Object Tracking In Distributed Systems

by

Yingchun Xu

B.Sc., South China Institute of Technology, China, 1984
M.Sc., Beijing University, China, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

IN

THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

University of British Columbia

April 1993

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date April 30, 1993

Abstract

Object mobility (or process migration) is a very important feature in modern distributed systems. By allowing objects to migrate from one machine to another preemptively, the system can provide several benefits. These include sharing load, reducing communication cost, increasing availability, and improving invocation performance.

One problem with object mobility is object tracking. Several schemes exist to handle the problem. A forwarding address scheme is used in DEMOS/MP and Emerald. To reduce the number of forwarding messages, DEMOS/MP uses an urgent updating policy to compress the message path. Whenever an object is moved, an updating message is sent to the last node immediately.

In Emerald, a lazy updating policy is adopted. When an object migrates, no updating message is sent. The new location is piggybacked on each reply message. This can also compress the message path for later invocations. The difference between the two is that DEMOS/MP places the cost on object migration while Emerald places the cost on object invocation. They both use the same updating policy for all objects.

We adopt a philosophy in which objects with different behaviors use different updating policies. Objects are divided into two groups: active objects and quiet objects. Active objects are defined as objects which move more often than they are invoked. Quiet objects are invoked more often than they are moved. To optimize object moving and invoking, active objects should use a lazy updating policy while quiet objects should use an urgent updating policy. We call this policy the adaptive updating policy.

The object tracking function is separated from Emerald and implemented as an independent protocol layer, OFP. A reliable datagram protocol, RDP has been designed to support OFP. Experiments are done to validate the protocols' performance. The protocols are implemented on the x-kernel platform.

Contents

List of Figures

List of Tables

Acknowledgement

I would like to extend my special thanks to my supervisor, Dr. Norm Hutchinson, for his advice, encouragement and guidance throughout the whole work. He patiently directed every step for this research.

I would like to thank Dr. Gerald Neufeld, for reading through this thesis and offering many valuable suggestions.

I would also like to thank Catherine Leung and Chris Healey for carefully reading this thesis and corrections.

Especially, I would like to thank my wife, Sophia, for her support and endless love.

Last but not the least, I would like to thank my daughter, Iris, who makes my life and work full of happiness.

Chapter 1

Introduction

As computer networks are developed and become more common, distributed systems become more and more realistic. One of the major goals of a distributed system is to support resource sharing.

Object mobility (or process migration) has become an important feature supported by several existing distributed systems. Such systems include Charlotte[Aea89], DEMOS/MP[PM83], Locus[PW85], Sprite[DO91], V[Che88], and Emerald[Hut87][RTL+91]. By allowing objects to migrate from one node to another preemptively, we obtain several advantages[Jul88].

1. **load sharing** – By moving objects around the system, one can take advantage of lightly used processors.

2. **Communications performance** – Active objects that interact intensively can be moved to the same node to reduce the communications cost for the duration of their interaction.

3. **Availability** – Objects can be moved to different nodes to provide better failure coverage.

4. **Reconfiguration** – Objects can be moved following either a failure or a recovery or prior to scheduled downtime.

5. **Utilizing special capabilities** – A movable object can take advantage of unique hardware or software capabilities on a particular node.

In addition, Emerald's fine-grained mobility provides the following three additional benefits:

1. **Data Movement** – Mobility provides a simple way for the programmer to move data from node to node without having to explicitly package data. No separate message-passing or file-transfer mechanism is required.

2. **Invocation Performance** – Mobility has the potential for improving the performance of remote invocation by moving parameter objects to the remote site for the duration of the invocation.

3. **Garbage Collection** – Mobility can help simplify distributed garbage collection by moving objects to sites where references to them exist.

One major problem that must be solved in any system with object migration is object tracking. When objects are allowed to move without limitation, this problem becomes even more important. To simply keep tracking the object's current location is expensive and unnecessary. Existing distributed systems deal with this problem using several different schemes.

Some systems depend on a third party such as an object's original node, home node, or a name server to get the object's new location when the old one becomes invalid. The single node that knows the location of an object causes unreliability and is a potential bottleneck. Some systems use special communication tools such as broadcast or multicast to send an object's new location whenever the object migrates. Broadcast or multicast is unreliable and can only be effective in a local area network. Finally, a forwarding address scheme is used in DEMOS/MP and Emerald.

Whenever an object moves, it leaves a forwarding address behind. This forwarding address can be used as a hint to the object's location. The forwarding address may not be the current address of the object since the object may have moved again. Simply using forwarding addresses to send messages incurs a serious overhead. Some method has to be used to shorten the forwarding address path. Different strategies are discussed in [Fow85].

In DEMOS/MP, an urgent update policy is used. Whenever an object migrates, an update message is sent to the last node on the path. The new forwarding address path goes from the last node to the correct node directly jumping over the node which sent the update message.

In Emerald, a lazy update policy is adopted. When an object migrates, no update message is sent. Only the object source and destination nodes update the object's location. When there is an invocation on an object, the object's new location is piggybacked on the reply message. Later invocations will now follow a shorter message transfer path.

The difference between DEMOS/MP and Emerald is that DEMOS/MP places the cost on the object moving stage while Emerald places the cost on the object invoking stage. They both use the same update policy on all the objects.

Some ideas can be derived from real life by comparing objects to people. Suppose there are two groups of people. The first group of people move frequently and also receive very few

letters from their friends. The second group of people move less, but receive many letters from their friends. The first group of people does not have to write letters to tell their friends their new address every time they move, since they know the new address is temporary and they also know that their friends have little chance of writing letters to them. This group of people sends out change-of-address notifications lazily.

For the second group of people, things are different. Whenever they move, they will write letters to tell their friends their new address. They know that they will live at the new address for a long time and they also know that their friends often write them. Sending letters to inform friends of their new address is urgent for this group of people.

In some cases, if people in the first group want to keep in touch with their friends, even though they move a lot, they have to notify their friends of their new location whenever they relocate. Although it costs more to send these change-of-address notifications, if their friends have good things to tell them, they can receive this news very quickly.

Back to the object world, our point is that objects with different behaviors should use different update policies. The objects can be divided into two classes: active objects and quiet objects. Active objects are defined as objects which move more often than they are invoked. Quiet objects are invoked more often than they move. To optimize object moving and invoking, it is not difficult to see that active objects should use a lazy update policy while quiet objects should use an urgent update policy. This policy is called an adaptive update policy.

Another component of the update policy is selecting which nodes should receive updated location information when an object moves. We do not want to update a node which has never invoked the migrating object. There are two kinds of relationships: object relationships and node relationships. When an object invokes another object, the first object is defined as a dependent of the second one. This is called an object dependency. A node dependency is derived from object dependencies. When an object at one node depends on another object at another node, the first node is defined as a dependent of the second one. The dependency can be strong or weak. The more invocations performed, the stronger the dependency. In order to improve performance, only the dependent nodes are updated.

A metric *activity* which is a real number between 0 and 1 can be used to measure the level of activity of an object. When the activity of an object is near 1, the object is defined to be active. Conversely, when the activity of an object is near 0, the object is defined to be quiet.

How should we choose the activity of an object? This depends on the performance requirements of the object. If we require an object to have good performance, we have to compute its activity according to its real behavior. The only way to do this is to record its behavior.

We may assign an object an initial activity estimate when the object is created. We can then adjust its activity based on its behavior.

If we require an object to exhibit some special behavior, we need to assign it an appropriate activity. Sometimes this is necessary and important. In some applications, the invoking objects require a quick response from an invoked object. If the invoked object's real behavior is active and lazy updating is used during migrating, the invocation messages will go through a long transfer path to reach the invoked object. Thus the invocation response will be delayed. In order to provide the invoking object with a quick invocation response, the invoked object can be assigned a low activity, which forces the system to use an urgent update policy. Although this will increase the object migration cost, a quick invocation response is gained as a tradeoff.

The object tracking function is separated from the Emerald run-time system and implemented as an independent protocol layer, OFP. The reliable datagram protocol, RDP is also designed and implemented to support the OFP protocol. RDP is built on UDP. The Emerald run-time system can also send messages directly through RDP. This is often used to reply to an invocation. The protocol relationships are given in Figure 1.1.

OFP presents its user, the Emerald run-time system, with the ability *to send a message to an object.* The semantics of RDP is *to send a message to a node.* The function of OFP is to map an object's reference to its current location. In this sense, it is a kind of distributed mapping.

When the distributed mapping fails, broadcast is used to query the nodes within a local area network. Because of its unreliability, OFP will force every node on the local area network to reply to the broadcast message (except nodes which are down). This ensures that we will get an object's location information if the object is on the local area network or if some node knows where the object resides.

One of the nodes in a local area network is used as an object tracking server. When an object migrates out of the local area network, the server must be informed. The same is true when an object migrates into the local area network. When a server receives a local broadcast message for an object that has left the local network and has registered with the server, the server will reply to the broadcast message with the object's foreign address.

If a local server loses an object address, the server should contact other internet servers to find the object. Currently, this part is not implemented in this thesis, and is left as future work.

In the real world, object invocation patterns are not random. There is some amount of locality in real object invocation patterns. Locality is defined as the probability of the next

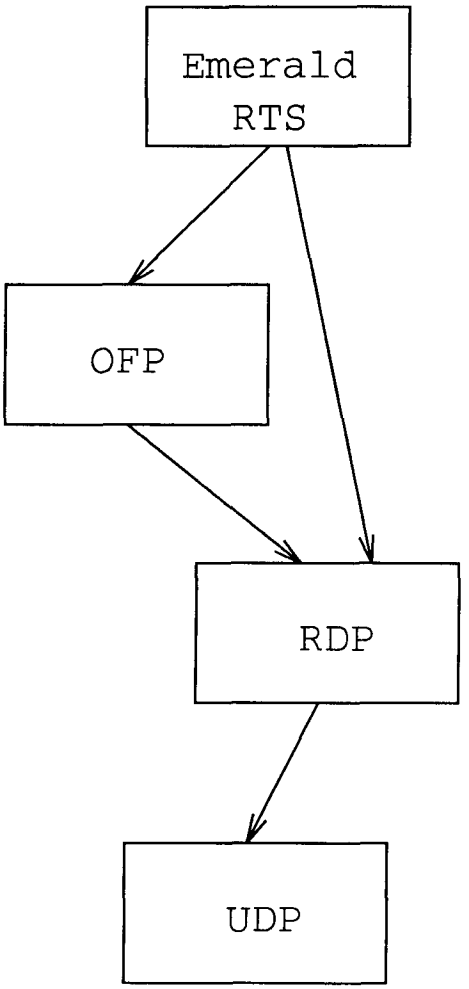


Figure 1.1: Protocols Dependency Relationships.

object to be invoked being the same as the previous one invoked. Through experimentation, the effect of object invocation locality on the performance of OFP protocols was studied.

All the protocols are implemented and tested on the x-kernel platform. The x-kernel is used to support protocol design and implementation. Its uniform interface and powerful utilities made it easy to implement and test our protocols.

Experiments have been designed and conducted to study the performance of the OFP protocol with different updating policies and different object invocation localities. The experiments are conducted on Sparc stations, using 12 x-kernel nodes.

The contributions of this thesis are:

- The use of an adaptive update policy in the forwarding address scheme to improve the performance of object tracking.
- The design and implementation of an object tracking protocol as a separated mechanism to transparently support Emerald object mobility.
- The design and implementation of a reliable datagram protocol to deliver reliable message transfer services to upper layer protocols or users.
- The extension of the object tracking protocol to the internet.
- The use of locality to model object invocation and study the effect of locality on OFP protocol performance.

The thesis is organized as follows: Chapter 2 will survey the various object tracking schemes used in existing distributed systems. Chapters 3 and 4 will discuss the design and implementation of the protocols RDP and OFP in detail. The experimental results will be given and analyzed in Chapter 5. Chapter 6 discusses the contributions of the thesis, future work and conclusions.

Chapter 2

Background and Related Work

Object tracking originates from object (or process) migration. Object migration has two components, *policy* and *mechanism*. The policy of a system decides which objects should be moved, when they should move, and where they should be moved, while mechanism decides how an object is moved.

2.1 Object Migration

Object migration in a distributed operating system is a facility to dynamically relocate an object from a processor on which it is executing (the source processor) to another processor (the destination processor). The basic idea of object migration is to move an object during its execution, so that it continues its execution on another processor, with continuous access to all its resources. This operation may be initiated without the knowledge of the running object or any object interacting with it. That means the migration is transparent; it should preserve all elements of computation and communication.

Here migration means both object mobility in object-oriented distributed systems such as Emerald and process migration in process-based distributed systems such as Charlotte, V, and Sprite. In fact, object mobility is different from process migration in two respects[JLHB88, Gos91]:

1. The unit of distribution and mobility is an object. Some of Emerald's objects contain processes, other objects contain only data such as arrays, records and single integers. Emerald processes are light weight threads. In process migration, the entire process address space is moved. Therefore the unit of object mobility can be much smaller than in process migration systems. Thus, Emerald's object mobility subsumes both process migration and data transfer.

2. Emerald has language support for mobility while process migration does not have language support.

In the previous chapter, several advantages of object migration have been mentioned. These include sharing load, improving communications performance, increasing the object availability, reconfiguration, utilizing special capabilities, moving data, improving remote invocation performance and simplifying garbage collection. There are many problems involved in object migration. The first group of problems occurs in the preparing stage. At this time, the migrating object and its destination have to be determined. For load sharing, there must be some mechanism to tell which nodes are heavy loaded and which are not. In Emerald, the user decides which objects should be migrated and where they should move.

The second group of problems occurs in the object moving stage. In this stage, the system has to decide how much of an object should be migrated, which parts of an object should be migrated and when they should be detached and moved. These problems are even more serious during process migration, since in process-based systems the processes which are migrated include an entire address space.

The process migration facilities which move entire address spaces often suffer a serious problem: as a program grows, the migrating cost grows in a linear fashion. Thus, moving the contents of a large address space stands out as the bottleneck during process migration. It dominates all other costs[Zay87].

A process to be migrated must be frozen, which means its activity is suspended and communication is deferred. Because freezing generates some problems (for example, a process can be suspended too long) the V system uses a special mechanism, called a *pre-copying* technique, to copy most of the process state before freezing it[TLC85][Che88]. This can reduce freezing time. However, this technique suffers from one drawback: it may increase the total time for migration.

To deal with this problem, a solution which allows instant execution on the destination node has been proposed in [Zay87]. The approach performs a logical transfer, which requires only a small portion of the address space to be physically transmitted. Thus, instead of moving the entire contents at migration time, an individual on-demand page (IOU) for all or part of the data can be sent. When a migrating process executing on the destination node attempts to reference memory pages which are not available locally, a request is transmitted to the source node to copy the desired blocks from their remote locations. This has two important consequences: context transmission times during migration are greatly reduced with this demand-driven **copy-on-reference** approach; and these times are virtually independent

of the size of the address space.

Unfortunately, this technique also introduces new problems. A destination node will depend heavily on its source nodes. The load of a source node will increase when the processes migrating from it increase. Also, the reliability of a destination node will depend on that of its source nodes.

2.2 Object Tracking

Object tracking is an implementation problem. When there is no limitation on object migration, tracking objects becomes even more difficult. Communication cost with a migrated object depends largely on the cost of tracking the location of the object.

In order to track a migrated object, its new location has to be made known. One option is to transmit information about the new location of the object to other nodes immediately after it migrates. Another option is to defer the transmission of location information until another object wants to communicate with the migrated object. In this case, the information about the new location of a moving object is made available either through a location server or through broadcasting. Node to node transmissions are expensive. Some systems use the first option, some the second. Other systems use both with the second option complimenting the first, since in the first option it is expensive to update all the nodes.

The advantages of a location server are that the migration mechanism is simple and the moving cost is reasonable. However, system reliability will depend on the reliability of the location server. It is also possible that the location server may become a performance bottleneck. Using only the second option to get a new location is expensive. It can increase communication time and cost.

If the system was aware of which nodes are likely to communicate with a migrating object, it could update only those nodes when moving the object. There are two approaches for choosing these communication *buddies*, either they are pre-determined or they are decided dynamically by the system. Different objects may have different communication buddies and it is unreasonable to assume that the system will have complete information. There may be some nodes which are not recognized as buddies of a particular object but which want to communicate with a migrating object. This means the second mechanism has to be used to complete the object tracking.

In the following subsections, object tracking schemes used by several existing distributed systems are surveyed in detail.

2.2.1 DEMOS/MP

DEMOS/MP is a distributed operating system developed at the University of California, Berkeley[PM83]. It is a message passing, process-based system. Messages are passed by means of links associated with a process. Links are managed by the kernel; the kernel participates in all link operations even though conceptual control remains with a process. Since a link always points to its creator process, links are context-independent. DEMOS/MP's migration facility uses the message forwarding approach to deal with the communication state and future communication.

DEMOS/MP implements message forwarding by leaving a forwarding address on the source computer. Messages sent to an object after it left will be resent to the object's new location by the source computer. One obvious problem is that indirect transmission of messages can degrade overall system performance because of long forwarding address paths. This conflicts with the use of process migration to improve overall distributed system performance.

There are two possible solutions. In the first, all messages addressed to the migrated process are returned to their senders as not delivered. In this system no forwarding process is required in the source computer. However, kernels of all senders have to perform some action to find the new location of the migrated process, for example, through broadcasting.

The second solution is based on updating links. By updating a link we mean updating the process address of the link. As a result of this operation, the link's unique identifier is not changed, but the computer location is updated to specify the new location of the migrated process. Updates of all links of a migrating process can be performed easily for processes which are not sending messages during migration. Since messages could be in transit, some forwarding mechanism is required. It was suggested that links should be updated as they are used, rather than all at once.

This update operation is performed by the kernel of the source computer when forwarding messages. A special message is sent containing the process identifier of the sender, the process identifier of the intended receiver (the migrated process) and the new location of the receiver. These solutions, however, are not implemented in DEMOS/MP.

In conclusion , DEMOS/MP uses a type of urgent update policy. Whenever an object is migrating, the source node will update all the links of the migrated process. This policy will increase the cost of migration but decrease the cost of further communication. On average, if processes communicate often and migrate rarely, the total migration cost will be reasonable. On the other hand, with frequent communications and rare migration, the cost of individ-

ual migrations will be high. Overall, system performance will depend strongly on processes' behavior.

2.2.2 Charlotte

Charlotte[Aea89] is a distributed operating system which offers a uniform, location-transparent message-based communication facility. Processes communicate with each other through links. One of the most important differences between DEMOS/MP and Charlotte is that in the latter, once a process migrates out of the source computer, no forwarding addresses or trail of stubs is left to perform future communication. The source kernel deletes all data structures relevant to the migrating process. The process migration facility of this system requires readdressing of the process's communication links.

In Charlotte, after a process is selected, process migration is performed by the kernel in three phases: negotiation, transfer and clean-up. During the transfer phase, the kernel of each of the migrating process's link ends is told the new logical address of the link. Each kernel that receives a link update message must acknowledge the message. Once all acknowledgements have been received, it is safe to start transferring the process's data structures, since no more messages will be received across its link.

Processes which have no link with the migrating process establish new links through the use of a name server. This name server must be informed of the new location whenever a process migrates.

Charlotte's updating policy is urgent update. However, it is different from the DEMOS/MP's policy. In Charlotte, the source node is not responsible for redirecting the messages of the migrated process. No forwarding address is left on the source node. But not all the nodes are updated. For the nodes which do not have links with the process, they have to depend on a third party to know where the process is. Therefore, whenever a process migrates, it is still necessary for the process to update a third party such as a name server.

The advantage of this policy is that future communication cost is low and better performance is achieved. However, there are also drawbacks to this policy. The disadvantages are that the dependency of a third node to establish a new link will reduce reliability, and the cost of migration increases. Charlotte may have good performance if processes move infrequently and communicate often.

2.2.3 V

The V[Che88] distributed system is an operating system designed for a collection of workstations connected by a high-performance network. The system consists of the V Kernel, a set of service modules, a command set, and runtime libraries. The user workstations are diskless, hence they communicate with various servers using the V kernel's inter-process communication (IPC) facilities.

Process migration in the V system is the migration of the logical host containing the process. A logical host is defined to be an address space within which multiple V processes may run. In other words, when a process is migrating, its child processes are also migrating, unless a child process is already executing remotely from its parent process.

After finding the destinated workstation, the logical host may be migrated through the following steps.

1. Initialization on the destination computer.

2. Pre-copying the state.

3. Freezing the state and completing the copy.

4. Unfreezing the new copy and rebinding references.

In step 3, once pre-copying of the state is completed, the logical host is frozen and the copying of its state is completed. Freezing is a very sensitive operation because although execution can be suspended within a logical host, messages can still be sent from remote computers.

Request messages are queued for the recipient process. A reply-pending packet is sent to the sender on each retransmission(as is done in the normal case). When the logical host is deleted from the source computer after migration, all queued messages are discarded. The sender has to retransmit its message to the destination host. Moreover, all the senders use the new binding of logical host to host destination address, which is broadcast when the logical host is migrated. Reply messages are handled by discarding them and replying using the retransmission capabilities of the IPC facilities.

The updating policy used by V is an urgent update. All the nodes on the local network will be updated by broadcasting. Thus updating depends heavily on V's broadcast facility. Each kernel which receives a broadcast message will keep the new binding in a cache for future communication.

One problem with this policy is the reliability of broadcast. There is no guarantee that all the nodes will receive the broadcast message. Some nodes will lose these messages, and in order to communicate with the migrated process, these nodes have to query the migrated process's new location by broadcasting.

Another problem is the broadcast can only be effective on a local network. For a long haul network the cost of broadcasting is prohibitive.

2.2.4 Locus

Locus[PW85] was a distributed version of UNIX, developed at UCLA. It has became a commercial product of the Locus Computing Corporation. Locus executes on a network of VAX minicomputers and M68000-based microcomputers connected by 10 megabit Ethernet. The system features a high degree of network transparency, a flexible and reliable file system, and a recovery facility for system failures. It also provides support for accessing resources on other machines running different operating systems.

In Locus, each process PID includes a site identifier SID which specifies the site on which the process was created. Locus uses this site as the synchronization site for the process. If the real original site, as specified by the SID, is not currently on the network, then a surrogate original site is used. All sites on the network agree in advance who will act as surrogate.

When a process migrates, its original site and surrogate will be updated with the new location of the process. This is an urgent update, however, updates are done only on the origin and surrogate and not on all nodes. Thus, when a process wants to send a message to the migrated process, it will send the message to the original site or its surrogate, and the original site or surrogate will redirect the message to the migrated process.

The policy used here has good performance for both moving and future communication. For each migration, only one or two nodes are updated. But since future communications depends on the original node, the original node shoulders more burden. Second, the reliability of the original node directly affects the whole system. Moreover, there is also the possibility that the location of some processes may be lost forever if both the original node and the surrogate fail.

2.2.5 Sprite

Sprite[DO91] is an experimental network operating system developed at the University of California at Berkeley. It is part of the SPUR project, whose goal is to develop high performance multiprocessor workstations.

Transparent process migration is one of Sprite's features. Sprite uses a home node concept with each migrating process supporting transparency. In Sprite, there are two aspects of transparency, both of which can be defined with respect to a process's home node. When there is no migration, the process will be executed on the home node. If a process is migrated, it should appear as if the process were still executing on its home machine. In order to achieve transparency, Sprite uses four different techniques:

1. Location-independent kernel calls.
2. Transfering process state from the source machine to the target at migration time.
3. Forwarding kernel calls home and forwarding from home to remote.
4. A few kernel calls must update state on both sites: the home node and the current node.

Sprite's home node is similar to LOCUS's original site. Home nodes are used for forwarding messages for future communication. To improve performance, each machine will cache the recently-used process identifiers and their last known execution sites. An incorrect execution site is detected when it is used again. The correct site is found by sending message to its home node.

In general, Sprite uses an urgent update policy. Future communications pass through the home node. Sprite shares the same drawbacks as Locus, namely, reliability and dependency on the home node performance.

2.2.6 Emerald

Emerald is an object-based language and system designed for the construction of distributed programs. It was developed at University of Washington[BHJ+86][BHJL86][Hut87]. Object mobility is one of the important features in Emerald.

Object mobility has the potential for improving the performance of remote invocation by moving parameter objects to the remote computer for the duration of the invocation; it can also simplify distributed garbage collection by moving objects to computers where references exist.

The natural construct which allows processes and data to be treated uniformly is an object. An Emerald object has four components:

1. a unique network-wide name;

2. a representation, the data local to the object, which consists of primitive data and references to other objects;

3. a set of operations that can be invoked on the object;

4. an optional process.

Objects containing a process are active. Otherwise, they are passive data structures.

The Emerald mobility concepts are integrated into the language. This results in the possibility of extensive cooperation between the compiler and the run-time system. This leads finally to large improvement on efficiency over previously discussed systems.

In particular, object mobility in Emerald is provided by a small set of language primitives. These primitives are shown in Figure 2.1.

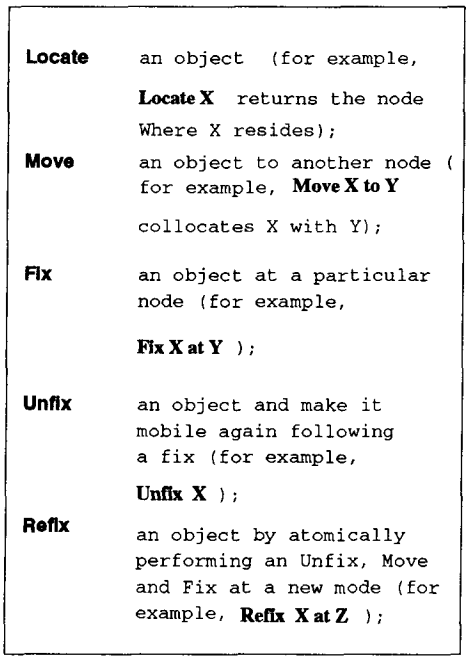


Figure 2.1: Emerald Language Primitives.

Some of the primitives require transparent object location. The Emerald run-time system supports location transparency by tracking migrating objects. In Emerald, there is no limitation on object migration. So it is very important to efficiently track migrating objects. The principle used by Emerald is to put the migration cost onto invoking nodes, not the source nodes of migrating objects. In other words, Emerald adopts a lazy updating policy.

Emerald uses a forwarding address scheme to track migrating objects. Emerald's forwarding address is a tuple (timestamp, forwarding-address). The timestamp is used to determine whether a forwarding address is up-to-date. When updating a node, if the forwarding address of an object in the updating message is older than that of the object on the updated node, no updating is done.

Using clock time to generate timestamps is expensive because the protocol used to maintain time consistency on every node is not trivial. In Emerald, an object's *hop count* or migrating count is used as the timestamp of the object. When an object is created, its timestamp is set to zero. Whenever the object migrates, its timestamp is incremented. This timestamp is good enough to maintain up-to-date forwarding addresses and abandon delayed or duplicated updating messages[Fow85]. Another usage of timestamps is to limit the moves of an object when its hop count reaches some threshold(e.g., the number of nodes in the system) to avoid a situation in which an object might "outrun" messages trying to reach it.

In fact, Emerald's updating policy is not completely lazy. When an object migrates, its source and destination nodes are updated since this can be done without cost. Each node invoking an object gets the object's new location through piggybacking on a reply message[Jul88].

Emerald uses two protocols to locate an object. First it optimistically uses weak location. If that fails, it uses the more expensive strong location to ensure correctness. For example, when the kernel performs a remote invocation, it uses weak location to obtain a location hint and sends an invocation message to the presumed target node. If the hint turns out to be right then the invocation is performed efficiently using a minimum number of network packets. If the hint is wrong then strong location is used to locate the object and the invocation message is forwarded. Using weak location works well in most cases where objects are not moved frequently.

Emerald strong location uses a forwarding address scheme. Every time an object moves, it leaves behind a forwarding address so that any reference to the object using its old location may be forwarded to the object's next location. In the absence of node crashes, a forwarding address can be used to provide strong location semantics. When a node crashes, its forwarding addresses are lost but objects that have moved away from the node before the crash will still be accessible even though they cannot be found using an access path that includes the crashed node[Jul88].

Broadcasting is used as the next step of strong location when the forwarding address fails to locate an object. In case some nodes fail to reply to a broadcast message, node to node communication is used to force all nodes to reply. Timeout is used to avoid unlimited blocking

when waiting for a reply.

2.3 Conclusion

Although different schemes have been used by the above systems, one of the common feature of these schemes is that the same updating policy is used by each scheme on all migrating object. Its advantage is that the object tracking mechanism is simple. But migration costs of some objects are higher, depending on their behaviors. In order to have good migration performance for every object, as mentioned in last chapter, different updating policy should be used on objects with different behaviors.

A forwarding address scheme is also used in our object finding protocol. The relationship between object behavior and optimal updating policy is studied. The following chapters will discuss these protocols in detail.

Chapter 3

Reliable Datagram Protocol

Object tracking requires a reliable data transmission service. There are two types of protocols, connection-oriented and connectionless, which can be used to support our object tracking. Which one should be used? We consider that a reliable packet oriented protocol is most appropriate for our problem although we could also use TCP. RDP is a Reliable Data Protocol which is built on UDP. It adds reliability to UDP.

RDP is a protocol which is designed to provide a reliable datagram transfer service for OFP, the Object Finding Protocol. TCP is more suitable for large, continuous data transmission. The data transmission unit in TCP is a byte. For RDP, the data transmission unit is a packet. Packets are completely independent of each other. The semantics of RDP is to *reliably send a message to its destination with a given internet address.*

RDP can also be built directly on IP. However for portability, we choose to build RDP on UDP.

3.1 Reliable Data Transmission

In order to reliably send a packet, the sending side must receive an acknowledgement for each received packet. Any packets which are not acknowledged in a specific period of time are assumed to be lost and are retransmitted. To prevent the sender from waiting forever for a reply, a timer is used for each transmission of a data packet. If the timer expires before receiving an acknowledgement, the sender will resend the packet and reset the timer. After some number of failed retransmissions, the sender will stop trying and report the failure to the upper layer.

A sequence number is used for each data packet. An acknowledgement includes the acknowledged data packet's sequence number. A sequence number is four bytes long and suitable

for future high speed networks; in a high speed network, a two byte sequence number can easily wrap around in the network life time so that it is impossible to know whether a message is a duplicated message or a new message with the same sequence number (See TCP extension RFC1323).

The first sequence number is randomly generated. Then it is incremented and used as the sequence number of the next sending packet. Each RDP session has its own sequence number.

A sliding window mechanism is used on both the sender and receiver sides. These windows are used to control transmission throughput. Currently window size is fixed on both sides. A sender creates its own sliding window when a packet is sent by an upper layer. At the receiving side, when a first data packet is received, the window is initialized based on the received packet.

The RDP sending window mechanism consists of a start sequence number, L_SEQ_NUM, which is initialized to a random number generated by an RDP session, the size of window, W_SIZE, an array of bits, WIN_BITS, which represents the status of the packets that have been sent, and a base pointer which points to the start of the window.

A RDP receiving window mechanism consists of a base sequence number, B_SEQ_NUM, which is initialized to the sequence number from a first data packet, the starting of sequence number, L_SEQ_NUM, which is initialized to B_SEQ_NUM, the size of window, W_SIZE, an array of bits, WIN_BITS which represents the status of packets that have been received and a base pointer which points to the start of the window.

There are several message types defined in RDP. These are **FIRSTDATA, NACK, MSG_SND, MSG_ACK, MSG_SYN**. They are explained in detail as follows:

- **FIRSTDATA:** The first data packet sent is of type FIRSTDATA. This data type is used to transfer the first data packet between two nodes and triggers the construction of the receiver's sliding window.

- **NACK:** Packet of this type is sent to the sender whenever the receiver receives a FIRSTDATA packet with the same base sequence number as that of the receiving window mechanism. The data packet may be a duplicated packet or one that is sent after the sender recovered from a crash and happened to use the same sequence number as the one used last time. If the sender has crashed and recovered, it resends the FIRSTDATA packet with a new sequence number after it receives the NACK message.

- **MSG_SND:** The data packets sent after a FIRSTDATA packet have this type.

- **MSG_ACK:** This type of packet is sent as an acknowledgement of the received data

packets including both MSG_SND and FIRSTDATA packets.

- **MSG_SYN:** This type of packet is replied when a node receives a MSG_SND packet before receiving a FIRSTDATA message. This may happen when the FIRSTDATA message got lost. In this case, the sender will resend all the outstanding data packets with the first one as a FIRSTDATA packet.

Each time a new RDP session is created, the first packet sent has type FIRSTDATA; the rest of the data packets sent through the session have type MSG_SND. When a machine receives a FIRSTDATA packet, and a window mechanism has not been established with this sender, a new one is created and a MSG_ACK is returned to the sender. Otherwise, the receiver will check whether the packet has the same base sequence number as that of the existing window. If this is the case, the packet can either be a duplicate or one that is sent after the sender has recovered from a crash. A NACK packet is then sent back to the sender to let the sender confirm that it is establishing a new connection. If the sender receives a NACK message for an unacknowledged FIRSTDATA packet, it will resend a FIRSTDATA packet with a different base sequence number. If a FIRSTDATA packet with a different base sequence number from that of the receiver's window is received, the old window will be destroyed and a new one will be created.

When a MSG_SND data packet is received by the receiver with no window, a MSG_SYN packet is returned to the sender. If a window exists, and the packet falls into the window, the corresponding bit of the window is set and the window slides if possible. Whenever the window slides, its lower bound sequence number will be updated. If the packet falls beyond the upper bound of the window, the packet is dropped. If the packet falls below the lower bound, the packet is dropped and a MSG_ACK message is sent back to the sender.

After receiving a MSG_SYN message, the sender will resend all the outstanding data packets with the first one as FIRSTDATA type. If there are no outstanding data packets, the MSG_SYN message is simply ignored.

3.2 Implementation

RDP is implemented in the x-kernel. In the x-kernel, all protocols has the same interface. The x-kernel has a group of utilities to manipulate messages and to manage mapping, therefore it is easy to implement protocols in the x-kernel (See [HP91]).

In the x-kernel, a protocol may depend on one or more other protocols. A protocol may also deliver services to one or more other protocols. There are two kinds of objects in the

x-kernel corresponding to each protocol: protocol objects and session objects. Each protocol has exactly one protocol object and may have zero or more session objects. A protocol object contains the information about the protocol. A session object contains the information about a specific communication channel through the protocol. Protocol objects are created statically while session objects are created dynamically.

RDP is built on UDP. The header structure of an RDP packet is given as follows:

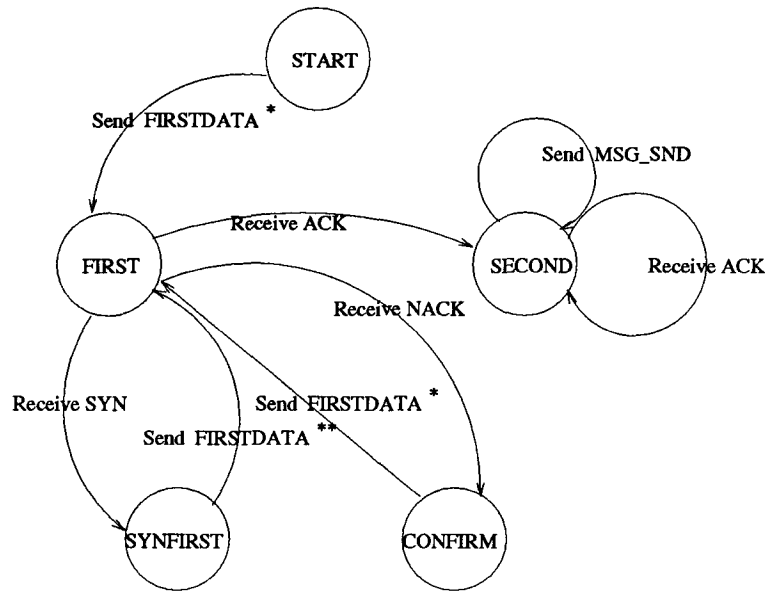
```
struct header {
        UDPport sport; /* source port */
        UDPport dport; /* destination port */
        u_long sqnum;  /* sequence number */
        u_short ulen;  /* message length */
        u_short msgtype/* the type of the message */
} HDR;
```

RDP uses a port number to identify the upper layer protocol. In other words, each upper layer protocol has to have its own unique port number registered with RDP so that a received packet can be sent to the correct upper layer protocol. A port number is a two byte short integer. The sequence number is a four byte integer. Each packet has a unique sequence number. Message length is a two byte integer which contains the sum of the header size and message size. Message type is a two byte integer.

To send a message through RDP, an upper layer user has to tell RDP its local port number and the message destination port number. The local and destination port number are used to uniquely create a RDP session at the local side as well as the remote side.

In RDP, a sender has five states. Its state transfer diagram is given in Figure 3.1. A receiver has three states as shown in Figure 3.2.

In the state transfer diagrams, both the sender and the receiver have no close state. Currently there is no close phase in RDP. In our current system, as soon as a RDP session is opened, it exists forever. If we choose to have a close phase, we have to decide who should be responsible for closing the RDP session. There are two choices. One is to let RDP users take the responsibility and another is the RDP. In the later case, a garbage collector can be used to automatically check each session. If a session has not be touched for a specific amount of time, it will be collected and a function supported by users can be called through the garbage collector. The function is user specified. In our current RDP, the function is null. Nothing is



* The FIRSTDATA message should use a new base sequence number.

** The FIRSTDATA message should use the original base sequence number.

Figure 3.1: Sender's State Transfer Diagram

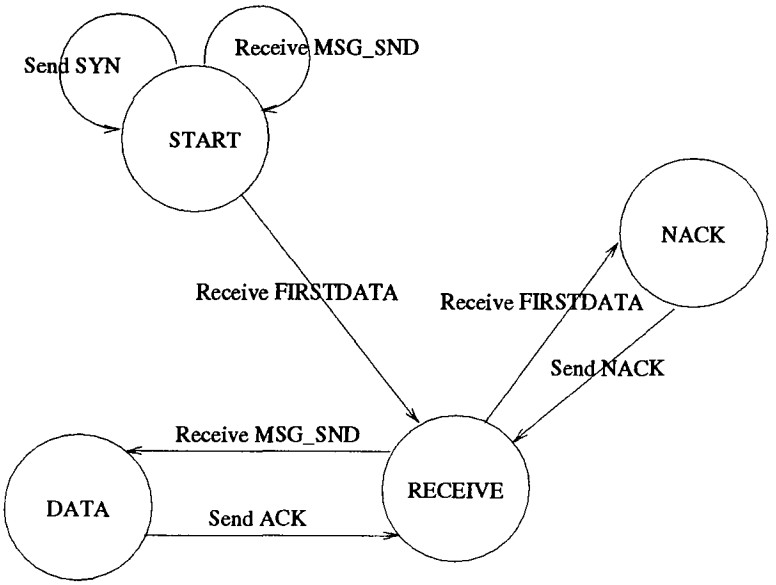


Figure 3.2: Receiver's State Transfer Diagram

done. This works well for RDP because RDP can recover itself from the crash of either side. Collecting a session is similar to a node crash.

3.3 Conclusion

RDP is a reliable packet oriented protocol. It is built on UDP and has a window mechanism which is similar to that of TCP. The window mechanism is used to control packet replies and retransmission. Communication is started by sending a FIRSTDATA data packet which is a data message as well as a control message.

A garbage collector is used to close the sessions which have not been used for certain amount of time so that the closing phase is not necessary in RDP.

Sliding windows have been used to control data throughput and message retransmission. In TCP, the data unit is byte, but in RDP the data unit is a packet and the size of a packet may vary. So sliding windows just roughly control communication throughput. In other words, they just control the packet transmission number but not the real data size.

Chapter 4

Object Finding Protocol

OFP is a protocol which is designed to send a message to an object. It is built on RDP. RDP sends a message to a node with a given address while OFP sends a message to an object with a given ID. OFP will map the ID to the node address at which the object currently resides. If objects do not migrate, it would be easy for OFP to map an object ID to a node address by simply keeping a static mapping table. If objects move arbitrarily, the mapping process becomes more complicated. Whenever an object migrates, its old address must be updated.

A forwarding address scheme is used in the OFP protocol to locate objects. Whenever an object migrates, it leaves its new address at its last location. The new location is called the forwarding address. It is used as a hint to locate the object.

A mapping table is necessary for each node. The forwarding address of an object is kept in the mapping table. Each object from the mapping table has a dependent set. The dependent set of an object contains the location of the other objects which rare recently invoked the object.

There are several schemes used in distributed systems to track object migration. In general, they can be classified into two categories. In the first category, a lazy update policy is used. In the second category, an urgent update policy is used. Both of the policies have some disadvantages if they are used separately. A better policy can be used instead of these two policies. It is called an adaptive update policy which is a combination of both the lazy and urgent update policies in one system.

How do we distinguish a lazy policy from an urgent policy? There is no absolute definition. Generally speaking, a lazy update policy puts more cost on object invocation while an urgent update policy puts more cost on object migration.

The adaptive policy is based on an object's *activity*, which was defined previously as a real number between 0 and 1. As the activity increases, there are more object migrations and less

object invocations. The activity of an object determines which policy is used on the object.

In the following sections we first describe the mechanism of the OFP protocol. Following that, we describe the various update policies which will be used in OFP to improve the performance of object tracking.

4.1 Mechanism

We have mentioned that the function of OFP is to map an object ID to a node address where the object is supposed to be located. Each node has a local mapping table. If a user wants to send a message to an object, the local mapping table is used to relate the message to its destination node. The local node first checks whether the object is on the local node. If not, it maps the object ID to a hint location and sends the message to that location through RDP. The node which receives the message does the same thing: checking whether the object is local and forwarding the message if necessary. The procedure continues until the message reaches its destination node. This is a dynamic and distributed mapping.

The semantics of OFP is to send a message to an object with a given an object ID, however it does not guarantee that messages are received. OFP can detect a failure on the sending side but cannot detect failures on the receiving side.

There are several reasons to have the specific kind of OFP semantics.

1. Send failure can be easily tested and reported to upper layer users.

2. An OFP message may go through more than one node. It is difficult and expensive to guarantee that a message is received successfully because every node on the forwarding address chain has to return an acknowledgement to its predecessor.

3. Most OFP messages are for object invocation, and invocations always request replies. The upper layer getting a reply implies that the invocation is successful. Otherwise, the invocation is considered a failure. Therefore, no stronger message transmission semantics are necessary.

4. Even if the object is received successfully, the receiver may still crash after receiving a message but before sending a reply. This is equivalent to a receiving failure from the sender's point of view.

In our system we assume that there are two important operations which can be applied on any object: invocation and migration. Object invocation involves sending a message to an

object. This is supported by OFP. Object migration involves sending an object itself to a node. This is supported by RDP. Some applications request to move an object to a location where another object locates. In this case, a migration message should be treated as an invocation message and sent through the OFP layer. We assume that the object migration always require a node address.

In each node, there is a local mapping table which is used to keep track of objects migrated in and/or out of the node. Whenever an object migrates, the OFP layer has to be notified about the migration by the upper layer user.

4.1.1 Forwarding Address Scheme and Broadcasting

OFP uses forwarding addresses to track moving objects. Whenever an object migrates, a forwarding address is left behind on the original node. Future messages sent to the original node will be forwarded according to the forwarding addresses. A node's internet address is used as a forwarding address.

The forwarding address scheme has the following advantages:

- The location information of an object is distributed among a variety of nodes on the network. No single node has to take the responsibility of keeping records of all the objects' location information. Systems are more fault-tolerant.
- Highly replicated information can accelerate object searching, accelerate object invocation and improve object tracking efficiency.
- An adaptive update policy can be used to optimize the performance of object migration and invocation.

Each forwarding address has a timestamp associated with it. The timestamp indicates the age of the forwarding address. When updating a forwarding address, the timestamp of the update message is compared against the one found in the mapping table. Only when the timestamp of an update message is up-to-date can the updating be considered effective and executed. The timestamp is useful in avoiding forwarding address chain loops.

The hops that an object has moved through are used as the timestamp. Initially, each object's timestamp is set to zero. When an object migrates, its timestamp is incremented. Using an object's moving hops as its timestamp has been proved effective and correct in eliminating old and duplicated update messages and in preventing the forwarding address chain from forming a loop.

The forwarding address mechanism can fail to find an object in only two situations: either the sending node has no location information at all for an object or a node failure has caused a break in the forwarding address chain. When the forwarding address scheme cannot locate an object, broadcast is used as the next step to locate an object. Broadcast is only effective on a LAN where broadcast is supported.

When the upper layer user wants to send a message to an object through OFP, it transfers the message and the object ID to the local OFP. OFP checks the local mapping table, and then decides whether the message should be sent or not. If the object is not at the local node and also there is no hint from the local mapping table, the message is broadcasted on the local area network. Otherwise, the message is sent to the location mapped from the local mapping table.

We assume that every node knows of the existence of every other node on the local network. Every node will reply to the received broadcast message. If a node has an object, it will reply with an UPDATE message. Otherwise, it will reply with an UNKNOWN message. The UPDATE message contains the current location of the object, while the UNKNOWN message indicates that the object is not on the node.

After broadcasting a message, the sender will count all the received UNKNOWN messages until an UPDATE message is received. Then it will drop the rest of UNKNOWN messages. If an UPDATE message is received, it will use the new object location contained in the UPDATE message to update the local mapping table provided that the UPDATE message has an effective timestamp. If no UPDATE message is received and all the UNKNOWN messages have been received before a timeout, the local node will tell the upper layer that the object does not exist on the local area network. If not all the UNKNOWN messages are received before timeout, the local node will use point to point messages to ensure that the nodes from which no message is received are either down or eventually reply.

OFP adopts non-blocking message transmission semantics. The reason to use non-blocking transmission is that the OFP message transmission does not guarantee that messages are received by objects. After a message is transferred to the OFP layer, control returns to the sender immediately after initiating the message transmission. If broadcasting for an object's location is required, all the messages sent to the specific object are kept in an object waiting queue so that message transmission can proceed in parallel with the object location.

After receiving a DATA message for an object, the receiver will first check whether the object is local. If it is, the receiver will send the message to its upper layer, add the source node to the object dependent set and send an UPDATE message to the source node if the

object's activity is higher than the selected threshold. If the object is not local, the receiver then tries the local mapping table. If it fails to get any hint from the local mapping table, the last node is added to the dependent set and an NDATA message is sent back to the source node to indicate that something is wrong with the object location. If local mapping is successful, the last node of the message is added to the dependent set, the local node is added to the message header as the last node and the message is forwarded to the next node.

We mentioned that object migration bypass the OFP layer and goes directly through the RDP layer. When migrating an object in or out of a node, the local OFP must be informed about the migration. OFP will register the migration in its mapping table.

When OFP updates its mapping table, it also checks the object dependent set. If the dependent set is not empty and the object activity is larger than the activity threshold, then update messages are sent to all the dependents. Finally, the dependent set is cleared.

After receiving an update message, OFP first checks whether the update is effective by comparing the object timestamp contained in the update message with the timestamp in the mapping table. Then OFP checks if the dependent set is empty. If not and the object activity is larger than the threshold, an update message is sent to each of the dependents.

To reduce the transmission of update messages, each update message contains an updated node set. At first, when a node sends an update message, the sender and dependent set of an object are added to the updated node set, because a node may be present in several dependent sets for the same object. The dependent set is checked against a received updated node set to eliminate duplicate transmission.

In the x-kernel, when a user wants to send a message through a lower layer protocol, it first has to open a session of that protocol. Opening a session usually requires some communication and therefore two participant addresses must normally be given as parameters to the open request. For some server sessions, only the local address is known when the session is opened.

When opening an OFP session, only one participant address need be specified. The object ID of the object to which messages should be delivered need also be specified.

4.1.2 Message Types

There are several message types defined, which are DATA, UPDATE, UNKNOWN, NDATA and BDATA. They are described in detail as follows:

- DATA message: This is the OFP data message sent to an object from one node to another node. Usually, object invocation requests are sent as DATA messages.

- UPDATE message: The message is used to update an object location. It is sent when a node finds that an object hint is out of date. The message is sent when an object migrates or when an object is invoked. When an object migrates, all its dependents are updated. When an object is invoked, only the source node is updated if the object hint on the source node is out of date.

- UNKNOWN message: This type of message is used to reply to a broadcast message when the location of an object is unknown.

- NDATA message: This type of message is used to reply to a DATA message that the object is unknown and no forwarding address exists for the object.

- BDATA message: This type of message is broadcast to all the nodes on a local network. When local mapping fails, message broadcasting is used. Broadcasting only works on local networks.

  If an object is out of a local network, a server is used. Each local network has a server. In principle, any object which migrates in or out of a local network has to be registered on its local server. If the local server has a hint for an object, it will reply with the hint to the local node which wants to communicate with the object. If the local server has no hint or the hint is lost, it will contact the other local servers on the wide area network through end to end communication. This is left as future work. Currently we assume that the objects only migrates within a local area network.

All the OFP message types have the same header structure which is given as follows:

```
struct header {
        u_short msgtype;
        u_short protocolnum;
        TSTAMP  timestamp;
        OBJID   objectid;
        IPhost  srchost;
        u_short pnodenum;
        u_short ulen;
        IPhost  lastnode;
        IPhost  lastlastnode;
} HDR;
```

- `msgtype`: The field specifies the type of the message as described above.
- `protocolnum`: There may be more than one OFP user. To identify these users and correctly send a received message to its upper layer user, each user has an ID or protocol number registered in OFP.
- `timestamp`: This is the timestamp of an object. It is initialized to zero and records the number of times an object has been moved. It is used only in UPDATE messages.
- `objectid`: Each object has a unique ID. It is used to identify which object the message is for. For a DATA message or a BDATA message, the object ID represents the object to which a message is being sent. For an UNKNOWN message or a NDATA message, it represents the object for which no location information is known. For an UPDATE message, it represents the object for which new location information is included in the message.
- `srchost`: For a DATA type a BDATA type or an UPDATE message, it indicates the source node of the message. For an UNKNOWN or NDATA message, it indicates the source node of the message to which the UNKNOWN or NDATA is a reply.
- `pnodenum`: This field is used to count the number of nodes a specific message has passed through. This field can be used to test whether the last node is same as the source node. If the pnodenum is larger than 1, it means the source node is different from the last node. This is useful to eliminate unnecessary update message sending.
- `ulen`: This is the total message length including the body of a message.
- `lastnode`: This contains the address of the last node through which a message has passed. In a forwarding address chain, it is the node before the destination node.
- `lastlastnode`: In a DATA or BDATA message, it contains address of the second last node by which a message has passed. In the forwarding address chain, it is the node before the `lastnode`. In an UPDATE message, this field is used to store the new location of an object.

4.2 Policies

When a system is designed, there are two aspects which need to be taken care of. They are correctness and efficiency.

In last section, we have discussed the mechanisms used in OFP protocol. The mechanisms only guarantee that the OFP protocol functions correctly but not necessarily that it functions efficiently. To make the OFP protocol function efficiently, the updating policies used in OFP protocol have to be studied. In the following sections, we will study different updating policies which can be used in the forwarding address scheme and try to find one with the best performance.

4.2.1 Lazy Update Policy vs. Urgent Update Policy

The forwarding addresses can form a chain. The length of the chain depends on the system behavior and the update policy used. If the lazy update policy is used in a system with a lot of object migrations a long chain can be easily formed. A long address chain makes the costs of invocations higher although there is almost no update cost for object migration. In this system, invocation cost dominates while the migration cost is less important. Every invocation has to pass through a long chain and generates a lot of forwarding messages. On the other hand, if the urgent update policy is used in a system with a lot of active objects and very few object invocations, the result would be low invocation cost and high migration cost. In this case, the cost of object migration dominates the cost of invocation.

To reduce the total object migration and invocation cost, different objects should use different update policies. For active objects, migration is the main part of their behavior. To reduce its behavior cost, we have to reduce its migration cost. On the other hand, for quiet objects with a lot of invocations, invocations are the primary part of their behavior. To reduce its behavior cost, we have to reduce its invocation cost. We use the lazy update policy to reduce object migration cost and the urgent update policy to reduce object invocation cost.

The problem is how to define a quiet object and an active object. We use an activity threshold as our classification standard. An object with an activity larger than the threshold is called an active object. Otherwise, the object is called a quiet object. The most appropriate threshold value is determined through experiments. Experiments will be discussed in the next chapter.

Using the urgent update policy can reduce the object invocation cost. On the other hand, it increases the object migration cost. On the contrary, using the lazy update policy can reduce the object migration cost but it increases the object invocation cost. The system performance depends on the total cost of object invocation and object migration. We believe that by adjusting the ratio of migration cost with that of invocation cost, optimal system performance can be achieved. The adaptive update policy tries to reach an optimal system performance by

making each individual object reach an optimal performance through adjusting the cost of its migration and invocation based on its activity. This will be discussed in the following section.

The lazy and urgent concepts are not absolute; they are relative to each other. Precisely, our lazy policy is that when object is moved, no update messages are sent to any other objects. This is an extreme case of the lazy policy. By using this update policy, object migration cost is minimum. There are no update messages sent when objects migrate.

There are two parameters which affect the urgent updating policy. One parameter is the size of the dependent set carried to the next node when object migrates. Another parameter is the number of hops the dependent set is carried on. By adjusting the two parameters, many different versions of the urgent update policy can be generated.

We define 3 versions of our urgent update policy, cleverly named version 1, 2, and 3, in which dependent sets are carried for 0, 1, and 2 hops respectively. Therefore, version 3 is more urgent than version 2, while version 2 is more urgent than version 1. For all of these policies, every time an object migrates, an update message is sent out to all its dependents. The larger the dependent set becomes, the more update messages that are sent. The further the dependent set is carried on, the larger the dependent set becomes.

4.2.2 Adaptive Update Policy

One of the important parameters in a forwarding address scheme is the update policy. This includes when to send update messages and to which nodes the update messages should be sent. Two important things should be taken into consideration when an update policy is selected: object behavior and object relationships. Object behavior decides when to send an update message. Object relationship determines nodes to which an update message should be sent.

We use the activity to describe the behavior of an object. An object with activity value A means that the object has probability A to migrate and probability $(1\text{-}A)$ to be invoked.

The dependent set is used to describe an object's relationship with other objects. Each object has a dependent set. The set can be built dynamically and/or statically. An object only sends update messages to the nodes in its dependent set when it migrates. When an object X invokes an object Y, the node on which object X is located is considered as a dependent of the object Y and is added into its dependent set.

How many dependents should be updated when an object migrates? This is another factor which affects the cost of object migration and object invocation. If all the dependents are updated, the object migration cost is high. However, the object invocation cost is low. Conversely, invocation cost is high and migration cost is low with the lazy update policy.

Currently the dependent set can only describe whether an object has relationship with the other objects. It cannot describe how strong the relationships are. We could use fuzzy sets to precisely describe the strength of each object relationship. By using fuzzy sets, the relationship can be described as a value from 0 to 1. The larger the value, the stronger the relationship.

Our adaptive update policy is based on object activity. It is a combination of the lazy and urgent update policies through an activity threshold. For a lazy update policy and an urgent update policy, if there exists a threshold from these two policies, then an adaptive update policy exists. This adaptive update policy is based on the specific lazy and urgent update policies and the specific activity threshold. The threshold is determined through experiments on the lazy and the urgent update policies.

4.3 Conclusion

OFP is built on RDP. The semantics of RDP is to send a message to a given node. The semantics of OFP is to send a message to a given object. It maps an object ID to an object's current location. The current location of an object changes with time, therefore the mapping is dynamic and distributed.

A forwarding address scheme is used in OFP to track object migration. To reduce the object migration and invocation cost, it is important to balance the cost of object migration and object invocation. We believe that an adaptive update policy can be used to optimize this cost.

There are two factors affecting an update policy. The first factor is when to send update messages. Updating can be done when objects move or when objects are invoked. Another factor is how many dependents should be updated. All the dependents, part of the dependents or no dependents may be updated depending on the selection standards.

Three different versions of urgent update policies and one lazy update policy have been discussed. By combining an urgent update policies with the lazy update policy, we can get an adaptive update policy. There is a decision table for each adaptive update policy. A decision table is generated by comparing the experimental results of an urgent update policy and that of a lazy update policy. The decision table tells you at what activity and what locality an urgent or a lazy update policy should be used. Based on the decision table, the activity, and the locality, the adaptive update policy choose to use either the urgent or the lazy update policy.

OFP has been implemented entirely in the x-kernel environment. In next chapter, we will

describe how to design and conduct experiments on lazy and urgent update policies and how to use these experimental results to generate a decision table which can be used by an adaptive update policy.

Chapter 5

Experiments and Analysis

Experiments were conducted in a system which includes an implementation of the OFP and RDP protocols. In OFP, the forwarding address scheme is used to track object location. The update policy used to cut forwarding address chains is quite important for improving OFP performance. Three different update policies were used: the lazy update policy, the urgent update policy and the adaptive update policy. This section discusses our experiments, presents the results of those experiments and discusses the experimental consequences.

5.1 Experiment Design

Each experiment was run using 12 Sparc workstations. Each workstation ran one x-kernel node, for a total of 12 x-kernel machines.

Each x-kernel machine initially registers 10 objects. Before starting, all the machines have information on the location of all objects. In each experiment, every machine independently performs 200 operations. Each operation is either an object invocation or an object migration. The number of invocations and migrations are controlled by the object activity: larger activity values result in migrations, smaller activity values result in more invocations.

In a real system, there is some amount of locality in the pattern of object invocation. To make our experiment more meaningful, locality is incorporated into our model. Invocation with locality 0 will randomly invoke objects. Invocation with locality 1 will always invoke the same object. Separate experiments have been completed using different localities. This study does not investigate the actual level of locality in a real system; that can only be done in the context of a real application of object mobility. Different systems may have different locality. Our purpose in using locality in our experiment is to show that locality does affect the performance of OFP and must be taken into account.

Update Message Cost Per Migration											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.80	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.99	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.1: Lazy Update Policy Update Cost Per Migration.

In our experiments, we vary the level of object activity and invocation locality, and collect statistics for all of our update policies. In each experiment, the following information is collected:

1. Total number of forwarding messages; reflects the cost of invocation.
2. Total number of update messages; reflects the cost of object migration.
3. Total object invocations.
4. Total object migrations.

Based on this data, we calculated the cost of each invocation, the cost of each migration and the cost of each operation. We used the cost of each operation as a standard to compare the performance of the update policies.

Normally, reply messages are required for invocations. Reply messages can be used to piggyback an object location. In our experiments, we assumed that there was no sending of reply message and all the object location information was contained in update messages.

5.2 Experimental Results of the Lazy Update Policy

The following tables show the experimental results from the lazy update policy using various object invocation localities and object activities.

From the results in Table 5.3, we can see that there was no update cost during the experiments. This is consistent with the lazy update policy which never sends any update messages. The only cost is from message forwarding which is generated during object invocation. This is shown in Table 5.4. From Table 5.5, we can see that cost becomes higher when the activity

Forwarding Message Cost Per Invocation											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.10	0.10	0.10	0.09	0.08	0.09	0.07	0.11	0.03	0.15	0.18
0.20	1.28	1.20	1.20	1.10	1.15	1.27	1.29	1.12	1.26	1.01	0.77
0.40	1.63	1.66	1.68	1.76	1.72	1.59	1.60	1.68	1.77	1.65	1.66
0.60	1.89	1.84	1.92	1.74	1.70	1.97	1.95	1.76	1.71	2.10	1.13
0.80	1.99	1.94	1.97	1.86	1.85	1.92	1.85	2.13	2.08	2.19	1.90
0.99	1.74	1.88	2.70	2.26	2.28	1.90	2.47	1.67	1.73	1.56	2.30

Table 5.2: Lazy Update Policy Forwarding Cost Per Invocation.

Update Message Cost Per Operation											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.80	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.99	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.3: Lazy Update Policy Update Cost Per Operation.

Forwarding Message Cost Per Operation											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.10	0.10	0.10	0.09	0.07	0.09	0.07	0.11	0.03	0.15	0.18
0.20	1.01	0.95	0.95	0.87	0.92	1.00	1.03	0.87	0.99	0.79	0.59
0.40	0.97	1.00	1.00	1.04	1.03	0.94	0.96	1.01	1.06	0.98	1.00
0.60	0.72	0.71	0.76	0.66	0.65	0.75	0.74	0.68	0.65	0.82	0.42
0.80	0.36	0.35	0.38	0.33	0.35	0.36	0.37	0.38	0.41	0.44	0.33
0.99	0.01	0.02	0.02	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.02

Table 5.4: Lazy Update Policy Forwarding Cost Per Operation.

Total Cost With 100% Update Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.10	0.10	0.10	0.09	0.07	0.09	0.07	0.11	0.03	0.15	0.18
0.20	1.01	0.95	0.95	0.87	0.92	1.00	1.03	0.87	0.99	0.79	0.59
0.40	0.97	1.00	1.00	1.04	1.03	0.94	0.96	1.01	1.06	0.98	1.00
0.60	0.72	0.71	0.76	0.66	0.65	0.75	0.74	0.68	0.65	0.82	0.42
0.80	0.36	0.35	0.38	0.33	0.35	0.36	0.37	0.38	0.41	0.44	0.33
0.99	0.01	0.02	0.02	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.02

Table 5.5: Lazy Update Policy Total Cost.

level is between 0.2 and 0.6. There are two factors which can affect the amount of forwarding message sending: the frequency of object migration and the frequency of object invocation. The corresponding graphs of the above table results are shown in Figure 5.1.

Activity is defined as the ratio of object moving times over the sum of object moving times and object invocation times. When the activity is very low, there is less object moving, So no long forwarding address chains are formed. Even if there are many invocations, there are few forwarding messages. When activity is very high, although many long forwarding address chains are formed, the forwarding messages are fewer because there are many fewer invocations. Only when the activity level is moderate (between 0.2 to 0.6), can there be many forwarding messages generated.

Theoretically, locality should not affect the lazy update policy because there is no update message sending. Strong locality will increase the utilization of the update message because the next object invoked will very likely be the same as the previous one. The new object's location contained in the last update message can be used by subsequent invocations, and no forwarding address chain will be required.

In reality, there is an implicit updating under the lazy update policy. When an object moves to a destination which is located on its forwarding address chain, the chain will be cut as if an update message were received. From the experiment results we can see that increasing locality has almost no effect on system performance.

5.3 Experiment Result with Urgent Update Policies

There are two factors which will decide how urgent an urgent update policy is:

1. Object dependent set — the set of machines which have sent invocations to the object. The larger the set is, the more update messages an object will send when it migrates. Each invoked object has a dependent set.

2. Dependent set moving hops. The longer the set is maintained by objects when they move, the more update messages an object will send when it migrates.

Generally speaking, the more update messages, the fewer forwarding messages. The question is how large the dependent set should be and how far a dependent set should be maintained to reach an optimum combination of forwarding messages and update messages.

By adjusting the dependent set and the hop number, we try to find an update policy which will optimize the performance of OFP. The following are the experimental results from three different versions of the urgent update policy:

- Urgent update policy version 1: In this policy, the dependent sets are empty. The number of hops that sets are maintained by objects is 0. The experiment results based on this

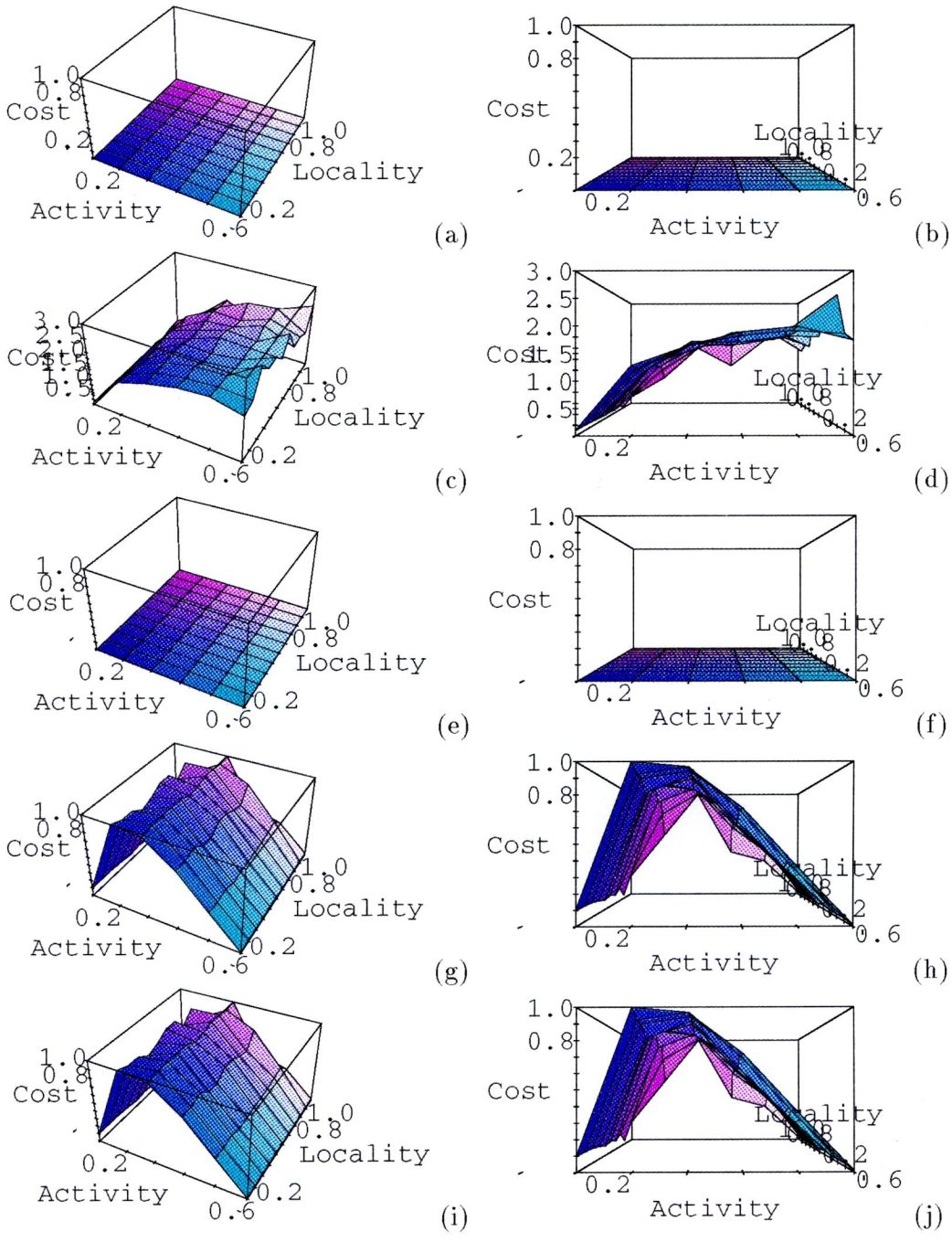


Figure 5.1: Lazy Update Policy. (a)Updating Cost Per Migration. (b)Front View of (a). (c)Forwarding Cost Per Invocation. (d)Front View of (c). (e)Updating Cost Per Operation. (f)Front View of (e). (g)Forwarding Cost Per Operation. (h)Front View of (g). (i)Total Cost Per Operation. (j)Front View of (i).

Updating Cost Per Migration											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	5.26	4.81	3.85	3.71	4.22	2.48	2.31	2.13	1.33	0.52	0.12
0.20	1.96	1.89	1.72	1.58	1.36	1.17	0.91	0.78	0.58	0.33	0.09
0.40	1.02	0.98	0.85	0.78	0.69	0.61	0.45	0.39	0.26	0.19	0.08
0.60	0.54	0.51	0.46	0.43	0.35	0.33	0.28	0.23	0.19	0.14	0.06
0.80	0.24	0.22	0.20	0.18	0.18	0.16	0.13	0.12	0.11	0.08	0.06
0.99	0.01	0.01	0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

Table 5.6: Urgent Update Policy Version 1 Update Cost Per Migration.

Forwarding Cost Per Invocation											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.06	0.07	0.09	0.07	0.08	0.05	0.08	0.12	0.11	0.09	0.00
0.20	0.80	0.77	0.84	0.84	0.85	0.98	0.87	0.99	0.90	1.14	0.09
0.40	1.19	1.24	1.21	1.23	1.27	1.18	1.17	1.42	1.15	1.15	0.05
0.60	1.49	1.38	1.50	1.51	1.45	1.62	1.35	1.09	1.16	0.92	0.25
0.80	1.87	1.86	1.56	1.69	1.53	1.66	1.59	1.38	1.15	1.10	0.37
0.99	2.73	1.95	2.00	2.44	1.91	2.05	1.25	1.42	1.95	1.64	1.85

Table 5.7: Urgent Update Policy Version 1 Forwarding Cost Per Invocation.

update policy are shown in Table 5.6, Table 5.7, Table 5.8, Table 5.9 and Table 5.10. Its corresponding graphs are shown in Figure 5.2.

Updating Cost Per Operation											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.05	0.07	0.05	0.04	0.05	0.03	0.03	0.03	0.01	0.01	0.00
0.20	0.41	0.39	0.36	0.34	0.28	0.26	0.19	0.17	0.12	0.07	0.02
0.40	0.41	0.40	0.35	0.31	0.28	0.24	0.18	0.16	0.11	0.08	0.03
0.60	0.33	0.31	0.29	0.26	0.21	0.20	0.17	0.14	0.12	0.09	0.04
0.80	0.20	0.18	0.16	0.15	0.15	0.13	0.11	0.09	0.09	0.07	0.05
0.99	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

Table 5.8: Urgent Update Policy Version 1 Update Cost Per Operation.

Forwarding Cost Per Operation											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.05	0.07	0.09	0.07	0.08	0.05	0.08	0.12	0.11	0.09	0.00
0.20	0.63	0.61	0.66	0.66	0.67	0.76	0.69	0.78	0.71	0.90	0.07
0.40	0.72	0.74	0.71	0.74	0.76	0.72	0.69	0.84	0.67	0.67	0.03
0.60	0.57	0.55	0.57	0.60	0.58	0.62	0.52	0.42	0.45	0.35	0.09
0.80	0.35	0.33	0.29	0.31	0.28	0.30	0.29	0.26	0.23	0.19	0.07
0.99	0.02	0.02	0.01	0.02	0.02	0.02	0.01	0.02	0.02	0.02	0.02

Table 5.9: Urgent Update Policy Version 1 Forwarding Cost Per Operation.

Total Cost With 100% Update Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.11	0.14	0.14	0.11	0.13	0.08	0.11	0.15	0.13	0.09	0.00
0.20	1.04	1.00	1.02	1.00	0.95	1.02	0.88	0.94	0.84	0.97	0.09
0.40	1.12	1.13	1.07	1.05	1.04	0.96	0.88	1.00	0.78	0.75	0.06
0.60	0.90	0.85	0.86	0.86	0.79	0.82	0.69	0.57	0.57	0.44	0.13
0.80	0.54	0.51	0.45	0.46	0.43	0.44	0.40	0.35	0.31	0.25	0.12
0.99	0.03	0.03	0.02	0.03	0.03	0.03	0.01	0.03	0.03	0.02	0.03

Table 5.10: Urgent Update Policy Version 1 Total Cost.

Total Message Cost With 100% Update Cost						
	Locality					
Activity	0.0	0.2	0.4	0.6	0.8	1.0
0.1	0.10	0.10	0.09	0.11	0.07	0.00
0.2	1.20	1.08	1.11	0.94	0.88	0.03
0.4	1.41	1.27	1.71	1.15	0.80	0.13
0.6	1.19	1.06	0.97	0.74	0.55	0.07
0.8	0.71	0.63	0.56	0.39	0.33	0.07
0.99	0.04	0.03	0.04	0.04	0.03	0.03

Table 5.11: Urgent Update Policy Version 2 Total Cost.

Total Message Cost With 100% Update Cost						
	Locality					
Activity	0.0	0.2	0.4	0.6	0.8	1.0
0.1	0.14	0.13	0.16	0.07	0.12	0.41
0.2	1.25	1.13	1.18	1.06	0.91	0.06
0.4	1.51	1.40	1.32	1.17	0.98	0.06
0.6	1.34	1.25	1.03	0.94	0.71	0.07
0.8	0.82	0.70	0.57	0.44	0.33	0.08
0.99	0.03	0.05	0.03	0.05	0.04	0.03

Table 5.12: Urgent Update Policy Version 3 Total Cost.

- Urgent update policy version 2: In this policy, the dependent set contains the dependents accumulated on the last node. The number of hops that the sets are maintained by objects is 1. The experiment results based on this policy are shown in Table 5.11.
- Urgent update policy version 3: In this policy, the dependent set contains the dependents accumulated on the last two nodes. The number of hops that sets are maintained by objects is 2. The experiment results based on this policy are shown in Table 5.12.

By analyzing the above results, we can see the following phenomena:

1. From Table 5.10, Table 5.11 and Table 5.12 we can see that in general version 1 update policy has a better performance than version 2 and version 3 update policies. Version 2 policy has a better performance than version 3 update policy. This is true especially when the locality is between 0 to 0.8 and activity is between 0.2 to 0.8.
2. For each individual urgent update policy, as the locality increases, the performance becomes better especially for activity from 0.2 to 0.8.

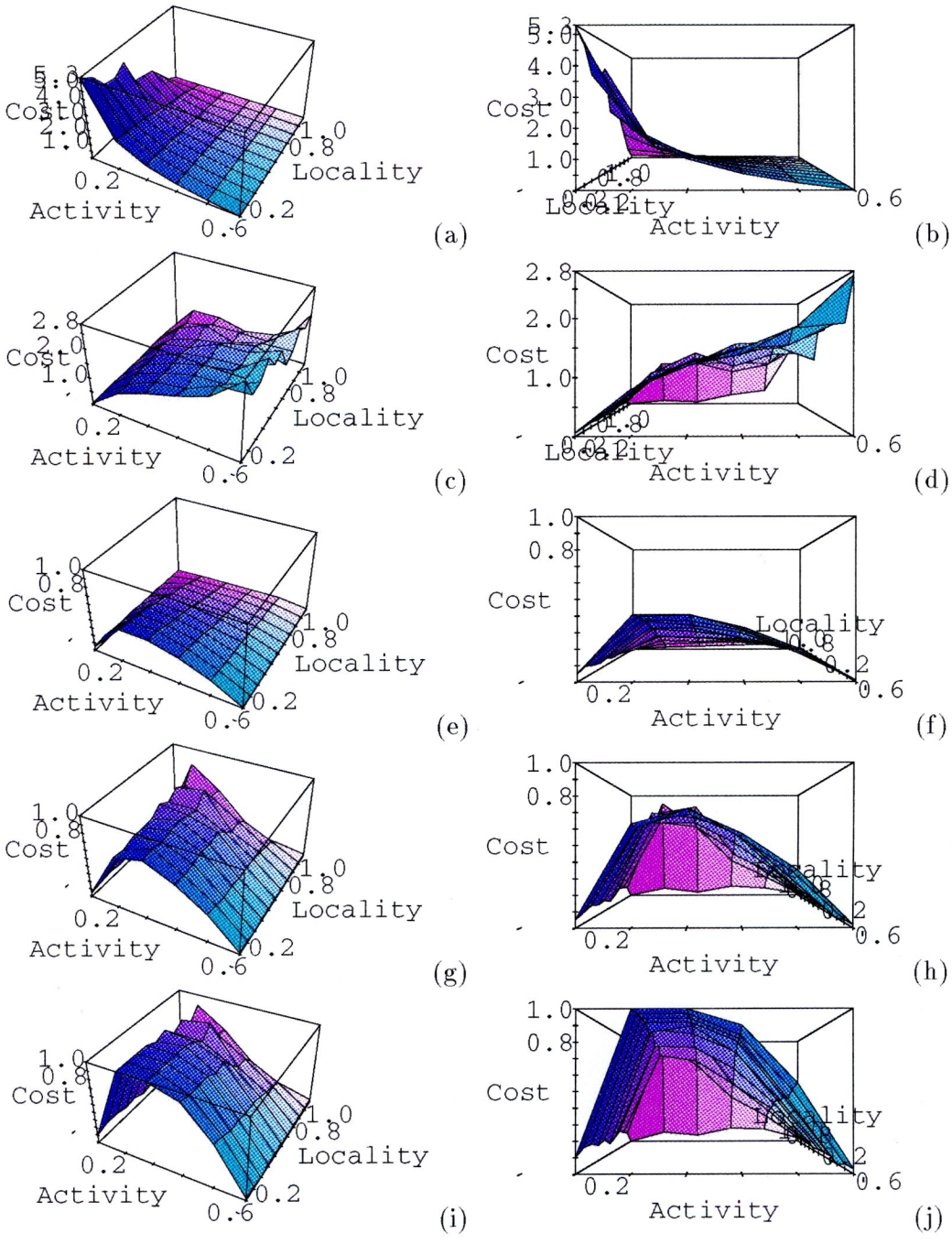


Figure 5.2: Urgent Update Policy. (a)Updating Cost Per Migration. (b)Front View of (a). (c)Forwarding Cost Per Invocation. (d)Front View of (c). (e)Updating Cost Per Operation. (f)Front View of (e). (g)Forwarding Cost Per Operation. (h)Front View of (g). (i)Total Cost Per Operation. (j)Front View of (i).

The above phenomena can be explained as follows:

- For each individual urgent update policy, when the locality increases, object location information contained in one update message can be efficiently used by the subsequent object invocations which invoke the same object. Thus, the forwarding message volume is reduced. Also, because of higher locality, the dependent set becomes smaller so the update message volume is also reduced. The result is that the OFP performance is improved.

- Comparing version 1 urgent update policy with version 2 and version 3 update policy, we found that with the same locality and activity, the number of forwarding messages in version 1 is almost the same as that in version 2 and version 3. But the update message numbers in version 2 and version 3 are more than that in version 1. This is because of the dependent set carried by each moving object. The larger the dependent set and the further it is carried, the more update messages that are generated. The issue is whether these update messages can be used efficiently by subsequent invocations so that forwarding messages can be reduced. The reality is no. It turns out that version 2 and version 3 update policies are more expensive than version 1 update policy.

Because, in most cases, the version 1 urgent update policy is better than version 2 and version 3, we decided to use version 1 urgent update policy as a representative to compare with the lazy update policy and to discuss the new adaptive update policy.

In the above discussion, we assume update messages are sent in the foreground. In a real system, it can be done partially or fully in the background. If we assume that 0%, 20%, 50%, or 80% of update message sending can be done in foreground, we obtain the following performance results, which are better than those in Table 5.10. These results are calculated based on the experimental results which assume update message sending is done 100% in foreground. The results in Table 5.10 are calculated as follows: $Totalcost = updatingcost * 100\% + forwardingcost$. If we assume that 50% of updating is done in background, then the performance is calculated as follows: $Totalcost = updatingcost * 50\% + forwardingcost$.

Here, we assume that background updating costs nothing. This assumption is based on the fact that in a real system object invocation is relatively infrequent, so in general background updating is fast enough to update an object location before the next object invocation. In our experiments, the object invocation is done as quickly as possible.

The results discussed above are shown in Tables 5.13 to 5.16 and their corresponding graphs are shown in Figure 5.3.

Total Message Cost With 0% Update Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.05	0.07	0.09	0.07	0.08	0.05	0.08	0.12	0.11	0.09	0.00
0.20	0.63	0.61	0.66	0.66	0.67	0.76	0.69	0.78	0.71	0.90	0.07
0.40	0.72	0.74	0.71	0.74	0.76	0.72	0.69	0.84	0.67	0.67	0.03
0.60	0.57	0.55	0.57	0.60	0.58	0.62	0.52	0.42	0.45	0.35	0.09
0.80	0.35	0.33	0.29	0.31	0.28	0.30	0.29	0.26	0.23	0.19	0.07
0.99	0.02	0.02	0.00	0.02	0.02	0.02	0.01	0.02	0.02	0.02	0.02

Table 5.13: Urgent Update Policy Version 1 Total Cost.

Total Message Cost With 20% Update Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.06	0.08	0.10	0.08	0.09	0.06	0.09	0.13	0.11	0.09	0.00
0.20	0.71	0.69	0.73	0.73	0.73	0.81	0.73	0.81	0.73	0.91	0.07
0.40	0.80	0.82	0.78	0.80	0.82	0.77	0.73	0.87	0.69	0.69	0.04
0.60	0.64	0.61	0.63	0.65	0.62	0.66	0.55	0.45	0.47	0.37	0.10
0.80	0.39	0.37	0.32	0.34	0.31	0.33	0.31	0.28	0.25	0.20	0.08
0.99	0.02	0.02	0.00	0.02	0.02	0.02	0.01	0.02	0.02	0.02	0.02

Table 5.14: Urgent Update Policy Version 1 Total Cost.

Total Message Cost With 50% Update Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.08	0.11	0.11	0.09	0.11	0.07	0.10	0.14	0.12	0.10	0.00
0.20	0.83	0.80	0.84	0.83	0.81	0.89	0.78	0.86	0.77	0.94	0.08
0.40	0.92	0.94	0.89	0.90	0.90	0.84	0.78	0.92	0.73	0.71	0.04
0.60	0.73	0.71	0.71	0.73	0.68	0.72	0.60	0.49	0.51	0.39	0.11
0.80	0.45	0.42	0.37	0.39	0.36	0.36	0.34	0.30	0.28	0.23	0.10
0.99	0.03	0.03	0.00	0.03	0.03	0.03	0.01	0.03	0.03	0.03	0.03

Table 5.15: Urgent Update Policy Version 1 Total Cost.

Total Message Cost With 80% Update Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.09	0.13	0.13	0.10	0.12	0.07	0.10	0.14	0.12	0.10	0.00
0.20	0.96	0.92	0.95	0.93	0.89	0.97	0.84	0.92	0.81	0.96	0.09
0.40	1.05	1.06	0.99	0.99	0.98	0.91	0.83	0.97	0.76	0.73	0.05
0.60	0.83	0.80	0.80	0.81	0.75	0.78	0.66	0.53	0.55	0.42	0.12
0.80	0.51	0.47	0.42	0.43	0.40	0.40	0.38	0.33	0.30	0.25	0.11
0.99	0.03	0.03	0.00	0.03	0.03	0.03	0.02	0.03	0.03	0.03	0.03

Table 5.16: Urgent Update Policy Version 1 Total Cost.

5.4 Adaptive Update Policy

Our results showed that for locality smaller than 0.5, the lazy update policy had a result similar to the urgent update policy. For locality larger than 0.5, the urgent version 1 update policy had a better result.

At first, we thought that the lazy update policy could easily build a long forwarding address chain which would make subsequent invocations expensive. But in reality, when an object moves back to any node on the forwarding address chain, the chain is shortened. When the activity level is around 0.5, there are enough invocations to help build the forwarding address chains, but there is also enough migration to help shorten forwarding address chains. In other words, the lazy update policy is not really lazy. Object migration includes an implicit update for the destination node. For the urgent update policy, when an object moves, the system does not have to send an update message to every node on the local network. By limiting the dependent set size and number of carrying hops, we got three different versions of urgent update policies. We found the lazier one (version 1) had the best performance. The urgent update policy is not really completely urgent.

Based on these two different update policies, we derived a new update policy based on the above analysis. The new update policy is called the adaptive update policy; it is a combination of urgent and lazy updating. This policy uses a decision table to decide when to use an urgent update and when to use a lazy update. The decision table is built by directly comparing the results of urgent update and lazy update.

We used the results from lazy update and urgent update with 100% and 50% background updating as examples to generate decision tables for our adaptive update policy. First we calculate the difference between urgent policy from lazy policy (Table 5.17 and Table 5.19). Based on these difference, we get the decision tables shown in Tables 5.18 and 5.20.

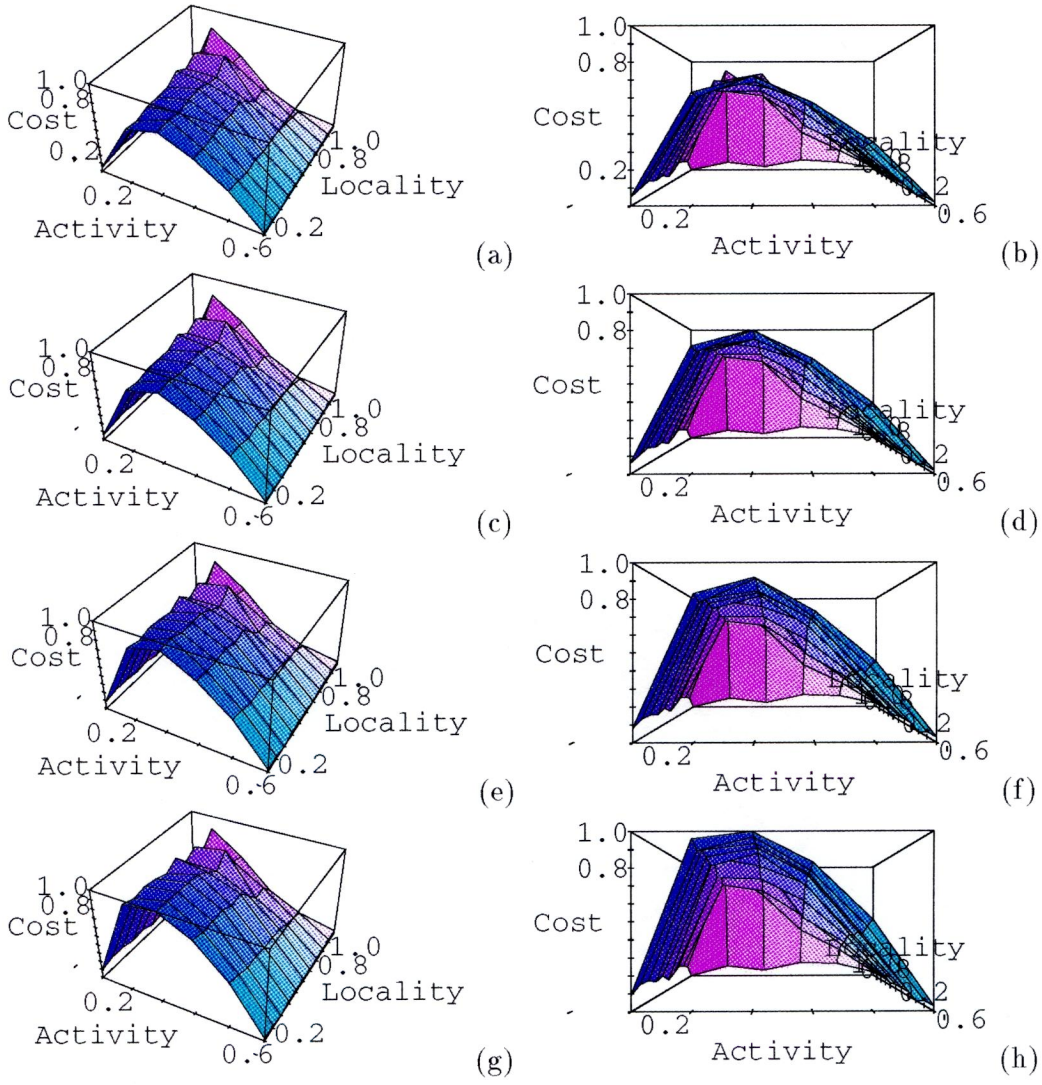


Figure 5.3: Urgent Update Policy. (a)Total Cost with 0% of Updating Cost. (b)Front View of (a). (c)Total Cost with 20% of Updating Cost. (d)Front View of (c). (e)Total Cost with 50% of Updating Cost. (f)Front View of (e). (g)Total Cost with 80% of Updating Cost. (h)Front View of (g).

Difference Table With 100% Updating											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.00	-0.04	-0.04	-0.02	-0.06	0.01	-0.04	-0.04	-0.09	0.05	0.18
0.20	-0.03	-0.05	-0.07	-0.13	-0.03	-0.02	0.15	-0.08	0.16	-0.18	0.50
0.40	-0.16	-0.14	-0.06	-0.01	-0.01	-0.02	0.09	0.01	0.28	0.23	0.94
0.60	-0.18	-0.15	-0.10	-0.20	-0.14	-0.07	0.05	0.12	0.08	0.38	0.29
0.80	-0.19	-0.16	-0.07	-0.13	-0.08	-0.07	-0.03	0.03	0.09	0.18	0.21
0.99	-0.02	-0.01	0.02	-0.01	0.00	-0.01	0.00	-0.01	-0.01	-0.01	-0.01

Table 5.17: The Difference Of Lazy Update Policy Performance and Urgent Update Policy Version 1 Performance.

Decision Table With 100% Updating											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	-	*	*	-	*	-	*	*	*	+	+
0.20	-	*	*	*	-	-	+	*	+	*	+
0.40	*	*	*	-	-	-	+	-	+	+	+
0.60	*	*	*	*	*	*	+	+	+	+	+
0.80	*	*	*	*	*	*	-	-	+	+	+
0.99	-	-	-	-	-	-	-	-	-	-	-

Table 5.18: The Decision Table of Adaptive Update Policy: * – Lazy, + – Urgent, - – Both

In the decision tables, + represents the urgent update policy; * represents the lazy update policy and - represents either policy. The decision tables are calculated by using a threshold value 0.03. Any absolute difference which is less than or equal to the threshold is replaced by -. A negative difference whose absolute value is larger than the threshold is replaced by *. A positive difference whose absolute value is larger than the threshold is replaced by +.

The performance of the adaptive update policy based on the above calculations is shown in Tables 5.21 and 5.22. The corresponding three dimensional graph is shown in Figure 5.4. By comparing the results with those of the lazy and urgent update policies, we concluded that the adaptive update policy had a better performance than both the urgent and the lazy update policies.

Difference Table With 50% Updating											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.02	-0.01	-0.01	0.00	-0.04	0.02	-0.03	-0.03	-0.09	0.05	0.18
0.20	0.18	0.15	0.11	0.04	0.11	0.11	0.25	0.01	0.22	-0.15	0.51
0.40	0.05	0.06	0.11	0.14	0.13	0.10	0.18	0.09	0.33	0.27	0.96
0.60	-0.01	0.00	0.05	-0.07	-0.03	0.03	0.14	0.19	0.14	0.43	0.31
0.80	-0.09	-0.07	0.01	-0.06	-0.01	0.00	0.03	0.08	0.13	0.21	0.23
0.99	-0.02	-0.01	0.02	-0.01	0.00	-0.01	0.01	-0.01	-0.01	-0.01	-0.01

Table 5.19: The Difference Of Lazy Update Policy Performance and Urgent Update Policy Version 1 Performance.

Decision Table With 50% Updating											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	-	-	-	-	*	-	-	-	*	+	+
0.20	+	+	+	+	+	+	+	-	+	*	+
0.40	+	+	+	+	+	+	+	+	+	+	+
0.60	-	-	+	*	-	-	+	+	+	+	+
0.80	*	*	-	*	-	-	-	+	+	+	+
0.99	-	-	-	-	-	-	-	-	-	-	-

Table 5.20: The Decision Table of Adaptive Update Policy: * – Lazy, + – Urgent, - – Both

Performance With 100% Updating Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.10	0.10	0.10	0.09	0.07	0.08	0.07	0.11	0.03	0.10	0.00
0.20	1.01	0.95	0.95	0.87	0.92	1.00	0.88	0.87	0.83	0.79	0.09
0.40	0.97	1.00	1.00	1.04	1.03	0.94	0.87	1.00	0.78	0.75	0.06
0.60	0.72	0.71	0.76	0.66	0.65	0.75	0.69	0.56	0.57	0.44	0.13
0.80	0.36	0.35	0.38	0.33	0.35	0.36	0.37	0.35	0.32	0.26	0.12
0.99	0.01	0.02	0.00	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.02

Table 5.21: The Adaptive Update Policy Total Cost

Performance With 50% Updating Cost											
	Locality										
Activity	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.01	0.08	0.10	0.10	0.09	0.07	0.07	0.07	0.11	0.03	0.10	0.00
0.20	0.83	0.80	0.84	0.83	0.81	0.89	0.78	0.86	0.77	0.79	0.08
0.40	0.92	0.94	0.89	0.90	0.90	0.84	0.78	0.92	0.73	0.71	0.04
0.60	0.72	0.71	0.71	0.66	0.65	0.72	0.60	0.49	0.51	0.39	0.11
0.80	0.36	0.35	0.37	0.33	0.35	0.36	0.34	0.30	0.28	0.23	0.10
0.99	0.01	0.02	0.00	0.02	0.03	0.02	0.01	0.02	0.02	0.02	0.02

Table 5.22: The Adaptive Update Policy Total Cost

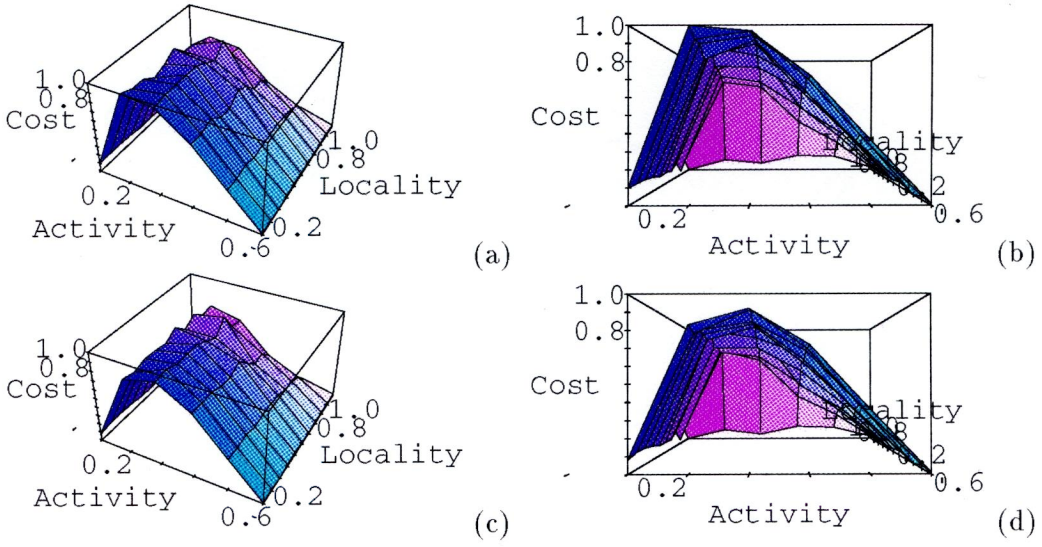


Figure 5.4: Adaptive Update Performance Graphs (a) Performance with 100% Updating. (b) Front View of (a). (c) Performance with 50% Updating. (d) Front View of (c).

Chapter 6

Conclusions and Future Work

6.1 Contribution

An extensive survey has been done on object and process migration. In the survey, we focus on the methods used to track mobile objects or processes. Several systems have been discussed. The purpose of the survey is to find the limitations of the methods and their potential improvements. Most of them have an ad hoc limitation. In our discussion, we assume that there is no limitation on object migration. Emerald is one such system in which any object can be migrated without limitations. Based on this assumption, we find that the methods used in the surveyed systems are no longer what we expected.

Using forwarding addresses to track mobile objects is not new. In [Fow85][Jul88], the authors have discussed this topic in depth. In [Fow85], more theoretical analysis has been done on different algorithms used to shorten the forwarding address chains formed by forwarding messages. In [Jul88], a simple algorithm has been used to track mobile objects. In this algorithm, reply messages to an invocation are used to piggyback the new location of the invoked object. The method used to cut forwarding address chains is called update policy.

None of the algorithms mentioned above has taken into consideration an individual object's behavior. One of the contributions of this thesis is to link together the object behavior with the update policy. This idea comes from the real world. For example, a tourist normally does not have to send mail frequently to his friends and relatives about his location. The reason is because he moves frequently and if he did that, it would cost him a lot of money. The policy used here is called lazy policy. But for a family, when they move to a new place, letters should be sent to tell their friends and relatives the new location. It will not cost too much because a family is more stable than a tourist. The policy used here is called urgent policy.

The concept of activity has been defined to describe object behavior. In a distributed sys-

tem, we believe objects with different activities should use different update policies to optimize system performance. Different update policies have been compared. A new update policy has been defined based on results from the lazy update policy and the urgent update policy. This new policy is called an adaptive update policy.

In the thesis, the lazy update policy and urgent update policy have been studied and tested. For urgent update policy, there are two variables. One is the dependent set size and the other is the number of hops the dependent set will be maintained by an moving object. By adjusting the two variables, three versions of urgent update policies are produced. Through experiments, we found that the lazier urgent update policy (version 1) has the best performance.

The update message sending can be done partly or wholely in the background, whereas forwarding message sending has to be done entirely in the foreground. By assuming that 0%, 20%, 50% or 80% of updating is done in the foreground, a better performance of urgent update policy can be obtained. This assumption only affects the urgent update policy, since the lazy update policy does not send update messages.

Locality has been defined and used to model object invocation in our experiments. Locality exists in reality. One purpose of our experiments was to examine the effects of the locality object invocations on the update policies. We found that for the urgent update policy, OFP has better performance with higher locality.

Another contribution of the thesis is to define and implement two layered protocols for transparent object tracking. The implementation includes 3500 lines of C source code. RDP is mainly used to support packet data transmission. It is similar to UDP, but with an extra time out and retransmission mechanism to guarantee reliable packet delivery. Its function is to send a data packet to a given machine. Object tracking is the responsibility of the OFP protocol layer, which maintains distributed mapping tables. Through the distributed mapping tables, OFP can map an object id to its machine address. OFP is built on RDP. The function of OFP is to send a message to a given object.

For any update policy in OFP, one should remember that there is always a tradeoff between the update message sending and the forwarding message sending. More update messages will help reduce the number of forwarding messages. Conversely, less update messages will increase the sending of forwarding messages. One of the purposes of our experiments was to find an optimum combination which can make the total of updating and forwarding approach an optimum balance. Adaptive update policy is our solution to this problem.

6.2 Future Work

Currently, our system only works on a local area network where broadcast is possible and efficient. To extend the system into an internet environment, we need to provide a local server on each local network. This server can be located on a router. When OFP cannot locate an object on the local area network, it will seek help from its local server. The local server will continue the object search on the internet. To make local servers more efficient, they could keep records on each object moving in or out of the local network.

Each local server maintains a mapping table which is the same as the one on each node. If an object location cannot be resolved by its local server, a further searching request will be forwarded to the other remote servers. Each remote server will check its local network and reply with a confirm message.

Another interesting direction is to experiment with update policies, especially the adaptive update policy, on objects with different activities and different locality and to further prove that the adaptive update policy can still reach optimum performance.

The next interesting issue is to integrate the adaptive update policy into a real system and measure the behaviors of real objects.

Bibliography

[Aea89] Yeshayahu Artsy and et al. Designing a process migration facility–the charlotte experience. *IEEE Computer*, pages 47–56, Sep 1989.

[And91] Gregory R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23:49–90, Mar 1991.

[BBea91] R. Balter, J. Bernadat, and et al. Architecture and implementation of guide, an object-oriented distributed system. *UNIX Computing Systems*, 4:31–67, 1991.

[BHJ+86] Andrew Black, Norman Hutchinson, Eric. Jul, Henry Levy, and Larry Carter. Distribution and abstarct types in emerald. TR 86-02-04, University of Washington, Jul 1986.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *OOPSLA '86 Proceedings*, page 78, Sep 1986.

[BL90] Joseph Boykin and Alan Langerman. Mach/4.3bsd: A conservative approach. *UNIX Computing Systems*, 3:69, 1990.

[Bla85] Andrew P. Black. Supporting distributed applications. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193. Association for Computing Machinery, 1985.

[Boo86] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, 12:211–221, Feb 1986.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21:261–321, Sep 1989.

[Car85] Luca Cardelli. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17:471–522, Dec 1985.

[CC91] Roger S. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23:91–124, Mar 1991.

[CD88] George F. Coulouris and Jean Dollimore. *Distributed Systems Concepts and Design.* Addison-Wesley Publishing Company, 1988.

[CDEG90] George A. Champine and Jr. Daniel E. Geer. Project athena as a distributed computer system. *IEEE Computer*, pages 40–51, Sep 1990.

[Cea90] L. Csaba and et al., editors. *Computer Networking.* North-Holland, 1990.

[CG90] M. Cosnard and C. Girault, editors. *Decentralized Systems.* North-Holland, 1990.

[Che85] David R. Cheriton. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems*, 3:77–107, May 1985.

[Che88] David R. Cheriton. The v distributed system. *Communications of the ACM*, 31:314, Mar 1988.

[Cla89] David D. Clark. An analysis of tcp processing overhead. *IEEE Communications Magazine*, pages 23–29, Jun 1989.

[CM89] David Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, 7:147–183, May 1989.

[CW89] David R. Cheriton and Carey L. Willaimson. Vmtp as the transport layer for high-performance distributed systems. *IEEE Communication Magazine*, pages 37–44, Jun 1989.

[DCea90] P. Dasgupta, R. C. Chen, and et al. The design and implementation of the clouds distributed operating system. *UNIX Computing Systems*, 3:11, 1990.

[DO91] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software–Practice and Experience*, 21(8):757–785, Aug 1991.

[DPH91] Peter Druschel, Larry L. Peterson, and Norman C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in lipto. TR 91-6A, The University of Arizona, Dec 1991.

[Esk90] M. Rasit Eskicioglu. Process migration in distributed systems: A comparative survey. TR 90-3, University of Alberta, Jan 1990.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:28–33, Aug 1991.

[Fow85] Robert Joseph Fowler. Decentralized object finding using forwarding addresses. TR 85-12-1, University of Washington, Dec 1985.

[Gos91] A. Goscinski. *Distributed Operating Systems–The Logic Design.* Addison-Wesley Publishing Company, 1991.

[GS90] Bezalel Gavish and Olivia R. Liu Sheng. Dynamic file migration in distributed computer systems. *Communications of the ACM*, 33:2, 1990.

[HMPT89] Norman C. Hutchinson, Shivakant Mishra, Larry L. Peterson, and Vicraj T. Thomas. Tools for implementing network protocols. *Software–Practice and Experience*, 19(9):895–916, Sept 1989.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17:64–76, Jan 1991.

[Hut87] Norman C. Hutchinson. Emerald: An object-based language for distributed programming. TR 87-01-01, University of Washington, Jan 1987.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation and Modeling.* John Wiley & Sons, Inc., 1991.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6:109–133, Feb 1988.

[Jul88] Eric Jul. Object mobility in a distributed object-oriented system. TR 88-12-06, University of Washington, Dec 1988.

[KP91] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, 9:364–373, Nov 1991.

[KW89] Stephen G. Kochen and Patrick H. Wood, editors. *UNIX Networking.* Hayden Books, 1989.

[Lis88] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31:300, Mar 1988.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International, 1988.

[MvRT+88] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba a distributed operating system for the 1990s. *Communications of the ACM*, 31, Mar 1988.

[NBL+88] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Communications of the ACM*, 31:258, Mar 1988.

[Nic87] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, Nov 1987.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the sprite network file system. *ACM Transaction on Computer Systems*, 6:134–154, Feb 1988.

[OD83] Derek C. Oppen and Yogen K. Dalal. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*, 1:230–253, Jul 1983.

[PHOR90] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao. The x-kernel: A platform for accessing internet resources. *IEEE Computer*, 23:23–33, May 1990.

[PM83] M. L. Powell and B. P. Miller. Process migration in demos/mp. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 1983.

[PW85] Gerald Popek and Bruce J. Walker, editors. *The LOCUS Distributed System Architecture.* The MIT Press, 1985.

[RC86] K. Ravindran and Samuel T. Chanson. Process alias-based structruring techniques for distributed computing systems. In *IEEE Sixth International Conference on Distributed Computing Systems.* IEEE, 1986.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software–Practice and Experience*, 21(1):91–118, Jan 1991.

[Rus89] D. Russell. *The Principles of Computer Networking.* Cambridge University Press, 1989.

[Smi88] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review*, 22(3):28–40, Jul 1988.

[Sta84] William Stallings. Local networks. *Computer Surveys*, 16:3–41, Mar 1984.

[Str86] Rob Strom. A comparison of the object-oriented and process paradigms. *SIGPLAN Notices*, 21:88–97, Oct 1986.

[Str88] Bjarne Stroustrup. What is object-oriented programming. *IEEE Software*, pages 10–20, May 1988.

[SW87] Sol M. Shatz and Jia-Ping Wang. Introduction to distributed-software engineering. *IEEE Computer*, pages 23–31, Oct 1987.

[Tan89] Andrew S. Tanenbaum. *Computer NetWorks.* Prentice Hall, 1989.

[Tic85] Walter F. Tichy. Rcs–a system for version control. *Software–Practice and Experience*, 15(7):637–654, Jul 1985.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles.* ACM, 1985.

[TR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *Computing Serveys*, 17:419–468, Dec 1985.

[Wat90] David A. Watt. *Programming Language Concepts and Paradigms.* Prentice Hall International, 1990.

[WPM90] Anthony I. Wasserman, Peter A. Pircher, and Robert J. Muller. The object-oriented structured design notation for software design representation. *IEEE Computer*, pages 50–63, March 1990.

[Zay87] Edward R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles.* ACM, Dec 1987.