, or battery life. The system monitors the resource levels and notifies the application as soon as that resource rises above or falls below the limits in the tolerance window. The application will capture this notification and adapt its behavior to these changes by changing fidelity, which is an application-specific notion of the "goodness" of a computed result or data object. The adaptive strategies are totally up to the applications. This approach provides a flexible fashion for applications to react to the changes of the mobile environment. However, like Coda, the adaptation in Odyssey is also data-oriented.

In addition to Odyssey, a number of similar approaches have also been discussed in the literature. In the Prayer system [4], the application-aware adaptation is supported with

the use of abstractions: QoS class and adaptation blocks. A QoS classes is defined by specifying the upper and lower bounds for resources. An application divides its execution into adaptation blocks. An adaptation block consists of a set of alternative sequences of execution, each associated with a QoS class. Another approach is implemented by Welling and Badrinath [52] under an event delivery framework. In this framework, a notification subsystem, called event channel, delivers different events that are generated by the environment monitor to applications based on delivery policies. The applications are notified of the events to react to the environmental changes.

Unlike KMOT, most of these systems employ system-level adaptation. For instance, Coda inherits many of the usage and design assumptions of Andrew File System (AFS), and Odyssey extends UNIX with a small but powerful set of extensions for mobile computing.

6.2 Mobile Code Toolkits

A mobile code toolkit is generally built upon one or more programming languages. These languages usually have some unique features as we discussed in Chapter 2 to support code mobility, and are enriched by some new APIs for the same purpose. The implementation of these APIs may require the extension of the language's interpreter or virtual machine. If our reference to a mobile code language implies its underlying execution engine such as the interpreter or virtual machine, both concepts, mobile code language and mobile code toolkit are interchangeable.

Mobile code toolkit can be categorized in several ways. For example, we can categorize them according to their forms of mobility, i.e., strong mobility or weak mobility, or the programming languages they support. Some toolkits allow mobile code

to be written in multiple languages; many allow mobile code to be written in only Java, which is the most popular mobile code language; and others allow mobile code to be written in some single language other than Java. For the comparison purpose with KMOT, here we primarily analyze toolkits implemented on Java-enabled platforms. However, other language supported mobile code toolkits are also mentioned briefly for completeness.

6.2.1 Java-based Toolkit

Algets [31]. Aglets was one of the first Java-based mobile code toolkit, in which the Java thread is its EU and the Java interpreter constitutes the CE. Algets dose not capture agent's thread state during migration, and hence only weak mobility is supported. Algets adopts variants of the TACOMA model for migrating agents, where agent execution is restarted from a known entry point after each migration. In particular, Algets uses an event-driven model. When an agent wants to migrate, it calls the dispatch method. The Aglets system calls the agent's *onDispatching* method to perform application-specific cleanup, kills the agent's threads, serializes the agent's code and object state, and sends the code and object state to the new machine. On the new machine, the system calls the agent's *onArrival* method, which performs application-specific initialization, and then calls the agent's run method to restart agent execution. Aglets uses the proxies to act as representatives for itself, and provides location transparency.

Sumatra [1]. Sumatra is an extension of the Java programming environment with the aim at supporting the implementation of resource-aware mobile programs. Sumatra supports strong mobility of Java threads. This mechanism is realized by extending the JVM with the capabilities of deciding the time and destination for code mobility, shipping of stand-

alone code and creating a copy of the execution thread at a remote site. It also defines object groups as dynamically created object aggregates, the unit of mobility. The advantage of this definition is to provide the programmer with the flexibility to control the granularity of the unit of mobility. Sumatra has a resource-monitoring interface, which can be used by applications to register monitoring requests and to determine current values of specific resources. This control manner enables programmers to explore different policy alternatives for adapting to mobile environments.

Voyager [36]**.** Voyager is a Java-based mobile code toolkit integrated with COBRA. Hence, it is an Object Request Broker (ORB) in nature. Voyager allows Java programmers to create remote objects as well as their proxy objects and provides a convenient way to interact, somewhat transparently, with the objects through these proxy objects. Voyager can move objects from host to host. When an object moves, it leaves behind a forwarder object that redirects any messages to the new location. "Agent" objects, unlike other objects, may move themselves autonomously by applying the *moveTo( )* method on themselves. Voyager moves the code and data of the agent, but not thread state, to the new location and invokes the desired method there. Therefore, Voyager supports weak mobility.

µCode. The design of µCode [41] was inspired by identifying precisely the benefits brought by mobile code in the design and implementation of distributed applications. It places more emphasis on fine-grained code mobility rather than mobile agents. Hence, it has small footprint and lightweight execution. From this perspective, it is much more similar to our toolkit than what we discussed above. In µCode, the basic operations enable creation and copy of thread objects on a remote µServer, and class relocation

among μServers. A μServer is an abstraction of the run-time support and represents a computational environment for mobile threads. Upon migration, thread objects retain their data state and lose their execution state. Thus, μCode only supports weak mobility. μCode supports code shipping and fetching of both code fragments and stand-alone code, with both synchronous and asynchronous invocation, as well as deferred and immediate execution of mobile code. In μCode, the unit of migration is the *group*, which is simply a container for classes and objects. This abstraction is reminiscent of TACOMA briefcases or, more closely, of the object-group abstraction found in Sumatra.

μCode is implemented in Java, and hence very amenable to Java-enabled PDA like PalmPilot with its small footprint. However, it lacks the mechanism to support adaptive mobile applications. It only supports distributed applications effectively such as active networks and network management.

Besides these toolkits, another Java-based toolkit, Mole [49] is noticeable due to its object migration strategy, which is described by a new concept called *island*. An island is the transitive closure over all the objects referenced by the main agent object. It is decided automatically upon the migration of the main agent object. This strategy is very similar to our complete migration strategy in KMOT design.

6.2.2 Other Language-based Toolkit

In addition to Java, there are several other programming languages, which can be extended to support code mobility. Mobile code toolkits based on these languages usually have functionality similar to the Java-based toolkits. In this subsection, we overview several of them sketchily to enrich our discussion.

TACOMA. In TACOMA, the Tcl language is extended to include primitives that support weak mobility. A piece of mobile code, which is called an agent, can be shipped among sites supporting the TACOMA system. The shipment is done in an encapsulating structure called *briefcase*, which contains a set of folders. The folder contents can be arbitrary data or code. The briefcase is sent to the new machine, which starts up the necessary computational environment and then calls a known entry point within the agent's code to resume agent execution. The type of agents can be system agent or mobile agent. The system agents are preinstalled and not movable so as to provide efficient system functionality such as compiling or interpreting mobile code. The mobile agent, typically written by an application programmer, uses the system agent to move to a remote host and install mobile code.

TACOMALite [27] is a scale-down version of TACOMA to extend to PDA environments such as PalmPilot and Windows CE devices. This extension overcomes the limitation of PDA memory and network management by encapsulating the limitation into its API functions.

Obliq. Obliq is an interpreted, lexically scoped, interpreted language. An Obliq object is a collection of named fields that contain methods, aliases and values. An object can be created at a remote site, cloned onto a remote site, or migrated with a combination of cloning and redirection. In Obliq, an agent is built upon these mobile objects and executes in a separate thread, which can also request the execution of a procedure on a remote execution engine. The code for such a procedure can be sent from the local site to that destination engine. In this case, the original references to the local objects are automatically translated into network references, allowing transparent access to the

objects distributed on a computer network. The sending thread suspends until the execution of that procedure terminates. Thus, Obliq supports weak mobility using a mechanism for synchronous shipping of stand-alone code.

Telescript. Telescript is an object-oriented language, which is similar to both Java and C++, and can be compiled into bytecodes for execution on a virtual machine. In Telescript, each network site runs a server that maintains one or more virtual places, which are the Telescript EUs. An incoming agent can enter the specific space under some security restrictions. A Telescript agent migrates with the *go* instruction, which captures the agent's code, data and thread state. On its new machine, the agent continues execution from the statement immediately after the go. Hence Telescript supports strong mobility. Although it has some advantages in security, and efficiency for migrating agents, Telescript has been withdrawn from the market, largely because it was overwhelmed by the popularity of Java.

Rover. The Rover toolkit offers applications a distributed system based on the client-server architecture. Clients are Rover applications that typically run on portable hosts and hold the long-term state of the system. The Rover toolkit provides mobile communication support based on two ideas: *relocatable dynamic object* (RDOs) and *queued remote procedure call* (QRPC). A relocatable 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. RDOs can be viewed as simple agents. 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.

From these perspectives, one can see that Rover is different from the mobile code toolkit discussed previously. It places much emphasis on handling the wireless connections for its application rather than on code mobility mechanism. Hence, the literature usually does not classify it into mobile code category.

6.4 Summary

In this chapter, we summarized some related work to the KMOT design. Coda and Odyssey are typical systems that realize the application-transparent and the application-aware adaptation to handle the changing amount of the resources available in the environment. The adaptation in these systems are data-oriented. They can not balance the computational load between the mobile clients and their proxy servers.

The current and future potential of mobile code technologies is mainly directed at implementing the mobile code paradigm MA. Although mobile agents can be used to conserve bandwidth, support disconnected operation and balance the workload, it suffers some drawbacks when applied in the portable devices with constrained resources. On the other hand, designing a well-defined mobile agent is a non-trivial work for a novice at programming, compared to developing a regular program. Other forms of mobile code such as REV or COD can avoid these problems, but do not support dynamic adaptive applications either. The mobile code toolkits we surveyed here verified this proposition since none of them provided mechanisms to monitor and notify the changes of mobile environment except Rover. Hence, these toolkits can not be applied to support adaptive applications directly.

KMOT combines application-awara adaptation and mobile code techniques to realize the object migration mechanism in the background as the fundamental functionality to

react dynamically to the mobile environmental changes. This feature distinguishes KMOT from the toolkits we discussed here (Rover excluded).

From the perspective of design principles, Rover toolkit is close to ours. Although its RDO concept has its counterpart, dynamic movable object, in KMOT, its QRPC gives it an advantage over KMOT. QRPC can be viewed as an asynchronous invocation for surviving disconnections, whereas the synchronous remote invocation can not make KMOT survive the disconnection. This limitation restricts KMOT to mobile settings where weak connectivity is the worst case. This situation also happens in μCode toolkit.

Java-based commercial systems like Voyager and Algets, always support weak mobility since strong mobility requires modifications to the standard Java virtual machine, which means that the toolkit could be used only with one specific virtual machine, significantly reducing market acceptance. KMOT enjoys this platform-independent feature by its middleware architecture. Another feature KMOT enjoys is its small footprint and KVM-based execution environment. μCode has small footprint, but whether it is KVM-based or not is not clear. TOCOMALite is designed for PalmPilot, but it based on Tcl, not Java. To our knowledge, KMOT pioneers the provision of support for adaptive applications on Java-enabled mobile devices with constrained resource like PalmPilot, SmartPhone, etc.

Chapter 7

Conclusion and Future Work

Mobile computing allows the user with portable devices to access the network services at any time without regard to location or mobility. However, due to the constrained resources in portable devices, the design and deployment of non-trivial mobile applications are complicated. How to cope with these constraints is a hot research area as well as a demand of the PDA market, especially with the advent of the PalmPilot. One promising technique to address this problem is mobile code. Code mobility can make mobile applications adapt to the context changes and hence improve its performance on mobile devices with the aid of a proxy server.

In this thesis, we presented our experiences from porting an existing mobile code toolkit for Windows CE (DMOT) to a new kind of emerging resource-constrained portable device, Palm IIIc. The new version of DOMT for this environment is called KMOT. In this chapter, we will summarize the work performed, present conclusions, and provide some suggestions for future research.

7.1 Thesis Contributions

KMOT is designed as a platform for mobile code applications on WinCE and PalmOS. Its framework was borrowed from earlier work on DOMT. However, due to the

limitations of KVM, our work is far from a simple portability exercise, and hence many issues have to be addressed. In particular, they are summarized as follows:

7.1.1 Object Serialization Protocol

The lack of object serialization in KVM prevents object migration in KMOT. In order to overcome this difficulty, a serialization protocol is designed, which adopts an externalization approach, asking the movable abject to serialize itself. In addition to this, a serializable interface and a new root serializable object are also defined in the protocol. Although this simple protocol allows any type of object to be moved, it imposes some workload on the programmer.

7.1.2 Class Reflection

Class reflection provides applications with the feasibility to investigate the object information on the fly. This mechanism is desirable in the design of KMOT. We simulate it by hardcoding the object information into its proxy object and defining an abstract ProxyObject class inherited by every proxy object to provide a unified interface for various remote method invocations, whereas the concrete implementation resides in the proxy objects. Our simulation avoids the cost of class reflection during the computing, but introduces extra logic into the proxy object.

7.1.3 Distributed Object Graph

Performance anomaly in DOMT is intrinsic to the standard object serialization. We get around this problem by proposing the concept of a distributed object graph. The construction of the distributed object graph requires a stateful migration, which means the relationship between objects that are delivered in different migration session must be kept in the object serialization process. In order to achieve this goal, our serialization protocol

is enriched to realize stateful migration, and the performance anomaly is eliminated in KMOT.

7.1.4 Distributed Recursive Method

Our goal in designing KMOT is to have a thin yet powerful client with functions comparable to its proxy server. DOMT adopts a symmetric architecture with the assumption that the mobile host and proxy host have the same computational power. Therefore, the DOMT architecture cannot be used directly by KMOT. We structure KMOT into a three-layer architecture, and propose a distributed recursive method to handle the nested method invocation. This method exploits the inherent order in the nested method invocations, and can be simplified to a single thread implementation. Using a single thread to realize the nested method invocation between the mobile client and the proxy server can reduce the overhead from thread switching, and the cost of thread synchronization.

7.1.5 Random Greedy Strategy

In order to react to the changes of the mobile environment, KMOT adopts a fully dynamic partitioning strategy, called Random Greedy Strategy, to schedule the objects to the remote proxy server. This strategy is designed based on the observation that an object always references its direct neighbors in the object graph. If the real object referenced does not reside at the local side, fetching it to local site may result in performance improvements in the forthcoming invocations.

7.2 Conclusions

Designing a mobile code toolkit for resource-constrained portable devices is not a trivial task. The difficulties stem from the tension between the limited resources provided and the plentiful functions required. KMOT is our effort to balance these two factors. In its implementation, the most inconvenient feature, is the lack of class reflection, which is required by object migration and remote method invocation. Nearly all the work involved in designing the fundamental support is directed at addressing this problem, and results in sub-elegant solutions such as the application-aware serialization protocol for object migration and hardcode and extra logic in proxy objects for remote method invocation. Actually, if class reflection were provided, all these mechanisms can be realized without the awareness of the applications and proxy objects. This fact indicates that class reflection is a powerful mechanism to design dynamic applications and we suggest Sun Microsystems to add it in the future version.

In spite of these deficiencies, KMOT exploits the KVM resources with the approaches discussed above to provide a proper set of functions to support adaptive applications. The efficiency of these functions to a great extent determines the applications set up above them. Hence, the evaluation of KMOT can allow us to not only obtain some insight into the efficiency of KMOT itself but also to deduce the feasibility of mobile code for adaptive applications. We conducted experiments for this purpose which will also satisfy our motivation. Our conclusion is that KMOT is a fine-grained mobile code toolkit to support small object migration and remote method invocations efficiently. This is a direct consequence of its basic mechanisms, object serialization and class reflection. In a mobile environment, CPU speed and bandwidth are important

factors. The quantitative analysis indicates that RMI is not always beneficial unless proxy computation power or available bandwidth exceed some application-specific value. It also shows the bandwidth will have different effects on KMOT installed on different platforms. For example, KMOT on PalmIIIc is more sensitive to the fluctuation of the link speed than on Handheld PC. So in an area with frequent changes of bandwidth, deploying KMOT on Handheld PC will gain more benefit than on PalmIIIc. Therefore, the soundness of applying mobile code to design adaptive applications is not absolute. It depends on the applications, the mobile environment as well as the platforms to execute the applications. Due to these diverse factors, flexibility is an important feature for the toolkit. The integration of object migration and remote method invocation in KMOT reflects this consideration, and provides a powerful basis for building mobile applications.

7.3 Future Work

A number of issues need to be addressed in the future work, some of which is currently under way. These issues can be generally categorized into two classes, enhancing KMOT functionalities and improving its performance.

Currently, the remote method invocation in KMOT is synchronous, which requires the maintenance of the connection between the mobile client and its proxy server during the invocation. Hence, KMOT can not survive a forthcoming disconnection. Disconnected operation is a typical function of mobile computing and supported by many mobile-aware systems such as Coda, Odyssey and Rover. In order for KMOT to adapt to a wide range of mobile settings, this function is highly recommended to integrate. Data hoarding and QRPC technologies in Rover can be referenced to realize this function.

KMOT supports weak mobility because the stack of a Java thread is not accessible. This limitation can be overcome by extending the Java virtual machine to support strong mobility. A side effect of this extension is to realize some mechanisms at the virtual machine level to monitor the changes of the mobile environments and notify the applications. The function and efficiency of this mechanism is expected to surpass the current monitor component. However, the extended virtual machine may be incompatible with the standard one, and the deployment of KMOT will become a problem.

Another approach to improve the performance of KMOT is to design an efficient strategy to schedule the objects to the remote site in order to improve the overall performance. Currently, the problem in RGS is the ignorance of object features such as its computational load and the relationship with other objects. This problem sometimes degrades the KMOT performance dramatically. We are working on this problem and are designing several new algorithms to overcome the above shortage. Some theoretical results have been obtained and the resulting algorithms are being integrated with our toolkit.

KMOT is implemented for PalmPilot in a simulation environment. Extending it to other Java-enabled resource-constrained portable devices, for example SmartPhones, in real wireless networks such as GSM or CDMA is another avenue of future research. This research will permit us to evaluate KMOT under various conditions in addition to the wireless bandwidth in our simulation, and allow KMOT to be deployed in practical environments.

In addition, there are several other points that need to be considered to improve KMOT functionalities. Examples include creating proxy objects on the fly, supporting multi-dimensional array migration, and optimizing the object serialization protocol.

References

[1] A. Acharya, M. Ranganathan, and J.Saltz, *Sumatra: A Language for Resource-aware Mobile Programs*, Lecture Notes in Computer Science, Mobile Object Systems, 1222, pages: 111-130, 1996

[2] ARDIS. *ARDIS Network Connectivity Guide*. Illinois, ARDIS, March 1992

[3] O.Angin. A.T. Campbell, M.E.Kounavis and R.Liao. *The mobiware toolkit: Programmable support for adaptive mobile networking* IEEE Personal Communications, Vol. 5, No. 4, pages: 32-43,Aug. 1998

[4] V. Bharghavan and V. Gupta, *A framework for application adaptation in mobile computing environments.* In proceedings of the 21st International Computer Software and Applications Conference (COMPSAC '97). IEEE Computer Society, New York, NY, pages: 573-579, 1997

[5] A.D. Birrell and B.J. Nelson, *Implementing remote procedure calls.* In Proc. ACM Symp. on Transactions on Computer Systems, pages 19-59, February 1984

[6] D. Box. *Creating Components with DCOM and C++*. Addison Wesley Longman, 1997.

[7] I. Brodsky. *The Revolution in Personal Telcommunications*. Artech House Pubilshers, Boston, London, 1995, ISBN: 0890067171

[8] L. Cardelli, *A language with distributed scope*. Computing Systems, vol. 8, no. 1, pp. 27-59, 1995

[9] A. Carzaniga, G.P. Picco, and G. Vigna, *Designing Distributed Applications with Mobile Code Paradigms.* In Proc. of the 19th Int. Conf. on Software Engineering (ICSE'97), R. Taylor, Ed. pages: 22-32, 1997

[10] *CDMA. Growth and Expansion of IS-95 CDMA*. CDMA Spectrum. The Journal

of the CDMA World, June 1997. http://www.cdg.org/magazines/spectrum/spec696

[11] C. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna, *Analyzing Mobile Code Languages*. Lecture Notes in Computer Science, Mobile Object Systems, 1222, pages: 93-109, 1996

[12] E. Davids, *Java Object Serialization in Parsable ASCII format,* AUUG-Vic/CAUUG: Summer Chapter Technical Conference 1998 http://www.metva.com.au/users/enno/papers/java_ser.html

[13] N. Davies, A. Friday, S.P. Wade and G.S. Blair. *L²imbo: A Distributed Systems Platform for Mobile Computing,* Mobile Networks and App., Vol. 3, No. 2, pages:143-156, Aug. 1998.

[14] A.DeSimone, M.C. Chuah, O-C. Yue, *Throughput Performance of Transport-Layer Protocols over Wireless LANs,* Proc. of IEEE GlobeCom, 1993

[15] J. Dollimore, *Object-Based Distributed Systems,* Draft material for 3rd edition of Distributed Systems-Concepts and Design, Dept. of Computer Science, Queen Mary & Westfield College, University of London, pages 1-45, 1997

[16] F.Douglis, *On the Role of Compression in Distributed Systems*. In 5th SIGOPS workshop on Models and Paradigms for Distributed Systems Structuring, Sept. 1992

[17] G. Forman and J.Zahorjan, *The Challenges of Mobile Computing*, IEEE Computer, Vol. 27, No. 4, pages: 38-47 Apr. 1994

[18] Grasshopper, http://www.grasshopper.de

[19] P. Honeyman, L. Huston, J. Rees, and D. Bachman, *The LITTLE WORK project.* In Proceedings of the 3rd Workshop on Workstation Operating Systems, Key Biscayne, FL, 1992

[20] B.C. Housel and D.B. Lindquist, *WebExpress: A system for optimizing Web browsing in a wireless environment.* In Proceeding of the 2nd Annual International Conference on Mobile Computing and Networking, 1996.

[21] IBM. *An Introduction to Wireless Technology*. IBM International Technical Support Center SG24-4465-01, October 1995

[22] T. Imielinski and B.R. Badrinath. *Querying in Highly Mobile Distributed Environments*. In Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92), 1992

[23] Evaggelia Pitoura and George Samaras, *Data Management for Mobile Computing* Kluwer Academic Publishers, 1998, ISBN: 0-7923-8053-3

[24] D. Johansen, R. van Rensesse, and F.B. Schneider, *An Introduction to the TACOMA Distributed System – Version 1.0*, Tech Rep. 95-23, Dept. of Computer Science, Univ. of Tromso and Cornell Univ., Tromso, Norway, June 1995

[25] A.D Joseph, A.F. deLaspinasse, J.A. Tauber, D.K. gifford and M.F. Kaashoek. *Rover: a toolkit for mobile information access* ACM Operating Systems Review, Vol. 29, No. 5, pages: 156-171, Dec. 1995

[26] R. Katz, *Adaptation and mobility in wireless information systems.* IEEE Personal Communications. Vol1, No.1, pages: 6-17, 1994

[27] Kjetil Jacobsen and Dag Johansen *Mobile Software on Mobile Hardware – Experiences with TACOMA on PDAs* Technical Report 97-32, Department of Computer Science, University of Tromsø, Norway, December. Nov. 1997.

[28] M. Kojo, K. Raatikainen, and T. Alanko, *Connecting mobile workstations to the internet over a digital cellular telephone network.* In Proceedings of the Mobidata Workshop. Rutgers University Press, New Brunswick, NJ.

[29] Kono and T. Masuda. *Efficient RMI: Dynamic Specialization of Object Serialization,* Proc. of the 20th Int. Conf. on Distributed Computing Systems (ICDCS 2000), pages: 308-315, 2000

[30] Thomas Kunz and Salim Omar *A Mobile Code toolkit for Adaptive Mobile Applications* Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2000), Monterey, California, USA, pages: 51-59, Dec. 2000

[31] D.B. Lange and M. Oshima. *Programming and deploying Java mobile agents with Aglets.* Addison Wesley, 1998. ISBN 0-201-32582-9

[32] Grace Hai Yan Lo and Thomas Kunz, *A case study of dynamic application partitioning in mobile computing - an e-mail browser,* discussion paper, presented at the OOPSLA'96 Workshop on Object Replication and Mobile Computing (ORMC'96), San Jose, CA, 6 pages, Oct. 1996

[33] Mocha Win32 PPP: http://www. mochasoft.dk.

[34] M.Mouly and M.B. Pautet. *The GSM System for Mobile Communications.* Published by authors, 1992. ISBN: 0945592159

[35] B.D. Nobel, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker, *Agile application-aware adaptation for mobility.* ACM SIGOPS Oper. Syst. Rev. 31, 5, 276-287

[36] ObjectSpace *Voyager core package technical overview*. ObjectSpace, Inc., Dec. 1997, Version 1. http://www.objectspace.com/

[37] S. Omar, *A Mobile Code Toolkit For Adaptive Mobile Applications,* Master's Thesis, Carleton University, Canada, 2000

[38] Open Software Foundation. *Introduction to OSF DCE: Release 1.1*. Prentice Hall, Englewood Cliffs, NJ, 1996. ISBN 0-13-185810-6

[39] Jay E. Padgett, Christoph G. gunther, and Takeshi Hattori, *Overview of Wireless Personal Communitions*. IEEE Communications Magazine, pages: 28-41, Jan. 1995

[40] M. Philippsen and M. Zenger. *JavaParty – Transparent Remote Objects in Java,* Concurrency: Practice & Experience, Vol. 9, No.11, pages: 1225-1242, 1999.

[41] G.P. Picco *μCode: A Lightweight and Flexible Mobile Code Toolkit*, http://mucode.sourceforge.net/

[42] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Huchinson, and Eric Jul. *Emerald: A general-purpose programming language.* Software – Practice and Experience, Vol. 21, No. 1, pages: 91-118, Jan. 1991

[43] A.K. Salkintzis, *A Survey of Mobile Data Networks*, IEEE Communications Survey, Vol. 2, No. 3, pages: 2-18, 1999

[44] M.Satyanarayanan, Mobile Information Access, IEEE Personal Communications. Vol.1, No.1, pages: 6-17, 1994

[45] M. Satyanarayanan, and J.J. Kistler, P. Kumar, M.E. Okasaki, and D. Steere, *Coda: A highly available file system for a distributed workstation environment.* IEEE Trans. Comput. Vol. 39, No. 4, pages: 447-459, Apr. 1990

[46] M. Shapiro, and P. Ferreira, Larchant-RDOSS: *a distributed shared persistent memory and its garbage collector.* Workshop on Distributed Algorithms, No. 972, Springer Verlag LNCS, pages: 198-214. 1995

[47] J. Siegal. *CORBA: Fundamentals and Programming*. Wiley, 2000, ISBN 0-471-29518-3

[48] W. Stallings, *Operating Systems, Internals and Design Principles*, Prentice-Hall International, Inc. 1998. ISBN 7-302-02976-8

[49] Straβer et al. *Mole—A Java Based Mobile Agent System*, 10th European Conf. on Object-Oriented Programming ECOOP'96, pages: 327-334, Jul. 1996

[50] Sun Microsystems. *The Java Language Specification,* Oct. 1995. http://java.sun.com/docs/books/jls

[51] S. Vuong, L. Mathy and M Toro *Architectural Approaches for Wireless Computing in the Internet*. Proceedings of OOPSLA '96 Workshop on Object Replication and Mobile Computing (ORMC '96) San José, California, Oct. 1996.

[52] G. Welling and B.R. Badrinath, *An architecture for exporting environment awareness to mobile computing applications.* IEEE Trans. Softw. Eng. Vol. 24, No. 5, pages:391-400, 1998

[53] J.E. White, *Telescript Technology: Mobile Agents.* In Software Agents, J. Bradshaw, Ed. AAAI Press/MIT Press, 1996

[54] A. Wollrath, R. Riggs, and J. Waldo. *A distributed object model for Java*. In 2nd Conf. on Object-Oriented Technologies and Systems (COOTS), pages: 219-231, Toronto, Ontario, Jun. 1996

[55] Zenel and D. Duchamp. *A general proxy filtering mechanism applied to the mobile environment,* Proc. of the Third Annual ACM/IEEE Conference on Mobile Computing and Networking, Budapest, Hungary, pages 248-259, Sept. 1997

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

[57] Peter W. Smith, *The Possibilities and Limitations of Heterogeneous Progress Migration.* Ph.D Thesis, University of British Columbia, Oct. 1997

Appendix A

1. Introduction

Floating some computational objects between a resource constrained portable device and its powerful proxy server to adapt to the changes of mobile environment so as to support adaptive application is the goal of KMOT [1]. KMOT accomplishes this goal by providing facilities to support object migration and remote method invocation. Object migration in KMOT can be done in two fashions, application-transparent migration and application-aware migration. Application-transparent migration is realized by the cooperation between the monitors residing at both sides, and application-aware migration is programmed by the application developers to move a specific object. However, no matter what fashion is used, object migration will incur two types of costs: the cost of execution of an object on a processor and the cost of inter-processor communication. In order to reduce the cost of inter-processor communication, the set of highly related objects should be moved into a single processor during the execution of those objects. To reduce the computational cost of a program, objects should be assigned to the processor on which they run faster. The two kinds of migrations can be incompatible. The problem, to which we refer as the assignment problem, is how to split a program dynamically into two parts, such that the collective costs due to inter-processor communication and to computation is minimized.

Harold Stone [6] once researched this problem at early time under a two-processor distributed system described by Fuller and Siewiorek [7]. He abstracted the program into a modified module interconnection graph, and proposed a static assignment algorithm with the aid of network flow algorithms to assign program modules between two processors. He concluded that the weight of a cutset of the modified graph equaled to the cost of the corresponding module assignment. This fact indicates that his algorithm is optimal. However, this algorithm can not be applied to our design in two aspects. First, the constraints in mobile environment have no place in the algorithm. Second, the static nature of the algorithm prevents the dynamic object migration required by KMOT. The first problem can be partially addressed by the algorithms proposed by Rao et al [8] whose efforts are to find the optimal assignment when there is a memory constraint on one of the processors. Their algorithm, in most cases, can reduce the size of the problem and thus make it feasible to find the memory constrained optimal assignment by enumerating all possibilities. Although this algorithm can be extended to handle other parameters of the mobile environment such as limited capacity of wireless link, it is also static in nature. A dynamic assignment algorithm, which can be used in our case is proposed by Bokhari [9] based on his assignment graphs. In this algorithm, he defined the phase of modular program, and reduced these assignment graphs in size to obtain the upper and lower bounds on the cost of the dynamic assignment. However, the difficulty of applying this algorithm to our design is the lack of place for mobile environment parameters.

All above efforts are inspired by the work of Harold Stone, and resorted to the network flow algorithms. Roughly speaking, combining of the work from Rao, et al. and

Bokhari can provide a feasible solution to attack the application-transparent migration in KMOT since it can construct module graph dynamically and split it into two parts to be allocated in the two processors for minimizing the collective costs, and at meantime, take the mobile environmental parameters into account. However, this combinatorial algorithm can not address the application-aware migration efficiently since an object chosen by application users to move will break the network flow algorithms' principle that requires the location of each object is not decided prior to the execution of the algorithm. DOMT [5] adopts an algorithm based on different a technique from the previous ones, called Greedy Graph Partitioning Heuristic (GGP) [4], which is strongly related to the subset optimization problem. This algorithm is static, not optimal, and has the same complexity $O(N^3)$ with Stone's algorithm in the worst case as well as the drawbacks we mentioned before when applied in mobile computing. The only advantage it enjoys is its simplicity. Another more straightforward algorithm RGS is adopted by KMOT. Although this algorithm executes efficiently, the application performance is not guaranteed to improve and may be even worse than that without object migration because like GGP, it suffers from the all drawbacks. Therefore, in order to support both application-transparent migration and application-aware migration as well as to exploit the mobile environmental resources efficiently, a high performance algorithm considering all these factors comprehensively is desired. In this appendix, we argue with particular emphasis on partitioning of an object graph to achieve this goal in a mobile environment that is characterized by its limited capacity of the wireless link. A partition is a group of related objects and generated by some initiators, which are movable objects selected explicitly by the application itself or implicitly by KMOT. Partitions float between the

mobile client and the proxy server for performance reason. Our method is to reduce this optimization problem to a variation of the 0/1 Knapsack Problem [2][3], in which the restriction that the objects are independent is broken. We proposed several heuristic algorithms based on graph search strategies to fulfill the various requirements of the mobile applications. The algorithms are general enough to be applied to other applications that use data hoarding techniques to access collections of data, including filesystems such as CODA and FICUS. Our primary contribution is a set of theoretical results that illustrate the potentiality to improve the application performance.

The rest of the paper is organized as follows. Section 2 defines the assignment problem formally and transforms it into a variation of 0/1 Knapsack problem. The Adaptive Algorithm and Non-Adaptive Algorithm for partitioning an object graph are described and analyzed in Section 3. These two algorithms can be realized in central mode. A distributed algorithm, which can be realized in every movable object itself is proposed and analyzed in Section 4. Section 5 concludes the paper and raises some open questions for future works.

2. The Formal Description of the Assignment Problem

Both application-transparent migration and application-aware migration require the identification of an optimal partitioning of object graph, especially in the latter case, in which we should decide which other objects will accompany the object chosen by the application to move to the remote side for high overall performance. We consider these problems in two ways. One is from a centralized view, in which a centralized controler is needed to schedule each object. The other is from a decentralized view. We let an object

decide by itself whether to move or not through coordinating with other objects. Our algorithms proposed in this paper can proximate this optimization problem in these two cases.

2.1 Model and Assumption

The mobile system in our scenario, denoted by (M, L, P) consist of two geographically distributed devices M and P which are connected by a wireless link L. M represents a mobile device characterized by its processor speed and bus bandwidth, denoted by M(φ_m, ω_m). The same with proxy device P, denoted by P(φ_p, ω_p). We use bandwidth, error rate and latency as parameters to characterize the current communication link L(β_t, ε_t, ρ_t) between client and proxy. If the wireless network is extremely poor, there will be no partition to be generated for performance consideration. With the improvement of network performance, the size of movable partition should be increased as well. We define the load of the wireless network as θ_t [10]

$$\theta_t = \frac{\beta}{3\beta_t} + \frac{\varepsilon_t}{3\varepsilon} + \frac{\rho_t}{3\rho} \qquad (1)$$

where β_t = bandwidth at time t

ε_t : error rate at time t

ρ_t : latency at time t

β, ε and ρ are nominal values describing the worst acceptable performance. When $\beta_t=\beta$, $\varepsilon_t=\varepsilon$ and $\rho_t=\rho$, we have $\theta_t=1$, which is the worst load that can be tolerated in the wireless

network. If $\theta_t \leq 1$, the wireless network is in a good condition. The nominal bandwidth β, nominal error rate ε and nominal latency ρ are set based on the estimated maximum tolerance. They may be adjusted by application developers for different type of networks or applications. θ_t reflects the capacity of L.

We further define processor ratio

$$\lambda = \frac{\varphi_m}{\varphi_p}$$

as a measure of the relative power of the processors. We always assume that $\lambda < 1$, which means that the proxy processor is more powerful than the mobile processor. We give the definition of bandwidth ratio as

$$\mu = \frac{\omega_m}{\beta_t} \qquad \eta = \frac{\omega_m}{\omega_p}$$

We suppose here $\mu > 1$ and $\eta < 1$. This definition and assumption is rational in that when an inter-object communication happens in a single machine, this communication, the messages or results must resort to the local bus, whereas if it happens between the two machines, it must goes through the wireless link. Hence, the bandwidth of the local bus and wireless link is critical to evaluate the efficiency of floating objects between processors. However, if the bandwidth of mobile host is much higher than that of wireless link, and much lower than that of proxy host, we can assume $\mu >> 1$ and $\eta << 1$.

A program is also abstract into object graph, in which the nodes represent the objects with their information such as the size and the computational load, the weighted edges indicate the cost of inter-object references when the objects are assigned to the mobile

host. Obviously, this cost will be μ times when one of the objects is moved to the proxy host. The cost function used to compute the edge weights may vary depending on the nature of the system. Initially, we choose to minimize the absolute running time of a program.

Our algorithms are set up on this object graph. However, its dynamic construction is beyond the scope of this appendix. It may resort to the support from Java virtual machine if the application is a Java program, or from the runtime system of some language. Here, we assume the dynamic object graph is always available for our analysis and design of the algorithms.

2.2 Problem Definition and Transformation

Formally, for a given object graph $G=(N,E)$, our goal is try to find a sub-graph $G' =(N', E')$, $N' \subseteq N$, $E' \subseteq E$ that maximizes the collective cost function $f(G')$ which is defined later. G' is constructed dynamically with the increment of the number of movable objects from mobile device to its proxy server. Each node v in G' has a computational load $C(v)$, size parameter $S(v)$, and inter-object communication cost w with its neighborhood. Its edges can be divided into two sets: $S_1(v)=\{(v,u) \mid u \notin G'(N')\}$, $S_2(v) = \{(v, u) \mid u \in G'(N')\}$, $|S1|+|S2|=deg(v)$. Hence, accordingly, the relation cost $w(v)=\{cost(v,u) \mid u \in N(v)\}$ has two categories $w_i(v) =\{cost(v,i) \mid i \in S_1(v)\}$ and $w_j(v) =\{cost(v,j) \mid j \in S_2(v)\}$ (see Figure 1)

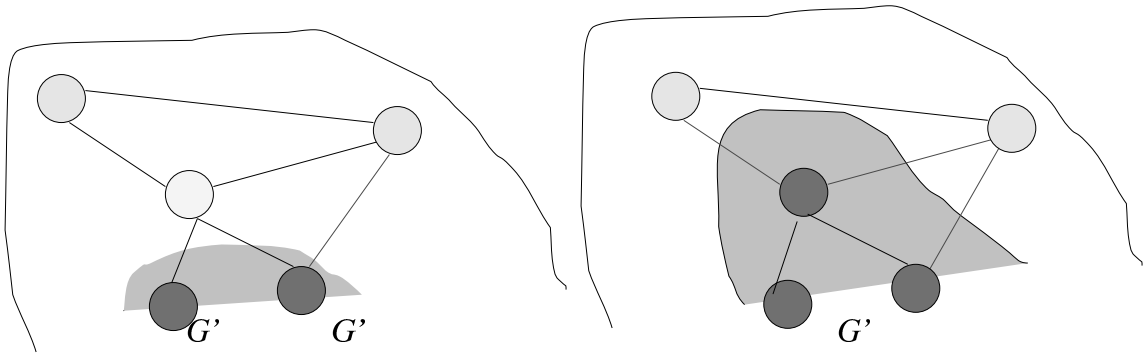


Figure 1: a partition of $G=(N,E)$ a) Before node v is moved b) After node v is moved

So the total cost of node v before migration is defined as (Figure 1a)

$$Cost_1 = C(v) + \sum_{i=1}^{|s1|} w_i(v) + \mu \sum_{j=1}^{|s2|} w_j(v) \qquad (2)$$

After node v has been moved to a remote proxy server, this value should be changed (Figure 1b)

$$Cost_2 = \lambda C(v) + \mu \sum_{i=1}^{|s1|} w_i(v) + \eta \sum_{j=1}^{|s2|} w_j(v) \qquad (3)$$

and at mean time $G'(N') = G'(N') \cup \{v\}$. Like this manner, we pick up node one by one from $G=(N,E)$ to construct a subgraph G' to satisfy:

$$f(G') = MAX \sum_{v \in G'} \{Cost_1(v) - Cost_2(v)\}$$

$$= MAX \sum_{v \in G'} \{(1-\lambda)C(v) + (1-\mu)\sum_{i=1}^{|s1|} w_i(v) + (\mu-\eta)\sum_{j=1}^{|s2|} w_j(v)\} \qquad (4)$$

We denote $(1-\lambda)C(v)+(1-\mu)\sum_{i=1}^{|s1|} w_i(v)+(\mu-\eta)\sum_{j=1}^{|s2|} w_j(v)$ as *B(v)*. However if we assume $\mu >> 1$ and $\mu >> \eta$, then we can simplify *B(v)* as

$$B(v) \approx (1-\lambda)C(v) - \mu\left(\sum_{i=1}^{|s1|} w_i(v) - \sum_{j=1}^{|s2|} w_j(v)\right) \quad (5)$$

If we denote $W1 = \sum_{i=1}^{|s1|} w_i(v)$ and $W2 = \sum_{j=1}^{|s2|} wj(v)$ then

$$B(v) \approx (1-\lambda)C(v) - \mu(W1 - W2) \quad (6)$$

Finally, we have the goal function:

$$f(G') = \max imize \sum C(v)B(v) \quad (7)$$

subject to the constraints: $\sum C(v)S(v) \le C(\theta_t)$

$C(v)=1$ if $v \in G'(N')$ otherwise $C(v)=0$;

from this transformation, one can see that the assignment problem is equivalent to a typical 0/1 Knapsack problem. What we should notice here is that in the traditional Knapsack problem, the objects are independent; picking up one does not affect the choice of the others. This requirement which is reflected in our scenario is that *B(v)* is a constant for any object v. However, *B(v)* in the assignment problem is not a constant again. It depends on the subgraph *G'*, i.e., the value of *W1-W2*, which means moving an object will have impact on the mobility of other objects. This complicates the situations. Fortunately, we can enumerate the nodes in the graph to calculate the impact of the migration of the initial object picked up by the user. This impact on each node can

decide the node to move or not if we define some criteria to evaluate this impact. Hence, the objects, which should accompany the initial object to move out, are definitely decided. This discussion gives us some hints to design our algorithms.

3. Algorithms

In this section, we propose two approximate algorithms to solve this optimization problem from a centralized view. We first give some basic definitions and then describe the algorithms.

3.1 Basic Definition

3.1.1 Node Data Structure

When an object graph is provided, we hope to move processor-bound objects with small size to the proxy side and not to move the object, which has high coupling with its fixed local neighbors. Naturally, if some of its neighbors are moved, the possibility of moving this object should also be changed. Every time, when we decide to move an object $\boldsymbol{v}$ we only take its direct neighbors into account. The factors we consider include the relationship with its neighbors and the computational load as well as its size. Hence, the object graph in our situation is a weighted graph with node weight representing the computational load, and edge weight indicating the cost of inter-object communication. We try to construct the subgraph $G' \subseteq G$ iteratively initiated by G' containing only one node, i.e., the initiator picked up by the application or assigned by the algorithms. We classify the edges of a node v into two classes $S1(v)$ and $S2(v)$ as we described in the previous section, and then, we design the structure of a node as the follows

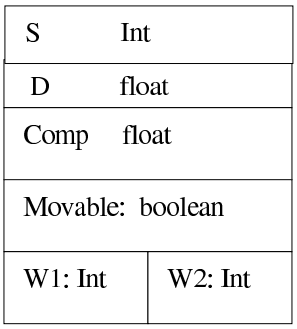


Figure 2: **Structure of a Node in Object Graph**

S : Object Size

Movable: a Boolean value assigned by partition algorithm to indicate if this node is marked move or not.

Comp: Computational Load

W1: definition is in the previous section

W2: definition is in the previous section

D: decision value typed float, we calculate *D* by using the following formula:

$$D(v) = \frac{B(v)}{S(v)} \qquad (8)$$

Whether a node **v** is movable or not depends on the value *D(v)*. But *D(v)* will depend on each other, which means that when an object is moved, it will have impact on its neighbor's *D(v)*. Due to this dynamic property, we can not resort to the traditional algorithm based on the nonincreasing sequence of the *Di(v)* , *i=1,…,n*. Our approach is to

provide some threshold value τ to control the objects selected and not to overflow the link capacity.

3.1.2 Threshold Value

Prior to the moving procedure, we can calculate *D(v)* for $\forall v \in G'$ based on $B(v) = C(v) + (1-\mu)W1$. Then we have $\overline{D} = \sum_{v \in G'} \frac{D(v)}{|G'(N')|}$. The initial threshold value τ is defined as the average of *D(v)* for $\forall v \in G'$.

$$\tau = \overline{D} \tag{9}$$

If $D(v) \geq \tau$ then *v* is moved. τ can be non-adaptive or adaptive. Non-adaptive τ means the value is not changed during the computation. Adaptive τ means the value is changed based on its previous value and other known information during the computation. However, static threshold can not reflect what have been done, but hopefully, it has high efficiency. In our algorithms, we first consider the adaptive method, adjusting τ to make the algorithms adapt to the variant of picking node sequence. Our adaptive formula is:

$$\tau = \frac{|G(N')|\tau - D(v)}{|G(N') - v|} \tag{10}$$

The basic idea behind this adaptive formula is that at each time, when a node has been decided to move or not, its weight will deduct from $\sum_{v \in G'} D(v)$, and the average of the rest is recalculated as a new τ. This adaptive threshold can not guarantee to select the optimal candidates, but can discard those which are the worst. Second, we consider the non-adaptive method with fixed threshold value as (9).

3.2 Adaptive Algorithm(AA)

3.2.1 Informal Description

Now, we describe the first algorithm how to select the nodes for migrating. First, we calculate *D(v)* for every node, here *S2(v)=ϕ, W2=0,* and then we calculate a threshold value τ. Second, we give the initiator v_1 or pick the node v_1 with biggest *D(v1)* which indicates that this node is the most advantageous to be moved. We compare it with the threshold, if *D(v1)* is below the threshold, the whole object graph should reside on the mobile client, no partition is generated, otherwise we mark v_1 movable. If the link capacity is big enough, the whole object graph should be moved to the proxy sides. After we have processed v_1, we start the diffusion procedure by processing all its direct neighbors. We intend to mark some neighbors and make them move with v_1 to the remote side and hence reduce the cost of remote invocations. This processing includes re-calculating *D(v),* and the threshold again, comparing the *D(v)* with the new threshold and marking it movable or not since at least one of its neighbor's state is changed. When we have processed all the neighbors of the initial node, we pick them one by one to investigate theirs direct neighbors again, and the diffusion procedure is repeated until we mark sufficient nodes whose total size exceeds the link capacity and no more nodes can be moved.

3.2.2 Formal Description

We give a formal description of the algorithm in this subsection. Component complexities are on the right side, some explanations are also provided following the algorithm.

Algorithm: (λ, μ, η θt are system parameters known by algorithm)

```
1.  Input: OG,  O, C (Object Graph, Object to be moved, Link Capacity)
2.  Output mSet; /* movable set */
3.  nQueue=new nQueue();
4.  mSet={}  /* movable set */
5.  dSet={}   /* dead set, non-movable */
6.  compute D(v) , τ as well as W1&W2 for each node.                    O(E)
7.  if(O is NULL) {
8.          get the node v with maximal D(v), i.e D(v)=max{D(vi)} i=1,…,n
9.  } else {v=O}
10. nQueue.addTail({v})
11. while(τ>0) {
12.    u = nQueue.deleteHead();
13.    if(D(u)≥τ and C-S(u)≥0) {
14.       for (each w∈ N(u)) {                                          O(deg(u))
15.          if (w∉ mSet) {
16.             W1(w)=W1(w)-w(u,w);
17.             W2(w)=W2(w)+w(u,w);
18.          }
19.       }
20.       mSet=mSet+u;
21.       /* mark u in mSet;*/
22.       move u to remote site.
23.       C=C-S(u);
24.       /* compute the threshold again */
25.       τ = (n*τ-D(u))/(n-1);
26.       n=n-1;
27.    } else {
28.       if (C-S(u) ≥0) {
29.          nQueue.addTail(u);
30.       } else {
31.          dSet = dSet+u;
32.          τ = (n*τ-D(u))/(n-1);
33.          n=n-1;
34.       }
35.       if (nQueue.size()==1) break;
36.    }
37.    nQueue.addTail(N(u));                                            O(deg(u))
38. }

39. Class nQueue {
40.   Queue Que;
41.    Public nQueue() {
42.    Que = empty
43.  }
44.   void addTail(Set N(v)) {
```

```
45.      for(each u∈ N(v)) {
46.         if( u ∉ mSet∪nQueue∪dSet)
47.              Que.add(u);
48.      }
49.  }
50.  Node deleteHead() {
51.       return Que.del();
52.  }
53. }
```

3.2.3 Explanation and Example

According to AA, the nodes will make contribution to *v* only if they are neighbors of *v* and they are movable. The following picture (Figure 3) demonstrate this case:

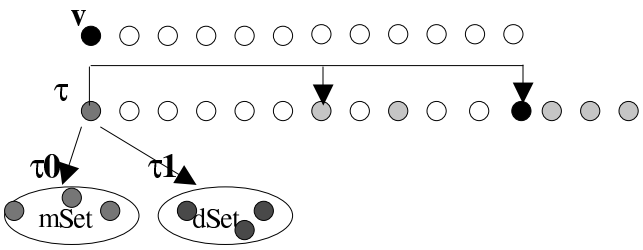


Figure 3: the Queue in AA

v is any node. Only movable nodes are green. Yellow nodes are neighbors of *v* and arrows indicate the contributions made by green nodes. Each time, the node at the head of the queue will be deleted and checked. The threshold value τ is initiated by the average of *D(u), u ∈ G(N).* This decision only occurs at the head of queue. If *v* is not movable, then there are two cases: 1) the size of *v* exceeds the current remaining link capacity, *v* will be deleted from *nQueue* to put into dead set (*dSet*) and will be never checked it again. In this situation τ will be recalculated. 2) $D(v) < \tau$, *v* will move to the end of queue and add its neighbors which are not already in the *nQueue* or *mSet* or *dSet* at the tail of the queue. τ is not adjusted in this situation. Nodes in front of v in *nQueue* can continually make contribution to the neighbors of *v* or *v* itself, and these neighbors naturally make

contributions to *v*. Otherwise, *v* will be inserted into the movable node set (*mSet*) and its direct neighbors will be added to the queue again. The procedure of processing nodes (which are some node's direct neighbors) and adding their direct neighbors in the queue is called *Diffusion Procedure*. In principle, every movable node makes an effort to contribute to the migration possibilities of its non-movable neighbors, as soon as it gets sufficient contribution it will become movable and make contribution to its neighbors again until all nodes have been investigated and no more nodes can be moved. The algorithm will terminate when the current link capacity is used up or no more nodes to be moved. Finally, there are two set, one is *dSet* which contains non-movable nodes, the other is *mSet*, which contains movable nodes.

Example: we have following object graph

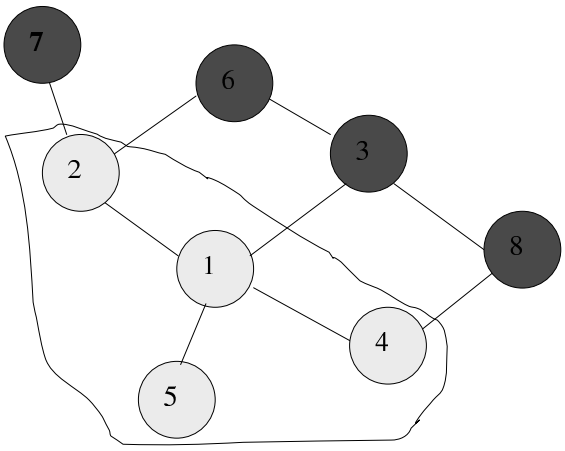


Figure 4: Example Object Graph

In this example, for simplicity we assume $\lambda \approx 0.01$, $\mu = 10^6$, $\eta = 0.01$, then $B(v) = (1-\lambda)C(v) + (1-\mu)W1 + (\mu-\eta)W2 \approx C(v) - 10^6(W1-W2)$.

Table 1: Node Information

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Load	169	295	35	144	3	47	231	93
Size	2	8	4	5	1	2	10	4
D	69	35	5	26	-7	17	23	21

Table 2: Reference Cost: 10^{-6}

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Node 1	0	8	4	9	10			
Node 2		0				6	1	
Node 3			0			7		4
Node 4				0				5
Node 5					0			
Node 6						0		
Node 7							0	
Node 8								0

Threshold Value

$\theta_t = \beta/3\beta_t + \varepsilon_t/3\varepsilon + \rho_t/3\rho = 1/3(45\% + 150\% + 150\%) = 1.15$

$C(\theta_t) = 16$

$\tau = D \approx 23.6$

	nQueue	**τ**	**mSet**	**C**
1.	{1[69]}	23.6	{ }	16
2.	{2[37],3[7],4[29.8],5[13]}	17.1	{1}	14
3.	{3[7],4[29.8],5[13],7[23.2],6[23]}	14.1	{1,2}	6
4.	{4,5,7,6,3,8[21]}	14.1	{1,2}	6
5.	{5,7,6,3,8[23.5]}	11.8	{1,2,4}	1
6.	{7,6,3,8}	16.5	{1,2,4,5}	0

dSet = {7,6,3,8}

Due to movement of node 1, we have two partitions: non-movable partition {7,6,3,8} and a movable partition{1,2,4,5}. Total object size is 16.

According to (2) and (3), we can calculate the benefit of this partition:

Benefit = B(v1)+B(v2)+B(v4)+B(v5) =138+296+148+13=605

3.2.4 Analysis of AA

In this section, we will give some properties of this algorithm and analyze its complexity.

Theorem 3.1: $\forall v \in G(N)$, we have $D(v) \leq D'(v)$ ($D'(v)$ is the reevaluation of $D(v)$ after the diffusion procedure initiated by any movable node.)

Proof: Suppose the node u initiates the diffusion procedure. The D value of its direct neighbors will increase, the other nodes will not change. Hence we have $D'(v) \geq D(v)$, $\forall v \in G(N)$.

Theorem 3.2: *mSet* construct a subgraph of G. If the algorithm is deterministic and link capacity $C2>C1$, then $mSet_{c1} \subseteq mSet_{c2}$

Proof: Since we only deal with those nodes based on neighbor relationship, they must be connected. So *mSet* constructs a subgraph of *G*. On the other hand, if we have deterministic method to decide a node's neighbors in step 14 and 37, then the algorithm is deterministic. If the algorithm is deterministic, step 20 determines that *mSet* is monotonic which means that if link capacity $C2>C1$, then $mSet_{c1} \subseteq mSet_{c2}$.

Theorem 3.3: AA is convergent

Proof: We will show that when the algorithm terminates, $nQueue = \Phi$. Suppose there exists $nQueue=\{v1, v2,\ldots,vk\}$ when AA terminate, we can conclude that $\exists \tau k$, $D(v)<\tau k$

and $C\text{-}S(v) \geq 0 \ \forall v \in nQueue$. But in fact, $\tau k = \frac{1}{k}\sum_{v \in nQueue} D(v)$, due to Theorem 3.1 there must be at least one element $u \in nQueue$ satisfying $D(u) \geq \tau k$, contradictory to our assumption. So when the algorithm terminates, $nQueue = \Phi$. Specially, if $C \geq \sum_{v \in G(N)} S(v)$, then AA will terminate with the result that all nodes are scheduled to the remote side.

Theorem 3.4: the complexity of AA is $O(NE)$

Proof: In the worst case $\sum_{u \in G} S(u) \leq C$, there are at most $n = |G(N)|$ nodes in the queue and at least one node whose $D \geq \tau$ will be deleted from the queue after n loops. Each loop processes one node with complexity of $O(deg(node))$. So deletion of one node needs $\sum deg = O(E)$ time. The next node will be deleted within next $n\text{-}1$ loops due to the reevaluation of threshold τ and Theorem 3.1. This process will continue. Total time is $O(NE)$. When the object graph is a complete graph, the complexity is $O(n^3)$.

3.3 Non-adaptive Algorithm(NA)

3.3.1 Description of the Algorithm

The Adaptive Algorithm can be seen as using breadth-first searching techniques since we first process a node's neighbors and then these neighbors' neighbors. This algorithm is efficient in our adaptive procedure since each node needs to be investigated many times with different threshold value as long as the link capacity is sufficient. However, it consumes a lot of time in this procedure, which may be not tolerable in some real condition. In this subsection, we propose a non-adaptive algorithm based on depth-first searching which is N times faster than AA, here N is number of objects in the object

graph. First, we give the formal description of the algorithm, and then give an explanation and analysis.

Algorithm:

```
1.  begin (λ, μ, η are system parameters known by algorithm)
2.      Input OG, O, C and τ (global variables)
3.     Output mSet;
4.      mSet={}
5.      compute D(v), τ as well as W1&W2 for each node.                O(E)
6.      if (O is NULL) {
7.            get the node v with maximal D(v), i.e D(v)=max{D(vi)} i=1,…,n
8.      }else {v = O)
9.      if(D(v)≥τ and C-S(v)≥0){
10.       mSet = mSet+v;
11.       move v to remote site.
12.       C = C-S(v);
13.       f(v);
14.     } else {
15.       no nodes to be moved
16.   }
17. end;

18.  f(Node u) {
19.      for (each w∈ N(u) ) {                                  O(deg(u))
20.        N(u) = N(u)-w;
21.        if (w ∉ mSet) {
22.          W1(w)=W1(w)-w(u,w);
23.          W2(w)=W2(w)+w(u,w);
24.          B(w)=(1-λ)C(w)+ (1-μ)W1(w)+(μ-η)W2(w)    /* compute D(w);*/
25.          D(w) = B(w)/S(w)
26.          if(D(w)≥τ and C-S(w) ≥0) {
27.              mSet = mSet+w;
28.              move w to remote site.
29.              C = C-S(w);
30.              f(w);
31.          }  else {
32.              cSet = cSet+w;
33.          }
34.       }/* if  */
35.     }/* for */
36.  }
```

3.3.2 Explanation and Example

Explanation: the form of this algorithm looks simpler than AA. But due to its recursive nature, it is hard to read. Our idea is that from any node *v*, which can be movable, we try to figure out which other nodes should accompany this node *v* to go to a remote side. We first check its neighbors one by one to see if any of them, say *u* can be marked *movable*. If successful, Node *u* will be marked *movable* and put *u* in *mSet* which is a global variable, and from this new marked Node *u*, we do the same thing recursively. Otherwise, we mark this node *checked* and put *u* in *cSet* which means that "I (node *v*) have made contribution to Node *u*, but this contribution is not enough to make it mark *movable*", here contribution means the recalculation of W1 and W2. After all nodes in the neighborhood of Node *v* are processed, Node *v* should go back to its parent node, say *w* which means exiting the current function and entering the previous one. Here the trick is that when you process the neighbor nodes, the nodes in $mSet \cup cSet$ cannot be checked because the nodes in *mSet* are movable, it is not necessary to check them again. The problem is the nodes in *cSet,* which are marked *checked.* From the point of Node *v* when tracing back from Node u, there are three events for a node say $p \in cSet(v) \cup mSet$.

1. $p \in cSet - mSet \Rightarrow p$ is not movable, and Node *v* can not make it movable again since Node *v* has made contribution to it, if it can be movable, it should receive more contribution from other nodes and made movable by someone else.
2. $p \in cSet \cap mSet \Rightarrow p$ is movable. Node *v* cannot make *p* movable but someone else in the later will make *p* movable, and from *p* the recursive function executes. Hence, for Node *v*, it is not necessary to check *p* again.

3. p $\in$*mSet-cSet*$\Rightarrow$*p* is movable, but not marked by node *v*. In this situation, just skip it.

So the nodes we need to check is *N(v)- cSet(v)*$\cup$*mSet*

Example: we use the Table 2 in 4.2.3 to represent the relationships among the nodes

Table 3: Node Information

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Load	169	295	35	144	3	47	231	93
Size	2	8	4	5	1	2	10	4
D	69	35	5	26	-7	17	23	21

Table 4: Procedure of NA

n	**N(v)**	**cSet(v)**	**mSet**
1	2,3,4,5		1,2
2	1,7,6	7	1,2,6
6	2,3	3	1,2,6
1	2,3,4,5		1,2,6,3
3	6,1,8		1,2,6,3,8
8	3,4		1,2,6,3,8,4
4	1,8		1,2,6,3,8,4
1	2,3,4,5	5	1,2,6,3,8,4

Capacity C=37, τ=27.

Ouput *mSet* = {1,2,6,3,8,4}

3.3.3 Analysis of NA

Theorem 3.5: NA is loop-free

Proof: this property is obvious, since each movable node will not be checked again. Loops will be broken by step 21 in NA. For example:

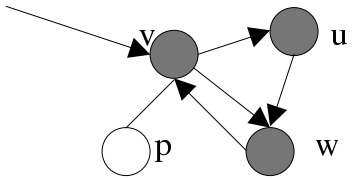


Figure 5: Loop free

Node *v* is movable, then it checks its neighbor *u, u* is movable then the control goes to its neighbor *w, w* is movable too. It will check its neighbor node *v*. But node *v* has been moved, control does not go to node *v* but return to *v* via *u*. Node *v* checks its neighbor *w* and finds it has been moved, processing continues with a new neighbor node *p*.

Theorem 3.6: The complexity of NA is *O(E)*

Proof: For $\forall v \in G(N)$, the processing of *v* is done as soon as $N(v)\text{-}cSet \cup mSet = \{\}$. Based on this procedure, the complexity of the algorithm is *O(E)* for a given node since for each recursive function for processing Node *v* the complexity is *O(deg(v))*. Totally there are *n* nodes and hence *n* recursive functions. i.e., $\Sigma deg = O(E)$.

Theorem 3.7: The lower bound of the Non-Adaptive Algorithms is $\Omega(E)$.

Proof: To decide which nodes should be move with the given initiator v requires every node in the graph to be checked. However, the mobility of a node depends on its relationship with its direct neighbors. Hence, checking a node *u* needs at least *O(deg(u))* operations since checking one neighbor of node *u* is a constant effort. Totally, at least $\Omega(\sum deg(u)) = \Omega(E)$ operations are needed.

4. Distributed Algorithm

When we apply the algorithms described in the previous section to our toolkit, it needs a central control logic, which has to suffer the risk of shutdown and loss of performance. In

this subsection, we design a distributed protocol based on the principle of our algorithms described to attack the assignment problem. Our idea is that we embed some control logic in Proxy Objects, which can communicate with each other by message passing, and decide locally whether its associated object moves or not based on these messages. We should point out that our techniques described here are for general purposes beyond the scope of mobile computing, and maybe not suitable for mobile device with constrained resource. In SMP, these proxy objects can be separate processes or even separate threads in a single process. In a large-scale distributed memory system, these proxy objects together with their real objects can reside in physically distributed sites. When an object in some site needs to move to other sites, some other objects related to this object in different sites should also move in order to improve the performance. In this situation, a distributed protocol for object migration is desired.

4.1 Basic Idea

Each object decides whether to move or not through communicating with its direct neighbors. The decision is based on the relationship between D and the current threshold τ as well as the current link capacity C. τ and C are dynamically adjusted based on the current $D(v)$ and object size. Because these two variables are inherently global, their changes have to be atomic. Hence, our algorithm is set up around a distributed mutual exclusion algorithm. We exploit Raymond's algorithm [11] which is a token-based scheme established on a tree structure for distributed mutual exclusion to design our algorithm. The algorithm is divided in several stages:

4.1.1 Token Generation

Obviously, there is only one token in the given object graph. The token generation can be realized by leader election. The leader will hold the token. Due to the fact that the leader election is equivalent to the spanning tree construction in an arbitrary graph, we can obtain a spanning tree rooted at the leader. A well-known algorithm for election in arbitrary network is Kingdom algorithm [12] proposed by Gallager, Humblet and Spira in 1984. In addition to this, there exist other algorithms [13, 14, 15, 16, 17] serving the same purpose. The advantage of the former algorithm is that when a leader has been elected, a spanning tree rooted at the leader is also constructed. In our algorithm, we do not pay more attention on the token generation with the assumption that the token has been generated and resides at the leader which is the root of a spanning tree of the given arbitrary object graph.

4.1.2 Token Circulation

Some objects are candidates which require to move to the remote site. This migration consumes the link capacity and adjusts the threshold value for others. Hence, these candidates will require the token for exclusive execution. Only one candidate can receive the token. The token circulates around the tree. We use the Raymond Algorithm for this purpose. As soon as a candidate node receives a token, it will decide to be movable, dead or keep its candidate status. When it decides to move, it will update the token with new τ and C and broadcast this new information to its neighbors. These neighbors may change state based on this message. If some neighbors become candidates, they will compete for the token following Raymond Algorithm. The candidates are not guaranteed to be moved after they obtain the token, they may be moved or dead, and may be checked many times

with different threshold and capacity value. If a candidate node receives a token and keeps its status, it will re-compete for the token again. The token will contain special information and be changed during the computing.

4.1.3 Termination Detection

As soon as a node computes $\tau \leq 0$, it will broadcast termination message on the spanning tree. If all nodes except the moved nodes receive this message, they will mark themselves dead nodes.

4.2 Model and Assumptions

Our distributed algorithm is symmetric since every proxy object runs the same algorithm concurrently. We design the algorithm under the assumption:

1) Complete network
2) Bidirectional links
3) No faults
4) Distinct IDs associated with the entities

4.3 The Algorithm

To implement the distributed algorithm, each node X must hold certain information.

4.3.1 Information Held by Each Node

1) All edges with weight incident with it, i.e., neighbors
2) Threshold value τ, and system parameters λ, μ, η
3) Data contained in the data structure of the node.
4) Destination of moving object

5) Associated object

6) Information needed by Raymond Algorithm

4.3.2 Formal Description

In this subsection, we will describe the distributed algorithm for object migration based on an arbitrary object graph. For simplicity, we do not provide details how to construct the spanning tree. On the contrary, we assume that each node has the local orientation and knows which neighbors are in the tree. The token is generated in the leader of the tree. Raymond Algorithm is employed for circulating this token for mutual exclusion since the adaptive threshold value and link capacity is only modified at one node. We assume that the nodes hold the information needed by Raymond Algorithm whose details are omitted in our algorithm. Readers can refer to his paper [16].

Every node in the graph will be in one of four states: *Movable*, *Dead, Candidate* and *Idle*. There are two kinds of messages in the algorithm: the first are Raymond messages which include the token message *TOKEN = (C, τ, n, v)* and the *REQUEST* message. The second are migration messages which include the termination message TERMINATION, the wakeup message *WAKEUP* and broadcast messages *BROADCAST= (C, τ, v)*. The definition of *C, τ* is the same as in Section 3.1.2. Initially every node is in Idle state except several initiators which are in Candidate state. Candidate nodes will compete for the token. Other kinds of nodes do not compete for the token. Our algorithm includes several stages:

Stage1: Node Initialization (for each node x.)
1) Construct spanning tree
2) Generat the TOKEN=(C, τ, n, leaderID) (in special leader node.)
3) Calculate its Locallist = N(x).
4) Termination detection is guaranteed in this stage.

Stage2: Node Migration (for any node x)
Idle
(1) receiving(TOKEN) from node j
do Raymond algorithm after receiving TOKEN

(2) receiving BROADCAST=(C, τ, j) from node j
if(j ∈ Locallist(x))
update W1/W2 by using w(x,j)
Locallist(x) = Locallist(x)-j

become(**Candidate**)

(3) receiving (TERMINATION) on the spanning tree
become(**Dead)**

(4) receiving (REQUEST) from neighbor j
do Raymond algorithm

(5) receiving (WAKEUP) from neighbor j
become(**Candidate**)

Candidate
broadcast WAKEUP to N(x) at first entry.
do Raymond algorithm for obtaining TOKEN

(1) receiving BROADCAST=(C, τ, j) from node j
if(j ∈ Locallist(x))
update W1/W2 by using w(x,j)
Locallist(x) = Locallist(x)-j

(2) receiving (TOKEN) from node i or itself
do Raymond algorithm after receiving the TOKEN

resolve the TOKEN= (C, τ, n, v)
if(S(x) ≤ C)
if(D(x) ≥τ)
C=C-S(x)
τ = (n*τ-D(u))/(n-1);
n=n-1;
construct a new TOKEN=(C, τ, n, v)
if(τ≤0)
broadcast TERMINATION on the spanning tree.
else
broadcast BORADCAST=(C, τ, v) to N(x)
become(**Movable**)
else

became(**Candidate**)
else
$\tau = (n*\tau - D(u))/(n-1)$;
n=n-1;
if($\tau \leq 0$)
broadcast TERMINATION on the spanning tree
else
become(**Dead**)

(3) receiving an associate object from node j
reset W1/W2;
compute D
become(**Idle**)

(4) receiving (REQUEST) from neighbor j
do Raymond algorithm

(5) receiving(TERMINATION) from node j
termination processing

Movable:
move associated object to destination;
do Raymond algorithm to release the TOKEN

receiving(TOKEN) from node j
do Raymond algorithm after receiving TOKEN

receiving(BOARDCAST) from node j
discard it

receiving(TERMINATION) from node j
termination processing

receiving (REQUEST) from neighbor j
do Raymond algorithm
receiving(WAKEUP) from node j
discard it

Dead
do Raymond algorithm to release the TOKEN
broadcast WAKEUP to N(x)

receiving(TOKEN) from node j
do Raymond algorithm after receiving TOKEN.

receiving(BOARDCAST) from node j

discard it

receiving(TERMINATION) from node j
termination processing

receiving (REQUEST) from neighbor j
do Raymond algorithm

receiving(WAKEUP) from node j
discard it

Example: the following example demonstrates our algorithm(Figure 6)

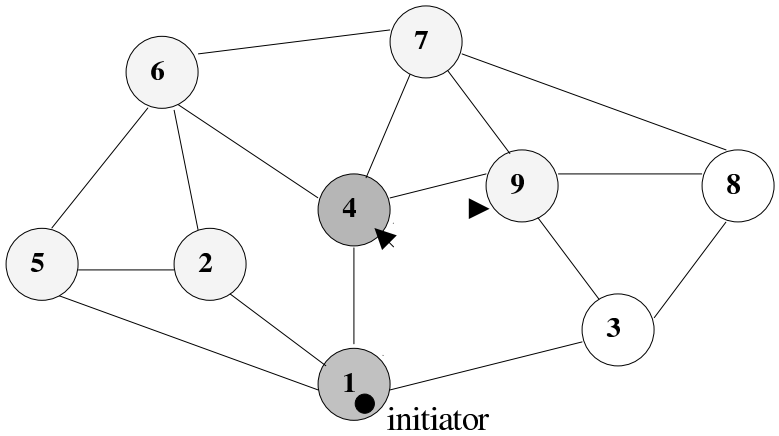


Figure 6: Distributed Example

Suppose, the spanning tree is constructed pictured by the bold lines, (1) is the leader with initial token. (4) is the initial candidate requiring the token. When it receives the token, it decide to move and broadcast to its neighbors (1) (6) (7) (9) . Next (9) receives the token and decide to move. Following (6) gets the token and is dead.

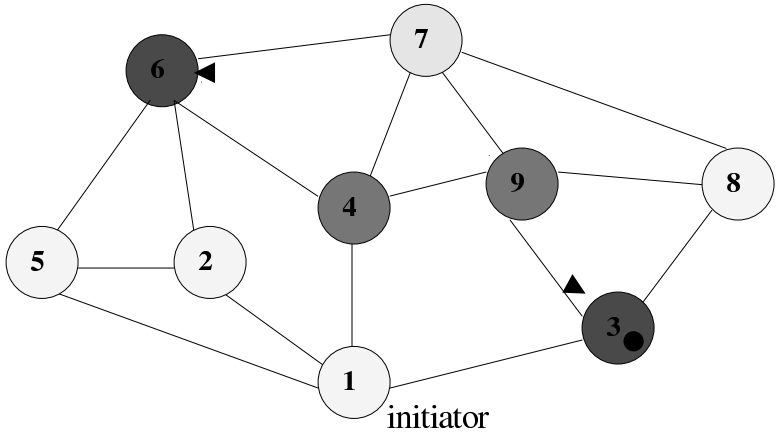


Figure 7: Distributed Example

4.4 Algorithm Analysis

In stage 1, constructing a spanning tree in an arbitrary graph reqire *O(NlogN+E)* messages. In stage 2, the average message complexity per critical section of Raymond algorithm is *O(logN)* under light demand and reduced to approximately four under saturated demand. In our algorithm, there are N^2 critical sections because each node will be check at most N times. So the total message complexity for critical section is $O(N^2 logN)$. Next we will consider the number of broadcast and wakeup message. The broadcast message occurs only at the node changing from candidate state to movable state. In the worst case, N nodes are movable, *O(E)* is the message complexity. The wakeup message occurs at the first time a node enters candidate state, and has *O(E)* complexity. Finally, the termination message is only on the spanning tree. Hence its complexity is *O(N)*. Totally, the message complexity of our algorithm is $O(NlogN+E)+O(N^2 logN)+O(2E)+O(N)=O(N^2 logN+E)=O(N^2 logN)$.

Theorem 4.1: The Algorithm is convergent.

Proof: Since there exist candidates at the beginning, every idle node will wakeup and compete for the token. Due to the property of Raymond algorithm, no starvation occurs.

Every candidate node can finally get the token. Based on the same principle as AA, each node will be either movable or dead. The algorithm is convergent, and the termination is guaranteed.

5. Conclusion and Future Work

How to schedule objects from resource-constrained mobile device to its proxy server for high performance is an interesting problem. This problem is effected by the current mobile environment as well as the application itself. We abstract this problem into a theoretical optimization problem based on the object graph. The factors we consider are parameters related to mobile systems as well as application features such as object workload, object size and object relationships. We hope to benefit from moving objects. But when we consider that the capacity of the wireless link is limited, we transform this optimization problem into a variation of 0/1 Knapsack problem, which is well-known NP hard. In the traditional Knapsack problem, the objects are independent. However in our situation, this restriction is relaxed which means that the objects have some relationship, picking up one will affect others. To resolve this problem, an adaptive algorithm with complexity $O(NE)$ is proposed, and its convergence is proved. A non-adaptive algorithm is also provided with complexity $O(E)$, i.e, N times faster than the adaptive counterpart. But the disadvantage of the non-adaptive method is its imprecision if the threshold value is not chosen well. Too big a threshold will waste some link capacity whereas too small a threshold will lose some valuable objects. Both algorithms are online in nature. We also provide a simple distributed algorithm based on Raymond Algorithm for solving this problem with message complexity $O(N^2 logN)$.

There are several problems unsolved in this research, which make up our future work. First the relationship between the link capacity and the load of the wireless network θt, i.e., $C(\theta t)$ need to be decided. We also need a simulation to evaluate the performance of these algorithms. We believe that the simulation may expose some statistical laws behind this problem. Exploiting these laws to guide the design of the adaptive threshold value is another challenge since the threshold value should guarantee both the convergence and optimization of the algorithm. To our knowledge, there is no existing work exactly satisfying our requirements, and hence our work can provide a baseline performance to evaluate the future research in this aspect. Although there are a lot of remaining open problems, we hope this work is useful in gaining some ideas on this problem.

Reference

[1] Thomas Kunz and Yang Wang, *A mobile code toolkit for resource-constrained portable devices*, Proceedings of the Symposium on Software Mobility and Adaptive Behaviour, pp 89-97, York, United Kingdom, March 2001.

[2] Allan Borodin and Ran EI-Yaniv, *Online Computation and Competitive Analysis,* Cambridge University Press 1998. ISBN 0-521-56392-5

[3] Kenneth A. Berman and Jerome L. Paul, *Fundamentals of Sequential and Parallel Algorithms* PWS Publishing Company 1997. ISBN 053494674-7

[4] Carsten Gerlhof, Alfons Kemper, Christoph Kilger, Guido Moerkotte, *Partition-Based Clustering in Object Bases: From Theory to Practic*es. Technical report 6.92, Fakultat fur Informatik, University of Karlsruhe, D-7500, Karlsruhe, Jun 1992

[5] Salim H. Omar, *A Mobile Code Toolkit for Adaptive Mobile Applications* Master's thesis Carleton University, 2000.

[6] Harold S. Stone, *Multiprocessor Scheduling with the Aid of Network Flow Algorithms.* IEEE Transaction on Software Engineering, vol. Se-3, No.1. pages: 85-93. Jan. 1977

[7] S.H. Fuller and D.P. Siewiorek, *Some observations on semi-conductor technology and the architecture of large digital modules*, Computer, Vol. 6, pages: 14-21, Oct. 1973

[8] G.S.Rao, H.S. Stone, and T.C. Hu, *Assignment of Tasks in a Distributed Processor System with Limited Memory*, IEEE Trans. On Software Engineering, vol. C-28, No. 4, pages: 291-299, Apr. 1979.

[9] S.H. Bokhari, *Dual Processor Scheduling with Dynamic Reassignment*, IEEE Trans. On Software Engineering, vol. SE-5, No. 4, pages: 341-349, Jul. 1979

[10] Grace Hai Yan Lo and Thomas Kunz, *A case study of dynamic application partitioning in mobile computing - an e-mail browser*, discussion paper, presented at the OOPSLA'96 Workshop on Object Replication and Mobile Computing (ORMC'96), San Jose, CA, 6 pages, October 1996.

[11] Raymond Kerry, *A tree-based algorithm for distributed mutual exclusion*, ACM Transactions on Computer Systems Vol.17, pages: 61-77, 1989

[12] R.G. Gallager, P.A. Humblet, P.M. Spira. *A distributed algorithm for minimum spanning tree,* ACM. Transactions on Programming Languages and Systems, Vol. 4, No.1, pages: 66-77, 1983

[13] K.E. Johansen, U.L., Jorgensen, S.H. Nielsen, S.E. Nielsen,S. Skyum, *A distributed spanning tree algorithm,* Proc.WDAG'87, pages: 1-12, 1987.

[14] E. Korach, S. Kutten, S. Moran, *A modular technique for the design of efficient distributed leader finding algorithm,* ACM TOPLAS 12, 1, 84-101, 1990.

[15] I. Lavallee, G Roucairol, *A fully distributed minimal spanning tree algorithm*, Information Processing Letters 23, pages: 55-62, 1986.

[16] B. Mans, N. Santoro, *On the impact of sense of direction in arbitrary networks*, Int. Conf. On Distributed Computing Systems (ICDCS'94), pages: 258-267, 1994.

[17] N. Santoro, *On the message complexity of distributed problems*, Int. J. Comp. Inf. Sci. 13, pages: 131-147, 1984.