Object Mobility
in a
Distributed Object-Oriented System

by

ERIC JUL

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1989

Approved by ______________________________
(Chairperson of Supervisory Committee)

____

____

____

Program Authorized
to Offer Degree ______________________________

Date ______________________________

Signature ______________________________

Date ______________________________

University of Washington

Abstract

OBJECT MOBILITY
IN A
DISTRIBUTED OBJECT-ORIENTED SYSTEM

by Eric Jul

Chairperson of the Supervisory Committee: Professor Henry M. Levy
Department of Computer Science

This dissertation studies the movement of objects both large and small within a locally distributed system. We propose that object mobility at a fine granularity can be a powerful tool for simplifying the construction of distributed applications. The unit of mobility in previous systems has typically been an address space. We propose using data of any size as the unit of mobility. Our study of fine-grained mobility is based on a new programming language called Emerald for which we have implemented a run-time kernel. Emerald supports a single model of computation: the object. All objects are described using a uniform semantic model. They can move freely within the system to take advantage of distribution and respond to dynamically changing requirements. We say that Emerald has fine-grained mobility because *all* objects can move regardless of size. Process migration is supported; light-weight processes that are executing within a migrating object are also migrated on-the-fly.

Our design of Emerald emphasizes efficiency, in particular, efficient communication between potentially distributed objects located on the same machine. By integrating distribution concepts into the programming language, the compiler is able to generate more efficient code for objects that do not require full generality. Emerald supports call-by-reference semantics—even for remote procedure calls. We introduce a special parameter passing mode, *call-by-move* which can be used to move parameters for efficiency reasons.

We present a design for a distributed on-the-fly garbage collector. The design includes a novel faulting mechanism that allows user processes to proceed during collection.

This dissertation discusses aspects of language design related to mobility and the design and implementation of an efficient Emerald kernel. Measurements of the system and of several small applications are presented.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I am grateful to Hank Levy for serving as my advisor—it has been a great experience. I wish to thank him for the many hours that we have spent together not only working on Emerald but skiing, eating, walking down the Ave, buying cookies, and many other enjoyable undertakings.

A special thanks goes to Norm Hutchinson for his warm and sincere friendship and for his immensely positive attitude toward life and family. I value his ability to listen to other people's worries; problems always seem to dwindle as soon as Norm whips around on his chair and says "So, tell me what the problem is."

Thanks to Ed Lazowska for his accurate critique of drafts of this thesis, for his support throughout the past six years, and for his concern for the health of my family.

I wish to thank all the students, faculty, coaches, and staff in the UW Deptartment of Computer Science—it has been a privilege to work in their midst.

I thank my sister, Susanne, for the long hours she put in editing and proof-reading this thesis. I thank my other sister, Elisabeth, for baby-sitting so that I could spend more time thesis writing.

My work has been supported in part by the National Science Foundation under grant no. MCS-8004111, grant no. DCR-8420945, and grant no. CCR-8700186, by the Danish Natural Science Research Council under grant no. FTU 5.17.5.1.35, by the University of Copenhagen under grant J.nr. 574-2,2, by a dissertation award from MicroSoft, and by an award from GE. I gratefully acknowledge this support.

A special thanks to Jack McCredie and the Digital Equipment Corporation for a very timely External Research Grant that was crucial in getting the Emerald project going.

I am grateful to my wife, Bit, and my children, Claus, Anne, and David, who all suffered to make this dissertation possible.

Finally, thanks are not adequate for the love, patience, encouragement, understanding, support, and financing that my parents have given to me throughout my life and which has made this dissertation a success.

To my parents

Chapter 1

Introduction

With the advent of personal workstations and local area networks, locally distributed systems are now commonplace. Such systems are inherently more complex than non-distributed systems, and the programming of distributed applications is still considered something of a black art. Numerous operating systems and programming languages have been proposed to reduce this complexity. Distributed applications typically require the sharing of data among remote entities. Such sharing is usually achieved by remotely accessing shared data or by copying shared data between cooperating entities. Existing systems and languages have either prohibited distributed sharing, or provided two levels of support, one for non-shared data contained entirely within a single machine (or process) and another for data which is shared across machine boundaries. Most systems permit a limited form of sharing by allowing smaller amounts of data to be *shared by copy*, for example, by including the data in messages or as parameters to remote procedure calls; we call such copying *fine-grained* because the copied data can be arbitrarily small.

We propose unifying these different approaches to data sharing by using a uniform object model. All data, shared as well as non-shared, is represented as objects ranging from data units as small as a Boolean value or as large as a transaction manager. Objects are accessed by reference using a single mechanism, namely that of *object invocation*. An invocation is essentially a procedure call, but if the referenced object resides on another machine the invocation will be a remote procedure call, making the network transparent to the programmer. An object is shared when more than one other entity has a reference to the object. Objects are shared by reference, not by copy. Sharing by copy can be perceived to be composed of two operations: the creation of a copy followed by a move of the copy to the remote location. We believe that the ability to move an object within a distributed system is important in itself. We therefore propose *on-the-fly fine-grained object mobility* as a fundamental concept in the system. By on-the-fly we mean that an object can be moved from one machine to another at any given time—even while one or more processes are executing within it. By fine-grained we mean that objects of arbitrarily small size can be moved individually. In contrast, earlier systems with mobility facilities allow only entire process address spaces to be moved; we call this *coarse-grained* mobility.

The viability of our ideas is demonstrated by construction. We have participated in the design of a new programming language called *Emerald* that is object-oriented, has a uniform object model, and includes primitives for fine-grained mobility. We have built an Emerald prototype system on top of a network of MicroVAX workstations. The system consists of a compiler for Emerald that generates native VAX code and a run-time kernel that provides support for distribution and object operations. The Emerald prototype is used as a testbed for our experiments with object mobility.

Because object-oriented systems hitherto have been plagued by performance problems, an important goal of Emerald is efficient execution; specifically efficient node-local execution. Anyone building an object-oriented system must be sensitive to performance because of the generally poor performance of such systems; numerous examples of Smalltalk performance problems are described by Krasner [Krasner 83]. While distribution and mobility increase the generality of a system they often reduce its performance. For example, the Eden system [Almes 85] implements a general object model but Eden objects suffer from poor node-local performance. The implementation of mobile objects in the Emerald prototype involved tradeoffs between the performance of mobility and that of more fundamental mechanisms, such as local invocation. Where possible, we have made these tradeoffs in favor of the performance of frequent operations and would typically be willing to increase the complexity of mobility to save a microsecond or two on node-local invocation. An object move, for example, takes a hundred times longer than a local invocation; adding 5 microseconds to the object move time makes little relative difference, while 5 microseconds is 25 percent of the local invocation time. The result of this philosophy is that, to a great extent, the existence of distribution and of mobility in Emerald does not interfere with the performance of objects on a single node.

In our system, we have obtained efficiency by (1) integrating the object concept, including mobility, into the Emerald programming language, by (2) relying on close cooperation between the compiler and the run-time system, and by (3) implementing all objects on each node within a single address space. We believe that node-local performance should not suffer due to the presence of distribution. To evaluate the efficiency of the Emerald prototype, we have measured the prototype and several simple applications.

In summary, our approach incorporates three main ideas:

- A uniform object model for both local and distributed computation
- Fine-grained mobility
- Language support for mobility to achieve an efficient implementation

The main emphasis of this dissertation is object mobility as it appears in Emerald and as it is implemented in the Emerald prototype. The implementation problems encountered during the construction of the run-time kernel are described extensively.

Emerald has been described in several articles, see [Black 86, Black 87]. Parts of this dissertation have been published in [Jul 88b]. The language design and the compiler are discussed in Norman C. Hutchinson's Ph.D. dissertation [Hutchinson 87a]. The language is described in a language report [Hutchinson 87b].

1.1 Underlying Assumptions

We restrict our attention to distributed systems consisting of a moderate number (in the hundreds) of autonomous computers connected by a local area network such as an Ethernet or a token ring. The computers are assumed to be homogeneous uniprocessors capable of independent operation and independent failure. Processors are assumed to fail in a *fail-stop* manner, that is, when a failure occurs the processor stops immediately and does not generate erroneous data. The network is assumed to be capable of unreliable delivery of point-to-point datagrams of limited size (on the order of one thousand bytes), and of unreliable broadcast of datagrams. Datagrams may be delayed, lost, duplicated, or reordered in transit but not corrupted—corrupted datagrams are assumed to be detected by the network and dropped. The network is assumed to be several orders of magnitude slower than intra-processor procedure calls; however, it should deliver at least ten to one hundred datagrams per second with a high probability of success. Neither long-haul networks nor network partitioning is considered.

We use the terms *machine*, *node*, and *host* synonymously for denoting a single computer in the network. We shall also use the terms *mobility* and *migration* synonymously to refer to the ability to move data or processes from node to node. We say that a process is *heavy-weight* when it has its own address space. In contrast, *light-weight* processes share address space with other processes.

1.1.1 Why Mobility?

Previous work has demonstrated the following benefits of mobility:

Improved resource utilization by load sharing – By moving processes within the system one can take advantage of unused resources such as idle CPUs.

Communications performance – Entities that interact intensively can be moved to one node (at least) for the duration of their interaction, to reduce the communications cost. Increasing locality of reference can significantly improve performance because remote communication can be three orders of magnitude as costly as local communication. Co-location can thus be essential to good performance.

It is worth noting that co-locating frequently communicating, CPU-bound processes may be counter-productive as CPU contention can result in a decrease in real concurrency. In the extreme, the co-location policy could lead to moving *all* processes on one machine to eliminate all overhead due to remote communication—clearly a poor idea.

Availability – Data or data replicas can be moved to different nodes to provide a higher degree of availability in the face of node failures.

Reconfiguration – Programs can be moved following either a failure or a recovery, or prior to scheduled down-time, for example, to run uninterrupted during preventive maintenance.

Utilizing special capabilities – An object can move to take advantage of unique hardware or software capabilities on a particular node, e.g., a floating point accelerator, or simply a faster CPU.

User mobility – When users move from one machine to another, running programs can migrate to the new machine without disruption. Zayas [Zayas 87b] describes an example where a student starts a long-running simulation program on a local workstation and later migrates it to a machine in a computer laboratory to view the progress of the simulation.

In addition to these advantages, fine-grained mobility offers further opportunities because the basic unit of mobility is smaller and hence less costly. Fine-grained mobility provides three additional benefits:

Data movement – Fine-grained mobility provides a very simple and inexpensive way for a programmer to move data from node to node without having to explicitly package it. No separate message passing or file transfer mechanism is required.

Permits the use of call-by-reference semantics – In most distributed systems call-by-reference semantics are avoided because of excessive implementation complexity and cost. Mobility has the potential to improve the flexibility of remote procedure calls by avoiding the performance penalties associated with call-by-reference semantics.

Garbage collection – Mobility can help simplify distributed garbage collection by moving objects to nodes where references exist [Hewitt 80, Vestal 87].

1.2 The Main Ideas Behind Our Approach

The next three sections elaborate on the three main ideas of our approach.

1.2.1 Uniform Object Model

Object-oriented systems typically lie at the ends of a spectrum: object-oriented languages, such as Smalltalk [Goldberg 83] and CLU [Liskov 77], provide small, local, data objects; object-oriented operating systems, such as Hydra [Wulf 74] and Argus [Liskov 88], provide large, possibly distributed, active objects. In distributed systems, machines are physically separated by the network and programmers have to deal with two levels of operations: local, accessing shared memory, or remote, accessing data across the network. Distributed systems that support both models of computation have a separate definition mechanism for each model. For example, in the Eden system [Lazowska 81, Almes 85] a programmer can use the object model of computation in addition to the model of computation presented by Concurrent Euclid [Holt 83], which can be used within objects. Similarly, the Argus system provides so-called *guardians* for distributed computing and CLU clusters for computing locally within a guardian [Liskov 88]. The presence of two models of computation complicates programming. In Eden, the programmer could choose to use the object model almost exclusively, but in practice is prevented from doing so because Eden suffers from poor performance for objects that are located on the same machine. The two models differ in that one can only be used locally and is fast while the other has full generality but is much slower—usually three orders of magnitude slower. In distributed object-oriented systems, e.g., Clouds [Spafford 86] and Eden [Black 85], a *local* execution of the general object invocation mechanism can take milliseconds or tens of milliseconds—even when both objects are located on the same machine. Thus a programmer can be forced to use one model of computation where another model would be more appropriate, or to accept the poor performance of the general mechanism. Two examples illustrate this. First, Pu developed a distributed transaction system that made extensive use of Eden's distribution features [Pu 86]. The object-oriented approach used in Eden was found to be quite useful, but its use was seriously hampered by the large resource requirements of Eden objects (each was a Unix process). Because of resource problems, parts of the transaction system could not be written as Eden objects. Second, while programming a Collaborative Editing System in Argus, Greif et al. have observed that a designer can be forced to use a distributed guardian where a local CLU cluster would have been more appropriate [Greif 86].

Emerald advocates the use of a single uniform model of computation for both distributed and non-distributed programming. For many applications, distribution need not be visible. Throughout the history of computers the concept of *transparency* has been used to simplify programming (see [Parnas 75]). High-level programming languages in general make the underlying computer architecture transparent to the programmer. In Emerald, we have made the underlying distributed architecture transparent: programmers use a single object definition mechanism with a single semantic for defining all objects. This includes small, local, data-only objects and active, mobile, distributed objects. The Emerald compiler analyzes the usage of each object and generates an appropriate implementation. For example, an array object whose use is entirely local to another object will be implemented differently from a globally shared array. The compiler produces different implementations from the same piece of code, depending on the context in which it is compiled. We use the Emerald object model as a basis for our investigation of mobility.

1.2.2 Fine-grained versus Coarse-grained Mobility

Previous work on mobility has emphasized *process migration*, where heavy-weight processes and their associated address spaces are moved in one operation. In terms of implementation, the entities moved are entire address spaces, so *virtual address space mobility* would be a more descriptive term. Individual data entities within an address space, e.g., a record, cannot be moved independently; the smallest movable entity is an entire process or program. Hence, we call this form of migration coarse-grained. Examples of systems with coarse-grained mobility include DEMOS/MP [Powell 83], Distributed V [Theimer 85], Accent [Zayas 87a], Eden [Almes 85], and Sprite [Douglis 87]. The main purpose of process migration in these systems has been to improve performance by better resource utilization, e.g., moving a large task to an idle workstation. Several of these systems are surveyed and used to illustrate important design issues.

We say that a migration facility is fine-grained if movable entities can be smaller than an entire address space. None of the systems surveyed provides fine-grained mobility. However, all have facilities for fine-grained data transfer. For example, Distributed V provides message passing for transferring small amounts of data (up to 32 bytes) between distributed processes. Larger data areas can be accessed remotely by passing suitable access rights in messages. Because the V system also provides coarse-grained migration of programs, the programmer has a choice of three different data transfer mechanisms.

In contrast to previous systems, Emerald uses a single mechanism for all data movement: Any object, regardless of size, is allowed to move freely from node to node, hence mobility is fine-grained. On the other hand, because objects can be arbitrarily large and can contain arbitrarily many processes, fine-grained mobility subsumes both data movement mechanisms and process migration. Note that it is the use of an object-oriented model of computation that makes the introduction of fine-grained mobility possible because objects are encapsulated and have a well-defined access mechanism. Fine-grained mobility provides several advantages but also poses new design problems.

1.2.3 Language Support for Mobility

In the Eden system, the migration of objects was not supported by the Eden Programming Language; instead it was supported by calls to the run-time system. Thus the semantics of the language cannot take mobility into account [Black 85] and the implementation thereof is more difficult because the compiler and the run-time system cannot cooperate concerning mobility. The implementation of mobility in Eden has several restrictions. First, when an object is moved, all processes executing within the object are lost. This includes all active invocations of the object. Second, until the moving object has been reincarnated at its new location, all incoming invocations fail.

In Emerald, the mobility concepts are integrated into the language. This permits extensive cooperation between the compiler and the run-time system. Such cooperation has resulted in large gains in efficiency over previous systems.

1.3 Goals

We expect to achieve the following goals in the Emerald prototype:

Full fine-grained mobility

A novel feature of the Emerald system is the full and unrestricted implementation of fine-grained mobility.

Efficient node-local objects

The use of a single data abstraction model does not imply a single data representation model. Most data is used solely on one machine and the full power of a general, distributed representation is not needed; an efficient, non-distributed representation will suffice. In earlier systems, such as Eden, the programmer must choose between the two representations by selecting one of two data abstraction models. By integrating the mobility concepts into the language, we enable the compiler to detect objects that are local to one machine and to choose a simple, non-distributable implementation for these objects. In general, a tight coupling of compiler and run-time system allows us to achieve the efficiency of non-distributed systems for operations that are contained within a single node.

Efficient distribution of objects

Using mobility, we enable distributed programs to run more efficiently by co-locating frequently communicating objects.

1.4 Semantic Issues

Integrating the concept of mobility into a programming language raises several semantic issues which are discussed in the following sections.

1.4.1 Transparency

As noted in previous work, distribution transparency is important for simplifying many applications. However, there are applications that need full control over distribution, for example, a replicated name server. Therefore, there is a need for facilities that allow applications to control distribution while at the same time making distribution transparent. We discuss the conflict inherent in attempting to provide distribution transparency and distribution visibility simultaneously.

1.4.2 Unit of Mobility

Previous systems with mobility have used a coarse-grained unit of mobility. In Emerald, we investigate the use of a smaller unit. Object-oriented systems seem to be particularly well suited for this purpose for several reasons. First, objects are encapsulated, that is, their internal representation is not externally visible (see [Parnas 72]). Second, objects interact only through a well-defined mechanism, namely that of object invocation.

1.4.3 What Moves?

An important issue when moving objects containing references to other objects is deciding how much to move. Obviously, moving an object without moving its local data structures does not make sense because the local structures would wind up always being remotely referenced. On the other hand, not all other referenced entities should be moved, e.g., anything local to the node, such as a node-local file system, should remain where it is. The question is, when moving an object what other objects should also be moved? In general, there does not seem to be an easy answer to this question. We discuss several different methods for grouping objects together for the purpose of mobility. Our solution calls for building groups of objects that are to move together by *attaching* objects to each other.

1.4.4 Move: Hint or Demand?

In Emerald, we introduce an explicit notion of location into the language. An object is at any time located at one and only one node. The programmer can request that the object be moved to another node. If the move is requested for reliability reasons, e.g., to create replicated copies of an object on distinct nodes, it is important that the object actually be moved to the requested node. If the move is requested for performance reasons, it is not essential to the correct operation of the system that the object actually is moved. The system may decide (based on run-time information as memory availability) not to move an object, or, at least, to delay the move. Delaying the move can increase performance by reducing the total number of network messages. The design of a mobility facility must specify whether a move request should be considered a *hint* or a *demand*, or whether or not both possibilities should be available.

1.4.5 Parameter Passing Semantics

Not all possible parameter passing semantics make sense in a distributed system. Because of the lack of a shared address space passing references over a network is non-trivial. Many distributed systems therefore have restrictions on how parameters are passed. Some even require all parameters to be passed by value [Herlihy 82]. An object does not have a "value" because among other reasons its state can include the execution state of processes. Hence, it is not always possible to make an exact copy of an object. Therefore, in Emerald we have abandoned the notion of value. All parameters are passed using call-by-object-reference. We discuss different parameter passing modes and how distribution and mobility affect them. Call-by-object-reference can be quite inefficient for remote calls because parameters must be accessed across the network. We overcome this problem by allowing parameters to move. This results in a new parameter passing mode that we call *call-by-move*.

1.4.6 Immutable Objects

CLU [Liskov 77] includes the concept of *immutable objects*. An object is immutable, if it cannot change over time. The object's operations must also always return the same result given the same parameters, that is, they are functions in the mathematical sense. For example, the object representing the integer "17" does not change over time: the negation operation will always return a reference to the object representing "-17". Immutable objects are especially useful in distributed systems because they can be replicated on all nodes where references to them exist. When an immutable object is accessed, the local replica is used and the network is not used at all, resulting in significant performance gains. When an immutable object is passed across the network, it is copied. In effect, we achieve call-by-value, while maintaining call-by-object-reference semantics.

1.4.7 Concurrency Issues

To fully utilize distribution, it is necessary to provide facilities for efficient multiple processes and to give the programmer explicit control over processes and their synchronization. Distributed computations can be concurrent and even non-deterministic. Different parts of a distributed program can be executing on separate nodes at the same time and events cannot always be chronologically ordered, although a consistent total ordering can be established based on the partial ordering determined by the messages exchanged between nodes [Lamport 78]. We discuss how to provide concurrency and synchronization.

1.5 Design Issues

In the following, the main design issues are introduced.

1.5.1 Address Space

The degree to which objects share address space has a large performance impact. There is a tradeoff between protection and efficiency. Objects in different address spaces are protected against one another and it is easy to provide good damage containment. However, crossing address space boundaries is expensive as are references spanning several address spaces. Objects within a single address space can communicate very efficiently, but are not well protected against each other.

1.5.2 Multiple Implementations

Objects that are to be moved independently within the network and that concurrently can be invoked remotely by other processes require extensive and expensive support. A major design issue concerns objects are referenced across the network in a manner that allows them to be moved on-the-fly without sacrificing intra-node efficiency. Not all objects require the full generality of fine-grained mobility. For example, some objects are entirely local to another object and can neither be moved separately nor be invoked remotely. For these objects, a simpler, more efficient non-distributed implementation suffices. Immutable objects can similarly be implemented efficiently because they never need to be invoked remotely. We discuss the idea of multiple implementations and describe the performance gains obtainable.

1.5.3 Moving Objects

When moving an object from a source node to a destination node, the source node must extract the object from the address space in which it resides, package its representation into a network message, send the message, and, finally, fix node-local references to point to the object's new location. The destination node must unpackage the message, reincarnate the object, integrate it into the node's address space, and, finally, fix intra-node references to the object so that they point to it. The main problem is that of handling addresses across the network. We discuss using a global name space and compare it to using node-local name spaces and translating addresses when they cross node boundaries.

When moving an object that contains one (or more) executing processes, the processes must also be moved. This involves extracting the processes from the shared object address space (including processor registers and the process stack) and sending them to the destination node. A major problem determining what processes are executing in an object at a given time without incurring excessive run-time overhead on every invocation. Another is handling processor registers for migrating processes. Processor registers contain addresses that are node-dependent, e.g., the current instruction pointer and the current stack pointer. Processor registers also present a problem when they are stored on the stack, e.g., in the form of procedure return addresses. When moving a process, all such registers and register copies must be found and translated.

1.5.4 Code

Earlier migration systems move entire address spaces including all code. When using fine-grained mobility, transferring the code on every move would substantially increase the migration cost for small objects because the code is much larger than the data. Furthermore, we expect that using fine-grained mobility will cause many objects that have the same code to be transferred. We discuss methods to limit the amount of code transferred and to maximize code sharing.

1.5.5 Keeping Track of Mobile Objects

When objects move frequently between autonomous nodes, it is difficult to keep track of them. There is a spectrum of methods for doing so ranging from complete knowledge to no knowledge. Complete knowledge requires that every node knows where every object is. This is computationally very costly and requires that every node be updated following every move. This would make mobility unacceptably expensive. No knowledge means that no location information is maintained. This makes mobility cheap, but every reference is very costly because the referenced object must be located on every reference.

Chapter 5 is devoted to a discussion of how to keep track of mobile objects. Our solution is based on the concept of *forwarding addresses* as described by Fowler [Fowler 85, Fowler 86]. In our model of computation nodes can crash independently. Such crashes can cause location information to be lost. We therefore extend Fowler's work by discussing recovery of such lost information.

1.5.6 Distributed Garbage Collection

Emerald, like many other object-oriented languages, does not concern itself with allocation and deallocation of object storage. Instead, object storage management is considered an implementation issue. Storage allocation does not present any major problem while deallocation (garbage collection) in a distributed system is difficult enough to warrant discussion in a separate chapter. A major problem is to provide a collector that can run in parallel with applications without causing excessive delays to the applications.

Distribution adds a whole set of new problems to uniprocessor garbage collection algorithms. First, access to information about remote objects is time-consuming due to network latency. Second, the inherent parallelism in distributed systems causes synchronization problems. Third, node crashes may cause information about object references to be inaccessible.

Fourth, object mobility itself complicates garbage collection although there are simple methods to overcome this complication. Thus the significant problems in distributed garbage collection stem from distribution itself—not from the presence of mobility.

The efficiency of a distributed garbage collector can be greatly increased on the basis of the observation that most garbage never has been referenced remotely. This means that the garbage collection problem can be divided into two problems solved by having two collectors: a node-local collector that runs frequently and avoids inter-node communication and a distributed collector that runs infrequently to collect garbage that spans more than one node. Chapter 6 discusses the problems involved in designing garbage collectors for a system supporting fine-grained mobility. A novel variation on the classic mark-and-sweep algorithm is proposed.

1.6 The Dissertation

This dissertation studies the design and implementation of fine-grained mobility in the Emerald distributed programming language and system. The language facilities are presented and the major mobility issues are described. The viability of the ideas presented is demonstrated by designing, implementing, and measuring a Emerald prototype system and by running several simple applications on top of the system. Emerald includes a full implementation of fine-grained object mobility. The compiler was written by Norman C. Hutchinson, whose dissertation describes the Emerald type system and the Emerald compiler [Hutchinson 87a].

1.6.1 Research Contributions

The major research contributions presented in this thesis include:

Fine-grained mobility

A novel feature of Emerald is the ability to efficiently move any object regardless of size in a transparent manner. Object mobility subsumes process mobility because processes that are executing inside objects move with their containing object. Mobility is on-the-fly; there is no restriction on when an object may be moved. For example, a migrating object is allowed to have any number of processes executing inside it and there may be processes waiting on monitors or on conditions inside the object.

The faulting distributed garbage collector

We present a novel variation of the classic mark-and-sweep garbage collection algorithm that allows the concurrent collection of garbage in a distributed system. We introduce the concept of a *garbage collection fault* which is used to obtain a collector that runs in parallel with user processes.

Efficient object implementation

We show how the object model of Emerald can be implemented efficiently.

Move groups

We present the concepts of move groups and attachment to allow related objects to move in unison. We also extend this idea to parameter passing by introducing the *call-by-move* parameter passing mode.

Support for multiple object implementations

A given object can be represented in one of several different forms depending on its usage. The use of multiple implementations allows all objects conceptually to have full generality while efficiently implementing those objects that do not require it.

Object finding

We extend Fowler's work on object finding by handling situations where location information must be recovered due to node crashes.

1.6.2 Limitations

This dissertation mainly addresses the mechanisms related to mobility. It does not explore higher-level issues such as policies for moving objects, load sharing, etc. This leaves an important area, namely the software engineering issue of demonstrating the claimed simplification of programming distributed applications. This can only be done by actually programming a large number of applications and running them over a long period of time—a task too large to be encompassed by this presentation. However, we do report the result of adding mobility to a moderately large application (section 7.5).

1.6.3 Dissertation Organization

This dissertation concentrates primarily on the language and run-time mechanisms that support fine-grained mobility while retaining efficient intra-node operation. In the next chapter, we present an overview of the Emerald language and system, and its mobility and location primitives. We present the language constructs, but defer the design decisions behind them until later chapters. Chapter 3 contains the main discussion of fine-grained mobility and the semantic issues involved. The discussion is limited to issues at the conceptual level; implementation issues are treated in Chapters 4 through 6. Chapter 4 describes how objects and processes are implemented and moved. Two of the implementation issues, object location and distributed garbage collection, are considered important enough to merit separate chapters. Chapter 5 discusses how to keep track of highly mobile objects and how to find them when the need arises. Chapter 6 presents the design of the Emerald garbage collector. Chapter 7 presents measurements of the prototype implementation. In addition to measurements of basic language features, we also present the results of adding mobility to a moderately large, nontrivial Emerald program. Chapter 8 draws conclusions from our design and implementation experience.

Chapter 2

Overview of the Emerald Language

The purpose of this chapter is to provide an overview of the Emerald language. We start by presenting a motivation for the language based on a historical view of developments in programming languages. We then present an overview of the language constructs and their semantics. The language constructs are not justified, they are merely presented. The mobility features are justified in Chapter 3 and the entire language is justified in [Black 86, Black 87, Hutchinson 87a]. For a detailed description of the language, see the Emerald Language Report [Hutchinson 87b].

The Emerald language was developed specifically for distributed computing and has been influenced by a number of previous languages and systems. We view Emerald as a step in a natural progression from early programming languages for primitive computers to distributed programming languages for complex networks of computers. Because distribution transparency was an important design goal, most language features are independent of distribution.

Our goal is to do for distributed programming what concurrent programming languages did for parallel programming and what high-level languages did for sequential programming. In the early days of computers, programming was a difficult and error-prone task because there was no language support. Programs ran on essentially bare machines and the programmer had to deal directly with numerous low-level issues such as allocation of storage and registers, absolute addresses, the number of bits in registers, etc. High-level programming languages, such as Algol 60 [Naur 60], immensely simplified sequential programming by introducing such abstractions as variables, types, and procedures, along with strict rules for their use (e.g., scope rules and formal syntax). The handling of storage and registers was simplified by the concepts of variables and expressions and control flow was simplified by the concepts of conditional and loop statements and the concept of procedure calls.

Similarly, before the advent of concurrent programming languages, parallel programming was seriously handicapped by a lack of support for concurrent operations and the practice of parallel programming was for wizards only. Concurrent programming languages, such as Concurrent Pascal [Brinch Hansen 75], included the important abstractions of process and of strictly controlled synchronization. The process concept quickly gained acceptance as a language construct while synchronization concepts have taken different forms: monitors [Brinch Hansen 73, Hoare 74], path expressions [Campbell 74], guarded commands [Hoare 78], and rendez-vouz [Ichbiah et al. 79].

At the same time, the concept of object-oriented programming was being developed. The first language to incorporate the object concept was Simula, which extends Algol 60 with objects and classes [Nygaard 70, Birtwistle 73]. Simula, however, does not provide full encapsulation of objects and has two models of computation: the model inherited from Algol 60, and the object model. Smalltalk did away with this dual model and committed itself completely to the object-oriented style of programming by making everything in a program, even statements, into objects [Goldberg 83]. The success of the object-oriented approach to programming lead to its introduction into operating systems, starting with Hydra [Wulf 74] and later followed by Eden [Lazowska 81, Almes 85].

In the area of operating systems, the concept of a remote procedure call as presented in [Birrell 84] and developed into the invocation concept as in Eden was found to simplify distributed programming [Black 85]. A problem in Eden is that the use of the object mechanism is expensive—the minimum time for an Eden invocation is in excess of 50 milliseconds even when the objects involved are located on the same node.

Although Emerald is a new language, it has benefited from several of these ancestors. We adopt the uniform object model of Smalltalk and combine it with the distributed object model presented by Eden. By incorporating these ideas into our programming language, we are able to provide an efficient implementation, especially for objects that are used in a non-distributed way. It is an imperative language in the style of Algol 60. The constructs for concurrency and synchronization are derived from Concurrent Pascal. Many of the distribution ideas were derived from experiences with the Eden Programming Language [Almes 85, Black 85].

2.1 Emerald Objects

An Emerald object consists of:

- A unique network-wide identity.
- A representation, i.e., the data local to the object. For non-primitive objects this consists of variables that reference other objects.
- A set of operations that can be invoked on the object. Exported operations can be invoked from outside the object. Non-exported operations are local to the object and can only be invoked from within the object.

- An optional **initially** section. When the object is created, the code in this section is executed before any invocations of the object are allowed. This allows the **initially** section to initialize the variables of the object and thereby establish invariants for the object before it is invoked.

- An optional **process** section. This section specifies the statements to be executed by an anonymous process which is created when the object has been initialized.

Furthermore, an object has two attributes: it has a *location* and it may be *immutable.* Its location is the node where it currently resides. If it is immutable, its state is not allowed to change.

2.1.1 Processes

When an object is created, the statements in the **initially** section are executed (if present). Thereafter the optional process is created and executes the statements in the **process** section. Emerald objects that contain a process are active; objects without a process are passive data structures. Objects with processes can invoke other objects, which in turn can invoke other objects, and so on to any depth. As a consequence, a process originating in one object may span many other objects, locally as well as remotely. Multiple processes can be active concurrently within a single object. Synchronization is provided by monitors and conditions [Hoare 74]. Each object is divided into a monitored and a non-monitored part (using the keyword **monitor**). In the monitored part, processes have exclusive access to the variables declared in it and are free to update them. Processes do not have exclusive access to and cannot update the variables in the non-monitored part, and cannot access the variables in the monitored part at all. To avoid a number of common concurrency errors, the compiler checks that all variable updates take place within the monitored section.

2.1.2 Object Creation

In object-oriented languages such as Smalltalk objects are created by invoking an operation of a *class.* In contrast, Emerald objects are created directly by executing an *object constructor.* An object constructor is an executable language construct. When executed, it returns a newly created object. The construct contains the specification of the new object. This includes the variables that hold the object's data, its operations, and its initially and process sections.

2.1.3 Invocation

The basic mechanism for communication between objects is similar to a procedure call in Algol 60 or Simula; in our terminology, we call it an *invocation.* Given a reference to an object, e.g., as a variable named *target* and three variables named *result*, *arg1*, and *arg2* then an invocation could look like:

```
result ← target.op1[arg1, arg2]
```

The operation *op1* of the object denoted by the variable *target* is executed with the objects denoted by *arg1* and *arg2* as arguments. The operation returns a single object, which is assigned to the variable *result*. Both arguments and results are passed by *object-reference*. Invocations are synchronous.

2.2 A Simple Emerald Example

```
var myDir ← object aDirectory
    export Add, Lookup, Delete
    monitor
        const DirElement == record DirElement
            var name : String
            var obj : Any
        end DirElement

        const a == Array.of[DirElement].empty

        function Lookup[n : String] → [o : Any]
            var element : DirElement
            var i : Integer ← a.lowerbound
            loop
                exit when i > a.upperbound
                element ← a.getelement[i]
                if element.getname = n then
                    o ← element.getobj
                    return
                end if
                i ← i + 1
            end loop
            o ← nil
        end Lookup

        % Implementation of Add and Delete
    end monitor
end aDirectory
```

Figure 2.1: An Emerald Directory Object Definition

Figure 2.1 shows a definition of an Emerald object—in this case a simple directory object called *aDirectory*. In the example, a variable *myDir* is declared and initialized by being assigned the object that results from the execution of an object constructor. The new object has the representation and operations as specified in the constructor. The representation of the object consists of an array of directory elements referenced by the variable *a*. The object exports three operations: Add, Lookup, and Delete (only the function Lookup is shown). The array and the operations are defined within a monitor because the array is updated. The variable *a* is defined as a constant reference to an array object. The **const** means that the reference stored in the variable will not change; however, the array itself can change. Emerald arrays are dynamically expandable, so initially it suffices to create an empty array. Array elements are referenced using the *getelement* operation which takes an integer index as an argument and returns the object indicated by the index. The array elements are records consisting of two fields. Records can be defined using object constructors or, as in Figure 2.1, by using a simpler notation similar to Pascal's record notation. In the example, the array and records are accessed using the object notation. For convenience, shorthand notations for array and record accesses exist. For example, *a.getelement[i].getname* accesses the *name* field of the *i*'th element of the array *a* and can be written simply as *a(i)$name*. Finally, comments start with a "%" character and continue to the end of the current line.

2.3 Types in Emerald

The Emerald language is strongly typed. Its type system is based on the concept of *abstract type* [Black 87]. An abstract type defines an interface to an object: the number of operations that the object exports, their names, and the number and abstract types of the parameters to each operation. For example, consider the abstract type definition for *SimpleDirType*:

```
const SimpleDirType == type SimpleDirType
   function Lookup[ String ] → [ Any ]
   operation Add[ String, Any ]
end SimpleDirType
```

This defines an abstract type with two operations, *Lookup* and *Add*. *Lookup* takes a parameter of abstract type **String** and returns an object of abstract type **Any**. *Add* also takes a parameter of abstract type **String** and does not return anything. *Lookup* is a function and cannot have any side effects. *Add* is an operation and is allowed to have side effects.

We say that an object *conforms* to an abstract type if it implements at least the operations (and functions) of that abstract type, and if the abstract types of the parameters of the operations conform in the proper way. For example, the object *aDirectory* defined in Figure 2.1 conforms to *SimpleDirType*. The built-in type **Any** has no operation and therefore all objects trivially conform to it. Conformity is also defined as a partial relation between types. Informally, we say that a type A conforms to a type B (written $A \circ\!\!> B$) when B defines all the operations defined by A and when the parameters conform suitably. For example, the type *SimpleDirType* conforms to *ReadOnlyDirType*.

```
const ReadOnlyDirType == type SimpleDirType
   function Lookup[ String ] → [ Any ]
end SimpleDirType
```

The conformity relation "○>" is reflexive and transitive but not symmetric. Types form a partial order with conformity as the ordering relation.

Conformity is the basis for type checking in Emerald. When declaring a variable, the programmer must specify its type. When an object is assigned to a variable, the object *must* conform to the declared abstract type of the variable. Every variable and expression must be typed and must conform to the required type when used. Conformity is further discussed in [Black 87] and thoroughly detailed in [Hutchinson 87a].

Abstract types permit new implementations of an object to be added to an executing system. To use a new object in place of another object, the new object must conform to the required abstract type. For example, we could assign the object *aDirectory* in Figure 2.1 to a variable declared to have abstract type *SimpleDirType* because *aDirectory* conforms to *SimpleDirType*. Note that each object can implement a number of different abstract types, and that an abstract type can be implemented by a number of different objects.

In contrast to Smalltalk, Emerald has no class/instance hierarchy. Objects are not members of a class; conceptually, each object carries its own code. However, for efficiency, identically implemented Emerald objects on each node do share code.

2.3.1 Viewing and Restrictions

Because Emerald objects may conform to more than one type, it may be appropriate to change one's *view* of a particular object at run-time. This change may either be a *widening*, that is, the number of operations viewed as being supported by the object is increased, or a *narrowing*, that is, the number of available operations is reduced. Narrowing requires no run-time check of its validity because any object conforming to a type X in the partial order also conforms to all types that are greater than X. Widening, on the other hand, requires a run-time check of its validity. The system must check that the object in fact does support the operations required by the new type.

An example of where such view changes are required is in the implementation of the directory system mentioned above. Consider the type *SimpleDirType*, the variable *myDir* (which references the directory object *aDirectory*), and an additional variable declared as

var *a: SimpleDirType*

The following invocation inserts the directory into itself under the name "myself".

myDir.Add("myself", myDir)

Because the type of the second argument to *Add* is **Any**, the type of myDir is narrowed to the type *Any* before the parameter is passed the directory. The object reference could be retrieved from the directory by:

$a \leftarrow$ *myDir.Lookup*["*myself*"]

However, the assignment will not be allowed by the compiler because the type of the result of *Lookup* is **Any**, and **Any** does not conform to *a*'s type which is *SimpleDirType*. Therefore, the assignment is not type-correct. But it is known that the object returned by executing *Lookup* on *myDir* with the argument *"myself"* conforms to *SimpleDirType*, so we can use an explicit change of view:

a ← **view** *myDir.Lookup*[*"myself"*] **as** *SimpleDirType*

The **view** operator widens the given reference to the type specified. Such widening can a priori only be type-checked at run-time so the compiler generates a call to a type checking routine in these cases. To place a limit on the widening of an object reference, an explicit restriction can be placed on the reference:

var *b*: *SimpleDirType*
b ← **restrict** *myDir* **to** *ReadOnlyDirType*

Any future widening of the reference contained in *b* will hereafter be limited to types that conform to *ReadOnlyDirType*. For example,

view *b* **as** *SimpleDirType*

will cause a run-time error. Note that it is the object reference that is restricted; neither the object nor the variable is restricted.

2.4 Distribution

In our model of computation, the underlying system consists of a number of autonomous nodes connected by a network. At any given time an object is located at one and only one node as indicated by the object's location attribute. The concept of location is encapsulated in a node object which is an abstraction of a physical machine. The current location of an object may be obtained by the **locate** operator:

locate *X*

The **locate** operator returns the current location of the object indicated by X. A location is represented as a reference to the (one and only) node object for the node. A location may be specified by naming either a node object or any other object. If the programmer specifies a non-node object, the implied location is the node on which that object resides. Node objects also support operations that allow access to information about a node, e.g., time of day.

2.4.1 Moving Objects

Object mobility in Emerald is provided by a small set of language primitives. An Emerald object can:

- **move** an object to another node, e.g., "**move** X **to** Y" moves the object X to the node where object Y is currently resident.

- **fix** an object at a particular node, e.g., "**fix** X **at** Y" atomically moves X to the node where Y is located and prevents X from being moved.

- **unfix** an object, making it mobile again following a fix, e.g., "**unfix** X".

- **refix** an object, atomically performing an **unfix** and a **fix** at a new node, e.g., "**refix** X **at** Z".

The **move** primitive is actually a hint; the kernel is not obliged to perform the move and the object is not obliged to remain at the destination node. **fix** and **refix** have stronger semantics; if the primitives succeed the object will stay at the destination until it is explicitly unfixed.

2.4.2 The Call-by-move Parameter Passing Mode

Parameters to invocations are passed using call-by-object-reference semantics. In some cases it may be more efficient to move the parameter objects to the callee's node so that they may be referenced locally. As we shall see in later sections, many expensive remote invocations can be avoided by moving those parameter objects that are frequently referenced.

For the purpose of parameter mobility, Emerald provides three variants on the call-by-object-reference parameter passing mode. The *call-by-move* mode is specified by prepending the keyword **move** to the actual parameter. For example, to use call-by-move for the second parameter, the following call

result ← *target.op1*[*arg1, arg2*]

is changed to

result ← *target.op1*[*arg1,* **move** *arg2*]

This causes the parameter object to be moved to the callee's node; in all other aspects the mode is the same as call-by-object-reference. The *call-by-visit* mode is specified by prepending the keyword **visit** to the actual parameter. Call-by-visit is the same as call-by-move except that when the invocation completes, the parameter object is moved back to the caller's node. *Call-by-move-return* is specified by adding the keyword **move** to a formal result parameter.[1] When an invocation of the operation completes, the actual result object is moved to the caller's node. For example, in Figure 2.1, to pass the result from *Lookup* using call-by-return, the declaration is changed to

function *Lookup*[*n* : **String**] → [**move** *o* : **Any**]

In Chapter 7 we show an example of how performance can be improved using parameter mobility.

[1] Note that call-by-move-return is specified for the formal parameter whereas the two other move parameter passing modes are specified at the point of call.

2.4.3 Node Failures and Partial Availability

A centralized system either runs or is crashed. In contrast, a distributed system based on autonomous nodes can be partially crashed. The problems that arise from this complication are central to systems that maintain on-line data for long periods of time, e.g., the Argus system [Liskov 88]. Such systems include facilities for long term storage and for consistent updating of the stored data.

In Emerald, we have restricted our attention to the computational problems arising from crashed nodes. We provide facilities for the detection of node crashes but leave it to the application programmer to handle the problem. Our goal has been to enable a distributed work-load manager object, for example, to keep track of available nodes and to be actively notified when nodes crash.

A node is said to be *available* when it is running; otherwise, it is *unavailable*. Correspondingly, an object is regarded as being *unavailable* when it cannot be located at any of the available nodes. When invoking an unavailable object (or when a node crashes during a remote invocation) the invocation fails. This failure is propagated back to the calling object where it can be handled by a special *unavailable handler*. An unavailable handler may be added to any block in Emerald. (Essentially, a block is a collection of statements surrounded by the keywords **begin** and **end** or the like.) The syntax for an unavailable handler is:

unavailableHandler ::=
    when[*identifier*[“:” *typeDenotation*]] **unavailable**
        declarationsAndStatements
    end unavailable

The handler specifies the action to be taken when an invocation to a remote object fails because the object is unavailable. The optional identifier is assigned the unavailable object so that the handler can identify the object that caused the handler to be executed. An unavailable handler enables a program to recover from node crashes. For example, a load balancer might protect its calls to objects on other nodes using unavailable handlers, so that the load balancer can continue execution despite other nodes crashing. If there is no unavailable handler when needed then a failure occurs as described in the next section.

2.4.4 Failure Handlers

When an Emerald process executes some invalid operation, it *fails*. Failures can result from a number of causes: attempting to invoke the **nil** object, division by zero, stack overflow, dynamic type check error, and many others. Emerald provides a limited form of exception handling by allowing the programmer to attach a *failure handler* to any block:

failureHandler ::=
        on failure
            declarationsAndStatements
        end failure

When a failure is detected, the statements in the failure handler are executed. If there is no failure handler, the failure is propagated back along the call chain until a failure handler is found. Along the way, each failed object is marked as failed and any future attempt to invoke one of them will fail. If no handler is found anywhere in the call chain then the process is terminated. The language also provides a **ReturnAndFail** statement so that, e.g., an operation called with invalid parameters may propagate the failure back to the caller without the called object being marked as failed. For example, **ReturnAndFail** is used by the system object *UNIXInStream* to return a failure when an attempt is made to read past end-of-file.

2.4.5 Node State Changes

Applications that take advantage of distribution often need to know the set of available nodes. When a node boots or shuts down, it sends a broadcast to tell all other nodes about its own state. Each node in the system maintains its own version of this set using these broadcasts. The sets can easily differ because broadcasts are unreliable and because nodes may crash at any time. Should a node crash during an invocation, it will be declared dead by the other node when the underlying network protocol has retransmitted too many times.

Each node object provides access to its version of the set of available nodes through an invocation called *getAllNodes*. Furthermore, applications can request that they be actively notified of node state changes. Applications make such a request by invoking the operation *setNodeEventHandler*[*handler*: *HandlerType*] where *HandlerType* is defined as:

```
const HandlerType == type iHandlerType
   operation nodeUp[Node, Time]
   operation nodeDown[Node, Time]
end iHandlerType
```

All subsequent node state changes cause the kernel to invoke the handler object (operation *nodeUp* or *nodeDown*, as appropriate). Such calls from the underlying system to application layers above it are termed *upcalls* by [Clark 85] and are a further development of the concept of an interrupt. [Ravn 80] shows how an interrupt may be considered a call of a procedure in a special monitor called a *device monitor*. [Jul 80] shows how to structure systems using device monitors and device processes. In Emerald, we have further simplified the idea because a device monitor is essentially a simple object.

2.5 An Example of a Distributed Emerald Program

Figure 2.2 shows a complete, executable Emerald program. Each node object supports a number of operations that allow a program access to some of the underlying kernel's data structures. The most important one is *getActiveNodes* which returns a list of available nodes. Our sample program consists of one process which calls its local node object to obtain the list of active nodes and then visits each of these nodes in sequence. The elapsed time for the entire round-trip is calculated and printed. Note that although the process is moved around in the system, the calls to its standard output still work because I/O in Emerald is done through I/O objects that are accessed using location-independent invocations.

```
import NodeListElement, NodeList from “Builtins”
const Kilroy == object Kilroy
    process
        const home: Node ← locate self
        var i: Integer ← 0
        var remoteNode: Node
        var startTime,diff: Time
        var myList: NodeList
        var theElem: NodeListElement
        stdout.PutString[“Start Node ” || home$LNN.asString || “\^J”]
        myList ← home.getActiveNodes
        stdout.PutInt[myList.upperbound + 1, 2]
        stdout.PutString[“ nodes active.\^J”]
        startTime ← home.getTimeOfDay
        loop
            exit when i > myList.upperbound
            remoteNode ← myList(i)$theNode
            move Kilroy to remoteNode
            stdout.PutString[“Moved to Node ” || myList(i)$LNN.asString || “\^J”]
            i ← i + 1
        end loop
        move Kilroy to home
        diff ← home.getTimeOfDay
        diff ← diff − startTime
        stdout.PutString[“Back home − total time ” || diff.asString || “\^J”]
        stdout.close
        stdin.close
    end process
end Kilroy
```

Figure 2.2: A Complete Emerald Program that Visits all Available Nodes

2.6 Language Summary

We have given a brief introduction to the Emerald language. Its most important feature is that it has a single object model for all computation—local as well as distributed. Emerald derives its distribution concepts from the Eden system, but goes further than Eden by integrating these concepts into the language. The remainder of this thesis discusses the design and implementation of a kernel that efficiently supports Emerald's distribution concepts.

Chapter 3

Mobility

This chapter investigates conceptual issues related to mobility. Three existing systems that provide some form of mobility are reviewed. This provides a basis for the subsequent discussion of various mobility issues. In presenting each issue, we start by discussing the general problem, proceed with alternative solutions, and finally present and justify the Emerald solution. Implementation issues are discussed in later chapters.

3.1 Previous Systems with Mobility

Mobility in some form has been implemented in a number of previous systems although none has been fine-grained. This section studies three systems that have a coarse-grained process migration facility. Each system is described briefly and the major issues that the designers of the system chose to emphasize are mentioned. We also summarize the experiences of the designers. The purpose of studying these systems is twofold. First, we review previous migration systems and can draw upon the experience obtained from them. Second, it allows us to introduce a several major design issues in an appropriate context.

3.1.1 Process Migration in DEMOS/MP

The DEMOS/MP system [Powell 83] is a distributed operating system developed at the University of California, Berkeley, based on the non-distributed DEMOS system [Baskett 77].

DEMOS was developed at the Los Alamos Scientific Laboratories for CRAY-1 computers. It supports heavy-weight processes and asynchronous message passing in a style similar to the RC4000 operating system [Brinch Hansen 70]. Processes refer to each other using special references called *links*.

DEMOS/MP runs on top of a number of Z8000 microprocessors connected by a local area network. DEMOS/MP, as DEMOS, supports processes executing in their own address space and extends the message passing implementation to allow message passing across the network. Process migration was added to DEMOS/MP for the purpose of allowing improved system throughput and fault tolerance. In DEMOS/MP, links are extended to be location-independent network-wide references so that the presence of a network and the actual location of message recipients is transparent to the sender of a message. All interaction with the environment of a process takes place using links, while procedure calls are used internally in processes. The only kernel calls supported are those for message passing; all other kernel operations are accessed by sending messages to the kernel.

DEMOS/MP is one of the first systems to support unrestricted on-the-fly migration of heavy-weight processes. Processes are the unit of distribution and of migration, so mobility is coarse-grained. Process migration is requested by sending a message to the kernel specifying what process to move and a destination location. The policy for deciding when and where to move a process is left to the process requesting the migration.

When a process is moved from a node A to a node B, it is said to have a *residual dependency* on node A if the process still depends on node A to function correctly. DEMOS/MP messages that are sent to a recently migrated process must be forwarded by the original node using a forwarding address. We say that the process has a residual dependency on the original node in the form of the forwarding address. To remove such residual dependencies, DEMOS/MP updates links as they are used. When a message is forwarded, a special update message is sent back to the originator of the message. The update message causes the originator's link to be updated. Such updating is similar to the *Jacc* protocol described in [Fowler 85], the only difference being that the Jacc protocol only sends one update message even if the original message was forwarded several times.

The purpose of adding process mobility to DEMOS/MP is to allow a process manager to perform load sharing. The design stressed transparency by isolating processes; all interaction with the environment of a process is through links. Such isolation eased the implementation but came at a price. Using message passing exclusively for communication can have performance penalties because a general communication mechanism must be used even where a simpler, less expensive mechanism (such as a procedure call) would suffice. The DEMOS/MP paper [Powell 83] includes some performance statistics but no timing data is given for message passing.

3.1.2 Remote Execution in the Distributed V System

A preemptable remote execution facility [Theimer 85] has been added to the Distributed V system built at Stanford University [Cheriton 88]. The purpose of the facility is to utilize idle workstations by off-loading programs such as compilations, text processing, and simulations. The facility allows the user of a workstation to request remote execution of programs. The user can either specify a workstation or let the system pick a workstation that has a low load. The requests are made at the command interpreter level, so the unit of distribution is an entire program. A remotely executed program may later be preempted and migrated away, should the workstation executing it be needed for other work. Distributed V distinguishes between *remote execution* and *process migration*. Remote execution is the ability to remotely create processes at specified nodes while process migration is the ability to move already existing and executing processes from node to node. For applications such as load sharing, remote execution may be sufficient if the life span of processes is short or loads are predictable. Migration becomes useful when processes execute for long periods of time or load conditions change rapidly. The V design addresses three basic issues.

The first issue is that of *transparency*. The executing programs must be provided with a network-transparent execution environment. With the exception of direct access to hardware devices, this goal was achieved. Distributed V uses location-independent message passing for communication between processes.

The second issue is that of *minimal interference*. The migration of a program should interfere as little as possible with its execution and with the rest of the system. Specifically, the time a program is suspended due to migration should be minimized. This was done by a technique called *pre-copying*. When migrating a program, its virtual memory pages are transferred while the program continues to execute. Thereafter the program is stopped and the modified ("dirty") pages are transferred again, after which the program is restarted on the remote node. Pre-copying reduces the delay experienced by the process although it does not reduce the actual transfer costs. (On the contrary, total transfer costs are increased because dirty pages are transferred more than once.)

The third issue is that of *residual dependencies*. A migrated program should have no residual dependencies, e.g., local temporary files. Files are usually accessed on a global file server thus avoiding residual dependencies due to local files. In general, the use of local files in the V system is not common because most of the workstations involved are diskless.

3.1.3 The Eden System

The Eden system [Lazowska 81, Almes 85, Black 85], designed and built at the University of Washington, is a distributed, object-oriented system. The purpose of Eden is to provide an experimental testbed for object-oriented, distributed programming. Informally, Eden objects can be described as distributed, communicating, and potentially long-lived Concurrent Euclid programs. Objects are addressed by secure capabilities and communicate using location-independent invocations. Invocations are essentially remote procedure calls (see [Birrell 84]) where the target object is specified by a capability. Parameters to invocations are restricted to be either capabilities, one of the basic Eden Programming Language data types, or sequences thereof. Eden runs on top of UNIX and each object is implemented by a UNIX process. The minimum size of such a process is on the order of 100,000 bytes because it contains the Concurrent Euclid program for the object, the run-time system for supporting Concurrent

Euclid, and support for communication with the Eden kernel.

An object may checkpoint itself by atomically writing its state into a special checkpoint file kept on stable storage. Should the object subsequently crash and be reincarnated, it can reestablish its state by reading the checkpoint file. Eden supports a limited form of object mobility through the checkpoint facility. An Eden object can move by specifying the target node as its reincarnation site and then deactivating itself (essentially it crashes itself). A subsequent invocation of the object will cause it to be reincarnated on the target node. The ability to move in this manner has been used mainly to achieve co-location of servers with clients. Inactive servers will typically checkpoint and crash themselves after a period of inactivity (typically 5 minutes) to release scarce system resources (virtual memory and swap space). When a client invokes an inactive server, the default is to reactivate the server on the same node as the invoker on the theory that the client will be the server's primary communication partner. In one case, object mobility has been used to migrate objects away from a diskless workstation with a serious hardware error. All checkpointed objects were restarted on a different workstation using checkpoint images stored on a file server.

Migration is not entirely transparent to users of the object. When an object moves, its UNIX process is crashed and all on-going invocations are lost and new incoming invocations fail until the object has been moved to another machine and reincarnated there. Migration is expensive—typically several seconds for a simple object. The unit of distribution is coarse-grained because each object is implemented as a Concurrent Euclid program running in its own UNIX process. A significant impediment to the implementation of object mobility is the use of standard node-local UNIX facilities and the use of C procedures inside objects. These ties to the current node represent an insurmountable obstacle to the migration of actively executing objects.

A number of different applications have been implemented using the Eden system. Pu describes the implementation of a distributed transaction manager [Pu 86]. Korry experiments with the criteria that should be used for distributed load sharing [Korry 86]. Banawan evaluates load sharing in locally distributed systems [Banawan 87]. Simulation is used to determine an appropriate load sharing measure to be used by a load balancing policy. Banawan performed several experiments using the Eden system but was prevented from experimenting with actual migration of objects because migration is not fully available. Pu, Noe, and Proudfoot describe maintaining replicated objects using a technique called *regeneration* [Pu 88]. They study implementation of replicated objects both at the kernel level and at the application level.

In general, the Eden experience showed that the object model is excellent for distributed computing [Black 85]. However, as noted by Black, Eden suffers from a lack of compiler support for the distributed and object-oriented features of the Eden Programming Language

(EPL). These features are supported by a preprocessor that converts EPL programs into Concurrent Euclid programs that are subsequently compiled by the standard Concurrent Euclid compiler. Thus the compiler does not directly support the special Eden language features. The Eden invocation mechanism suffers from poor node-local performance (because the general mechanism is always used). The full use of objects is hampered by the fact that all objects are large.

3.1.4 Summary of Previous Systems

The surveyed systems all provide similar migration facilities, generally in the form of migration of heavy-weight processes. In the following discussion, the term “process” is used as a generic term for both processes (as in DEMOS/MP) and objects (as in Eden). In general, the migration facilities enable a load managing process to perform load balancing by migrating processes on-the-fly. This emphasis on load balancing has consequences in several areas:

Transparency of migration

In most of the systems, the processes are oblivious to their own migration and even to the network. Therefore, distribution and migration is made transparent to the processes.

Coarse-grained mobility

Because load balancing involves large program units, the migration facilities are designed to be coarse-grained. The unit of mobility is an entire program.

On-the-fly migration

A migrated process is not involved in migration decisions and can be in any execution state. All the systems (except Eden) have on-the-fly migration.

Encapsulation and inter-process communication

To simplify implementation, inter-process communication (IPC) in the surveyed systems is restricted to kernel-supported IPC mechanisms. Common to these mechanisms is that processes reference one another through kernel protected data structures that we call *handles.* All inter-process communication is performed by special kernel operations where the destination process is specified by a handle. Handles encapsulate access rights and location information and can only be created or modified by the kernel. This eases the implementation of location-independent communication and of migration because the kernel has full control over handles. Handles can be passed from process to process; the kernel ensures that the handle data structure contains the information necessary to communicate with the addressed process.

No mobility at the application level

With the exception of Eden, none of the systems emphasizes mobility as a useful tool for distributed programming; rather, mobility is viewed as a tool for system load management. Most of the systems have a single "migrate" operation intended only for use by a load managing process. Eden is the first system to seriously develop location-dependent operations, i.e., operations that require or manipulate location information.

3.2 Design Issues

In this and the following sections, we discuss the major design issues, many of which were raised above. The issues are divided into the following categories:

- General issues concerning distribution.
- Language constructs to support mobility.
- The issue of how many other objects to move when moving a group of objects.

In the following subsections, we discuss issues that relate to distribution in general but that are not specific to mobility.

3.2.1 Distribution Transparency

Distributed programming can be significantly simplified by making distribution transparent to the programmer. One way of doing this is to extend the simple, well-established concept of a procedure call to the distributed environment; such inter-machine procedure calls are normally termed *remote procedure calls* (*RPC*). The first RPC system was developed by Birrell and Nelson at Xerox PARC [Birrell 84, Nelson 81] and allows remote modules to be *imported* (dynamically bound) to a variable in a program; subsequently, procedures in the module can be called as if the module is locally resident. Eden uses the same idea and provides for location-independent invocation of objects. However, in both these systems distribution is not entirely transparent. Distribution is visible because the programmer has two mechanisms to choose from: one for distributed computing and one for local computing. All the systems described in section 3.1 suffer from this problem. For example, DEMOS/MP extends the idea of links to the distributed environment to achieve transparent network access, but there are still two models of computation: procedure calls within processes, and message passing for IPC.

In Emerald, we have adopted a single model of computation and a single communication method, namely object invocation. Invocations are location-independent. The Emerald programmer does not have to be concerned about whether or not the invoked object is located on the same node as the invoking object. Invocations are also migration-independent: the invoked object or the invoking object is free to move at any time during the invocation. An invocation may start as a local invocation and later become a remote invocation, should any of the objects involved move. Similarly, a remote invocation may become a local invocation if one of the two involved objects is moved to the other object's node. However, at the language level, there is only one invocation mechanism used for all invocations—remote as well as local.

3.2.2 Should Distribution be Visible?

There are several kinds of application that run on distributed systems. In the words of Andrew Black [Black 85]: "some are born to distribution, while others have distribution thrust upon them." Distribution transparency simplifies the programming of the latter kind because it enables the programmer to ignore distribution. However, applications of the first kind require control over distribution if they are to take full advantage of it. For example, when designing a load balancing program, it is obviously necessary to have explicit control over location. Most of the surveyed systems were not designed for mobility at the application level. Only Eden provides a set of system calls that allows a programmer to perform location dependent operations [Sanislo 84, Black 85]. These operations provide facilities for obtaining a list of available nodes, remotely creating new objects (including processes), and specifying where an object should be checkpointed and restarted after crashes. Because on-the-fly mobility is not fully implemented in Eden, these operations are used mainly for providing remote execution in applications such as numerical calculations, load sharing [Banawan 87], and transaction management [Pu 86].

In Emerald, we have taken the Eden approach one step further and integrated the concepts of distribution into the programming language. There are several advantages to doing this rather than providing special system calls for distribution. We give the concept of "location" semantics. Because all the distribution concepts are present in the language, the programmer avoids having to mix concepts at the language level with those at the operating system level. Such mixing is undesirable because the two levels will invariably present two different models of computation (as noted in [Black 85]). As we shall see in Chapter 4, integrating distribution into the language also has implementation advantages that result in significant performance improvements.

Making distribution visible introduces a significant complication. There is an inherent conflict between distribution visibility and distribution transparency—it is not possible to have both. Therefore, we try to provide distribution visibility for those applications that need it and otherwise make distribution as transparent as possible; applications that do not use distribution do not have to be concerned with it. One area where the conflict clearly manifests itself is the problem of error handling. Every Emerald invocation is potentially a remote invocation and can be subjected to all the failures caused by the presence of distribution (e.g., node crashes). Either distribution is transparent and the programmer has no possibility of handling such failures, or it is visible and the programmer must be prepared for such failures on *every* invocation.

In Emerald we have chosen to make distribution visible because certain applications need control over it. A programmer must be prepared to handle failures caused by the presence of distribution. For non-distributed applications, we try to make distribution as transparent as possible, for example, by making invocations location-independent.

A programmer must be prepared to handle failures due to node crashes but does not have to worry about the actual location of objects unless required by the application. Invocations are also migration-independent in that the invoked object (or the invoking object) can move at any time during the invocation. We provide a mechanism whereby related objects remain together and are moved as a group. A programmer can thus ensure that a group of objects remain co-located and prevent invocations between them from failing due to other nodes crashing. Our goal is to maintain distribution transparency for any program that does *not* use distribution explicitly. Emerald programs can be written in a non-distributed manner and distribution can then be added with only minor modifications to the programs.

3.2.3 Unit of Distribution

When programs and data are distributed, individual entities are collected together in groups that are moved as a unit. For a given system, we call the minimum group that can be distributed independently the *unit of distribution.* Many systems use heavy-weight processes as the unit of distribution because a process neatly encapsulates a computation which can be distributed to other machines. Heavy-weight processes are normally restricted to use kernel supported inter-process communication. This simplifies the implementation because the kernel has control of all inter-process references. The unit of distribution is also the natural *unit of mobility*, i.e., the unit used when migrating processes and data. Distributed V uses *logical hosts* as the unit of distribution. A logical host encompasses several processes working on the same task, e.g., a text formatting task, which in turn corresponds to a single user job request at the command interpreter level. In object based systems, such as Eden, the natural unit of distribution is an object because objects neatly encapsulate the concepts of data and program and have well-defined interactions with other objects.

The unit of distribution chosen in various systems is shown in Table 3.1.

Table 3.1: Units of Distribution

System	**Unit of distribution**	**Granularity**
DEMOS/MP	Process	Coarse-grained
Distributed V	Logical hosts	Coarse-grained
Eden	Heavy-weight objects	Coarse-grained
Accent	Process	Coarse-grained
Emerald	Objects	Fine-grained

In Emerald, we have chosen the object as the unit of distribution and mobility. In contrast to Eden objects, Emerald objects are fine-grained. In Eden, every object is implemented in a separate address space and is consequently very large (the minimum size of an Eden object is about 100,000 bytes of program and data). In our approach, objects can be quite small (a few bytes).

3.3 Language Constructs for Mobility

This section justifies the basic distribution and mobility features that we chose to include in the Emerald language. When integrating a new concept into a programming language, it must be decided what new features are needed. What features are the basic ones? What level of abstraction should they be provided at? Obviously, it is desirable to have the most fundamental features present in the language itself. On the other hand, it is desirable to minimize the total number of language features. As Einstein noted: “Make things as simple as possible, but no simpler” (quoted in [Lampson 83]). There are three different ways of introducing new features into a programming language:

New language constructs

Fundamental features are best incorporated into the language itself by the introduction of new language primitives.

Standard objects

Supplementary features can be added by providing new system-defined objects. This means of system extension is a fundamental benefit of the object approach.[1]

Programmer defined

Other features can be left out of the language, if a programmer can construct them using other, more primitive, features.

As an example of how different features are used in programming languages, consider floating point operations in Algol 60. Addition is considered so fundamental that it is a defined by a syntactic construct in the language; the sine function is important enough to be included in the standard library; the factorial function is not provided because it is not used very often and can be programmed using other constructs.

The Emerald system differs from the surveyed systems in that we integrate operating system concepts into the programming language instead of providing system calls to support them. Besides providing the Emerald programmer with a nice conceptual framework for distributed programming, this integration enables the compiler to generate more efficient code.

[1] Unbeknownst to Emerald users, all the built-in types are written in Emerald (albeit most use “magic” keywords to make the compiler generate the correct machine instructions). For example, the built-in object **array** is written entirely in Emerald.

For example, the compiler can choose simpler, more efficient implementations for objects that do not use mobility.

3.3.1 Object Location

Because location is central to distribution, Emerald has a language construct for obtaining the current location of an object. Given an object X, "**locate** X" returns the node where the object X currently resides. At first glance, these semantics for the **locate** expression appear to be rather straightforward. However, because objects are mobile, the "current" location of an object can be constantly changing. The **locate** expression takes a non-zero time to execute and during this time, the object in question could move. We have chosen to specify that the **locate** expression evaluates to one of the nodes where the object was located *during* the execution of the expression. Note that "**locate self**" is the simplest way of obtaining a reference to one's own node (**self** returns a reference to the object within which **self** appears). Chapter 5 is devoted to a more detailed discussion of the semantic and implementation problems related to finding objects in a locally distributed system.

Permitting explicit control over object location requires that locations be explicitly represented. To the Emerald programmer a *location* is represented by a *node object* which encapsulates the concept of location. A node object is an abstraction of the underlying machine; in this context, the underlying machine is both the hardware and the software supporting Emerald at run-time, i.e., the Emerald kernel. Each node object represents exactly one node and implements a number of operations for accessing information about the state of the underlying machine and kernel, e.g., time-of-day, load average, node number, etc. In other systems, these operations would be special system calls; in Emerald, they are merely invocations on node objects. A list of available nodes may be obtained by invoking an operation on any node object. This can, for example, be used by a load balancing program to find out what nodes are available for use.

An advantage of our approach is that system resources are named and accessed in the same manner as everything else. For example, because many kernel operations are encapsulated in node objects, they may be invoked remotely. We have thereby achieved remote kernel calls, not by adding special code in the kernel for inter-kernel calls, but rather by using the general object mechanism.

3.3.2 Moving Objects

Previous systems have chosen to provide mobility through special system calls. There are several reasons for this. First, none of the systems has language support for mobility. With the exception of Eden, language independence is actually a design goal. Second, mobility is intended to be transparent; providing it through language constructs makes it visible.

In Emerald, we assign semantics to the concept of location and therefore provide mobility primitives as language constructs. The major advantage is one of efficiency because the compiler can provide simpler more efficient code for those objects that do not use mobility. In some cases, the compiler can also generate more efficient code for objects that do move by combining mobility with object invocation (section 3.6).

As in Eden, the location dependent primitives rely on the concept of *co-location.* Locations are specified by giving a reference to another object. The node implied is the one where the other object currently resides. To move an object X to the node where object Y resides, Emerald provides a move statement: "**move** X **to** Y". To move an object to a specific node, it suffices to specify co-location with the node object that represents the node. In many cases, it is desirable to move several objects at once. For example, when moving an array object, the individual array elements should probably also be moved. We have not provided a move statement for multiple object moves. Instead, such moves are handled using a more general mechanism (section 3.5.5).

3.3.3 Semantics of Move

In many cases, mobility is used for performance improvements. The actual location of any particular object is not important for the correct functioning of an application. For this reason, our **move** primitive has rather weak semantics. It is a hint; the underlying system is not obliged to perform the move and the object is not obliged to remain at the destination node. Conversely, the system is free to move an object. For example, objects entirely local to another object can be moved when that object is moved.

In other cases, it may be crucial for an application to control the location of objects. For example, Pu describes a scheme for maintaining high data availability by dynamically regenerating replicas lost due to node crashes [Pu 88]. In this scheme, it is a fundamental requirement that object replicas be placed on separate nodes. Obviously the replicas must not be involuntarily moved by the underlying system. For such applications, we provide a language primitive to *lock* objects to a particular node so that they cannot be moved. The statement "**fix** X **at** Y" locks the object X to the node where Y currently resides. If necessary, the object is moved first. In this case, the move is not a hint; the object always moves. An **unfix** operation is provided, so that objects can become mobile again. Fixed objects are not allowed to move; any move request for the object will be ignored. Attempts to fix an object that has already been fixed will fail. To enable such refixing to be performed, we provide a **refix** statement which performs an **unfix** and a **fix** atomically. **Refix** is atomic to prevent an object from being moved after the **unfix** but before the **fix**. Applications that require strict control over object location, such as the replication scheme mentioned above, can reliably move objects using **fix** and **refix**. Note that Emerald does not provide protection: any object may be moved (or refixed) by anyone having a reference to it. We assume that the application knows what it is doing.

3.3.4 Residual Dependencies

In some systems, after an object or process has moved away from a node, some of its state may remain behind so that the object remains dependent on the source node. In section 3.1.1, we called such dependencies *residual dependencies*. They can take many forms, some of which are implementation-specific. We call a residual dependency *inherent* if it is an unavoidable part of the problem being solved by the program. For example, direct access to an I/O device produces an inherent residual dependency and may prevent a process from being moved at all; some systems provide access to a graphics terminal by allowing processes to write large bitmaps directly to the device controller. If a process has direct access to I/O devices, it cannot be moved. All the surveyed systems avoid such problems by letting I/O be accessed in the same migration-independent manner as processes. The inherent residual dependency on the I/O device is still present but at least the process can be moved. Similarly, in Emerald all I/O is provided through (special system) objects, so user objects can freely move despite I/O dependencies. The implementation of mobility may also cause invisible residual dependencies. For example, in the Sprite system [Douglis 87], a moved process must retain access to the node where it was created because the original node maintains control over the process.

Residual dependencies are undesirable because they may cause a process to fail if a seemingly unrelated node fails. They can also introduce performance problems. For example, when a process is migrated to lessen the load on a node, residual dependencies may still cause the source node to carry some of the burden of the process, e.g., when the process writes to a bit-map display on the source. The Accent migration solution typically leaves such residual dependencies because a major idea behind the solution is that of lazy migration of processes [Zayas 87a]. Only the working set of pages for a process is migrated immediately—the rest of the pages remain on the original host. However, as mentioned by Zayas [Zayas 87b], the impact of this dependency is minimal when the pages are stored on a common file server. DEMOS/MP has residual dependencies in the form of forwarding addresses and includes a mechanism to remove them; this is studied in more detail in section 5.2.

In Emerald, we have reduced the problem of residual dependencies to that inherent in any migration system. Any migrated object is obviously dependent on communication partners that remain on the original host. Emerald supports fine-grained mobility which can lead to more residual dependency problems than in coarse-grained systems because fine-grained objects are much more likely to be moved away from communication partners. Emerald uses the concept of *hierarchical groups* to avoid separating closely associated objects. The next section formulates the problem more carefully and subsequent sections show how the hierarchical group concept is developed.

3.4 The Group Problem

An important issue when moving objects containing references to other objects is deciding how much to move [Sollins 79]. In systems with coarse-grained mobility, the decision is easy because the unit of mobility is entire, self-contained processes or programs. In object-based programming, objects are composed of references to other objects. The presence of fine-grained mobility implies that closely related objects can easily become separated. When moving an object, it may therefore be desirable to move other (related) objects. We formulate the *group problem* as follows: "What objects should be grouped together for the purpose of mobility?"

Every object is part of a graph of references where the nodes in the graph are objects and there is an arc for each reference from one object to another. The group problem can be restated as a graph problem: when moving an object, should only the object itself, a part of the graph, or the entire graph be moved? In the following, we discuss solutions to the group problem and mechanisms for defining groups.

The simplest approach—moving single objects alone—may be inappropriate. If the object belongs to a set of related objects that frequently communicate and operate together as a group, then future communication between the objects may require a significant number of remote accesses that would have been avoided if the other objects had been moved as well. Large performance gains can be achieved by avoiding remote invocations because local access is about three orders of magnitude faster than remote access. (For Emerald performance details, see Chapter 7.) Conversely, large performance penalties may be incurred if frequently communicating objects are inadvertently separated. For example, subcomponents of an object that are referenced only by the object should be moved along with the object. Consider the program to visit all nodes shown in Figure 2.2 on page 25. When the process moves to another node, the list *myList* should be moved because it is local to the process. Furthermore, each element in *myList* should also be moved. If the list and its elements were left behind, all future accesses would be remote. In Chapter 7 we present performance data showing that a small data object can be moved in about the same time it takes to access it *once*.

We conclude that it is important to keep related groups of objects together to get a high degree of locality of reference. The question is now whether the application programmer must provide such grouping information or whether the system can deduce the needed information.

In general, it is difficult to determine which objects should be kept together because such determination remains application-specific. Consider the Emerald mail system (which is based on the Eden mail system [Almes 84]). Mailbox objects contain references to the mail message objects that have been sent to it. A mail message has references to the text of the message, the date sent, a mailbox to which replies may be sent, and a reference to the mailbox to which the message has been sent. The following Emerald code shows the declarations of the fields in a mail message.

```
var to, from: MailBox
var text: String
var status: MessageStatus
var date: Time
```

When a mail message is moved, the message *text*, the message *status*, and the *date* should also be moved because they are local to the mail message object. The reply mailbox (the *from* field) should probably not be moved because it presumably will be invoked only once by the reader of the message while the mailbox owner will invoke it frequently. Obviously, the destination mailbox (the *to* field) should not be moved. In Figure 3.1, we have marked the references to those objects that must move when a mail message is moved. Similarly, when a mailbox moves, its mail messages should follow. Consequently, we have also marked the references originating in the mailbox object.

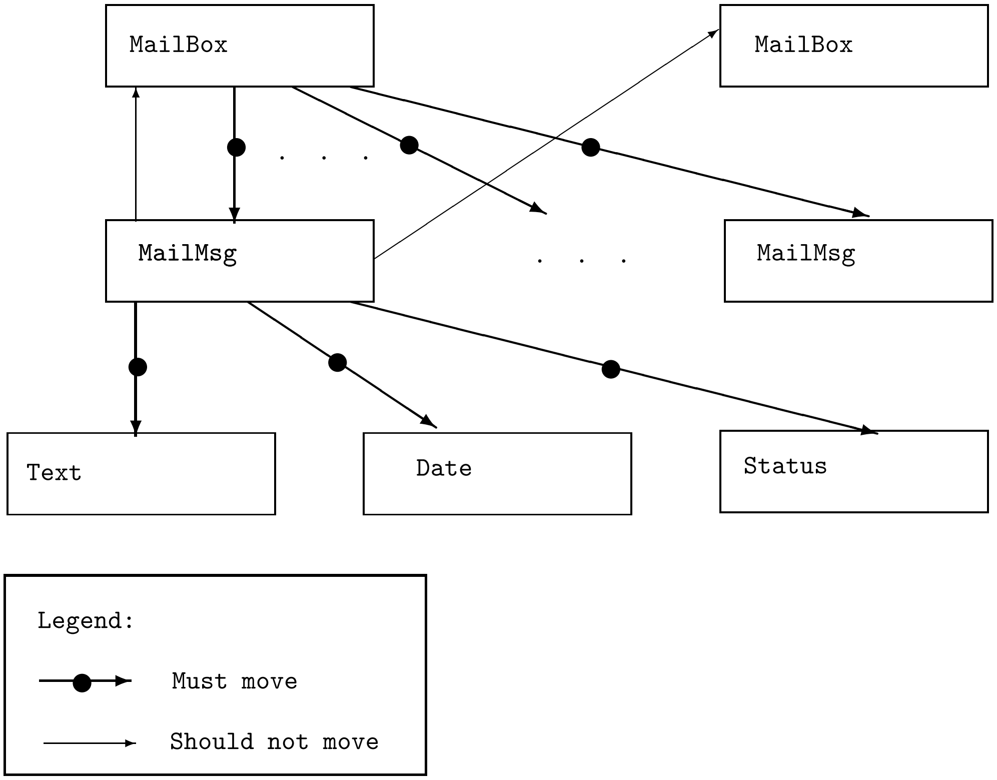

Figure 3.1: Mail Example Reference Graph

In Smalltalk [Goldberg 83], a problem analogous to the group problem arises when copying an object: should the subcomponents of a copied object be copied or shared? Unfortunately, there is no intrinsic way for the Smalltalk system to decide this question because it is inherently application-dependent. For this reason Smalltalk's Class **Object** supports three different copying operations: *shallowCopy*, *deepCopy*, and *copy*. Shallow copy copies the object including its references but does not make a copy of the referenced objects—these are shared with the original object. Deep copy recursively copies the referenced objects so that they are not shared. The third form of copy defaults to shallow copy but is designed to be overridden by those subclasses that desire their own combination of shallow and deep copy. In this way, the decision of what to copy can optionally be left to the application programmer (see Chapter 6 of [Goldberg 83]).

Before considering how to solve the group problem we note that there are also performance reasons for wanting to move more than one object at a time. Consider the following program that moves an array and its elements.

```
$i \leftarrow 1$
loop
    exit when $i = n$
    move $A(i)$ to remoteNode
    $i \leftarrow i + 1$
end loop
move $A$ to remoteNode
```

The loop performs n moves one at a time. Each move requires at least one network packet. If the elements were moved all at once and they are relatively small (on the order of 100 bytes) the move can be performed with considerably fewer network packets. Moving all the array elements together would enable the run-time kernel to pack them into as few network packets as possible, potentially reducing the total number of packets significantly. Our experience with the Eden system indicates that the performance of remote invocations is often a linear function of the number of network packets sent. Thus, reducing the number of packets can lead to significant performance improvements.

After considering the Eden Mail example, the problem of copying in Smalltalk, and other similar problems, we conclude that it is necessary for programmers of distributed applications to explicitly indicate what objects are to be grouped with respect to mobility. How such groups should be indicated is the subject of the following sections.

3.5 Solving the Group Problem

Solutions to the group problem fall into two general categories:

Implicit methods do not require any additional information from the programmer. The compiler (or the run-time kernel) can derive group information directly from the program. Implicit methods include compiler decisions based on the static program text and dynamic methods where the run-time system predicts future patterns of locality of reference and uses them to make mobility decisions.

Explicit methods require the programmer to specify explicitly what objects are to be grouped together. Explicit methods include imperative methods where the programmer states what objects are to be grouped together and dynamic binding methods where objects are grouped together using run-time system calls.

In the previous section, we concluded that it is necessary to provide explicit methods. Implicit methods are still of interest because they can ease the tedious work of ensuring that all objects that should move actually do so. In the following, we consider three implicit methods and three explicit methods.

3.5.1 Compiler Determined Groups

In general, the compiler cannot derive the usage pattern of an object merely from the static program text. However, for some object references, it is obvious that the referenced object should be moved along with the object containing the reference. Local objects cannot be referenced outside their containing object. If these local objects are not moved when the containing object is moved then all subsequent accesses will be remote, resulting in serious performance degradation. Usually it does not make sense to separate an object from its own *local* objects. We therefore argue that local objects should always be moved with their containing object.

In Emerald, the mobility primitives are part of the language and it is possible for the compiler to detect what objects are purely local. Such local objects are implemented in a simpler, more efficient form and are always moved when the containing object is moved. The benefits of detecting that objects are local are considerable [Hutchinson 87a].

3.5.2 Groups Based on Behavior

Another possibility is that the run-time kernel keep track of usage patterns and use these as a prediction of future behavior. Based on the predictions, the run-time kernel could co-locate objects to reduce the number of remote references. This approach has several drawbacks. First, past behavior is not necessarily a good predictor of future behavior, for example, bimodal behavior might lead to predictions that would cause worst case performance. Second, monitoring past behavior requires extensive monitoring of not only remote invocations, where the overhead would be relatively low, but also of local invocations, where the overhead would be substantial. The cost in run-time overhead and extra storage alone seems to rule out such monitoring.

3.5.3 Immutable Objects and Copying

A fundamental benefit of immutable objects is that they can be freely copied and any one of the copies can be used. For immutable objects that are not very large, it almost always pays to replicate the object when it is passed as a parameter in a remote invocation. Our measurements show that when performing a remote invocation a parameter object of size 1000 bytes can be copied faster than a *single* remote invocation of the parameter can be performed (see Table 7.6). The reason is that the parameter object can be piggybacked onto the network packet that contains the invocation request. This means that unless an immutable object is very large, it should always be copied.

In our implementation of Emerald, immutable objects are always copied. We see three reasons for copying immutables. First and foremost, not copying them would contradict our semantics for immutable objects because we define immutable objects to be omnipresent. If immutable objects were not copied when moved then a supposedly omnipresent immutable object would become inaccessible should its node crash. Second, our preliminary measurements of the system indicate that most immutable objects are quite small and so are inexpensive to copy compared to accessing them remotely. Third, by copying immutable objects, we can implement them more efficiently because they are always node-local. The compiler generates efficient calling sequences for immutable objects because the invocation can always be performed locally.

3.5.4 Group Objects

The first of the explicit methods that we consider involves explicitly representing groups by special *group* objects. The Argus system [Liskov 84, Liskov 88] has two different types of objects: movable guardian objects and non-movable local data objects. The purpose of providing guardian objects is to encapsulate related CLU objects for the purpose of providing resilient objects—objects that survive node crashes. Guardians are also the unit of mobility in Argus. Thus it is the application programmer who statically decides what objects are to be grouped together with respect to mobility.

Another solution would let the programmer construct a group by placing a reference to each group member in a special group object. The **move** statement would then take such an object as an operand and would move the objects referenced in the set. In this way, the programmer can dynamically build move groups by inserting and deleting object references from group objects.

In our quest for a uniform object view, special objects are undesirable and can be avoided as follows. Instead of using special group objects, every object can be made to function as a group object. Every object would uniquely determine a group which initially is empty. Two special primitives, **join** and **leave**, are used to add and delete group members. The statement

join A **to** B

would cause object A to join the group identified by object B. The statement

let A **leave** B

would remove object A from the group identified by object B. An object could be a member of any number of groups and would be moved any time one of the groups moves. We call this scheme the *join-leave* method. A minor variant is to permit two different move statements: one for moving the object only and one for moving the group associated with the object (corresponding to *copy1* and *copy-full* in [Sollins 79], and to Smalltalk's *shallowCopy* and *deepCopy*).

Reconsider the mail message code on page 39. The basic idea behind the join-leave method is that the group designated by the mail message contains references to those objects that must move when the mail message object moves. In the reference graph shown in Figure 3.1 (on page 39), these references are specially marked. The programmer builds the appropriate move groups by inserting the marked arcs into the groups. Graphically speaking, the **join** statement is used to mark arcs and the **leave** statement is used to unmark arcs.

The join-leave grouping method is quite powerful. The programmer is free to dynamically establish and change groups as requirements change. However, this method has a major drawback which we illustrate using the mail message example from above. Assume that when a mailbox object is moved, its mail messages should also be moved. This can be achieved by including the mail messages in the group identified by the mailbox object itself. However, this is not sufficient because each component object of the mail messages must also be moved. Unfortunately, data abstraction dictates that the mailbox has no knowledge of the implementation of the mail messages. Seen from a mailbox's point of view, the mail message is a single object and one should be able to move it without having to explicitly move its individual component objects or even have knowledge of their existence. Emerald uses abstract types, so there is no way of knowing exactly how the mail message is actually implemented; data abstraction hides the details. Indeed, it is possible to have several different implementations of mail messages with differing data structures and co-location requirements. Therefore, the mailbox object cannot include the component objects of mail messages in the mailbox group. In general, this drawback can be overcome by having each object add its component objects to its own group, although we thereby need to add extra code to most objects (code which incurs a run-time cost).

A second drawback of the join-leave grouping method is that the construction of groups is done by executing a statement. This requires extra overhead both in execution time and in storage needed to represent the groups. This is especially undesirable when many small data-only objects are to be included in a group because the extra overhead would be significant compared to operations on the objects In the following section, we discuss a variation upon the join-leave method that reduces the run-time overhead significantly.

3.5.5 Attachment

The Emerald solution to the group problem is based on the same idea as the join-leave method. The purpose of group objects is to indicate to the run-time kernel which of the referenced objects are to be recursively moved when an object is moved. The indication is explicitly set and removed for every object entering or leaving the move group. Instead, our idea is to statically indicate which *variables* hold references to objects in the move group. When declaring a variable, the programmer can specify that it should be an *attached* variable. For example, the declarations in *MailMsg* (see page 39) can be modified as follows:

var *to, from*: *MailBox*
attached var *text*: **String**
attached var *status*: *MessageStatus*
attached var *date*: *Time*

When the *MailMsg* object is moved, the objects named *at that time* by *text*, *status*, and *date* will be moved with it, while neither of the mailboxes named by *to* or *from* will be moved. Let mailboxes store their references to their mail messages in attached variables. When a mailbox is moved, its messages and—recursively—their contents are also moved. Graphically speaking, instead of marking the references (as for move groups), we mark the slots in the object that contain the references. Figure 3.2 is a modified version of Figure 3.1 where attached variables are marked. In this way, we avoid having to dynamically execute statements to mark or

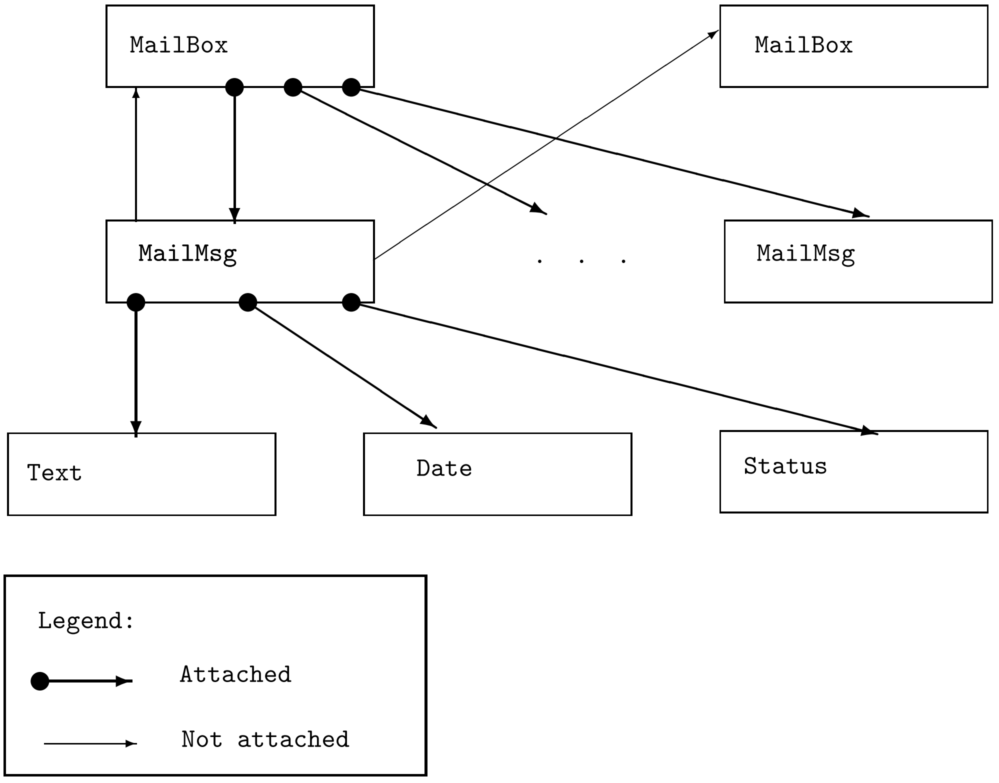

Figure 3.2: Mail Example Graph showing Attached References

unmark a reference. Instead, a reference is marked by assigning the referenced object to an attached variable.

Attachment is transitive. For example, when the top left mailbox in Figure 3.2 is moved, any objects attached to the object named by *text*, *status*, or *date* will also be moved. Attachment is not symmetric; a mail message can be moved without its mailbox being moved (and because the *to* field is not attached). The attachment method is as powerful as the join-leave method because the join-leave method can be simulated as follows. The effect of a **join** can be achieved by declaring an attached variable in the group object and assigning the object in question to it. Assigning **nil** to the attached variable effects a **leave**. In general, a move group can be represented by an attached set where each of the set members are themselves **attached** to the set.

In terms of performance, the biggest win is that attachment is declarative. It avoids run-time overhead in most cases because objects joining a group are usually assigned to a variable. The attached variables are declared statically which allows the compiler to provide the attachment information to the run-time kernel without imposing *any* run-time overhead.

Attachment can be used to provide *hierarchical grouping* of objects. In Figure 3.2 there are three levels of objects: mailboxes, mail messages, and mail message components. A mail message component, e.g., *status*, can be moved independently of the rest of the objects. When a mail message moves, its attached components follow. When a mailbox moves, the underlying hierarchy of mail message objects also moves. An advantage of this approach is that it allows an object and its underlying hierarchy to be moved without knowledge of the underlying hierarchy.

3.6 Parameter Passing Semantics

An important issue in the design of distributed, object-based systems (as well as remote procedure call systems) is the choice of parameter passing semantics. In an object-based system, all variables refer to other objects. The natural parameter passing method is therefore *call-by-object-reference*, and this is in fact the semantics chosen for Smalltalk [Goldberg 83] and for CLU [Liskov 79] (where it is called *call-by-sharing*). Other systems, e.g., Argus, require that parameters to remote calls be passed by value, not by object-reference [Herlihy 82]. Remote procedure call systems typically require call-by-value because addresses are context-dependent and have no meaning in the remote environment [Birrell 84].

The Emerald language uses call-by-object-reference parameter passing semantics for all invocations, local or remote. It is important to use the same semantics for both because one of our principal goals has been to promote a single object model where there is no distinction between local and remote objects. In fact, invocations may start as local and later become remote should one of the participating objects move. Choosing call-by-object-reference can lead to serious performance problems: on a remote invocation, access by the remote operation to its parameters is likely to cause additional remote invocations because the parameter objects remain at the caller's node. This is illustrated by the following executable Emerald program:

```
const ArgType == type T
   operation G
end T
const B == object B
   export G
   operation G
   end G
end B
const X == object X
   export F
   operation F[arg: ArgType]
      arg.G
      arg.G
   end F
end X
const A == object A
   process
      X.F[B]
   end process
end A
```

Object A invokes operation F on object X passing object B as a parameter. Subsequently, when F invokes the operation G on its argument, B, we say that F executes a *call-back* to access its argument. Assume that objects A and B are on node 1 and object X is on node 2. This situation is depicted in Figure 3.3. In the figure, two call-backs are shown. If the object B is moved to node 2 before the invocation $X.F[B]$ then the call-backs are avoided. Figure 3.4 shows the situation, where B is moved to X's node before the first invocation. Note that the two remote invocations of B have become local invocations.

In Emerald, we use two mechanisms to avoid performance degrading call-backs: immutable objects and mobility. When parameter objects are immutable, they can be freely copied. Many small objects such as integers, Booleans, text strings, and other built-in types are immutable. Obviously, it does not pay to remotely access such objects because they can be copied cheaply. As described in Section 3.5.3, we have chosen always to copy immutable objects, thus avoiding any remote access to them. This corresponds to using call-by-value at the implementation level, while retaining call-by-object-reference semantics at the language level. Because Emerald objects are mobile, it is possible to avoid many call-backs by moving parameter objects to the destination node of a remote invocation (shown in Figures 3.3 and 3.4). Whether this is worthwhile depends on (1) the size of the parameter object, (2) other current or future invocations of the parameter, (3) the number of invocations that will be issued by the remote object to the parameter, and (4) the relative costs of mobility and local and remote invocation. Note that it is entirely possible that neither the caller nor the callee has sufficient

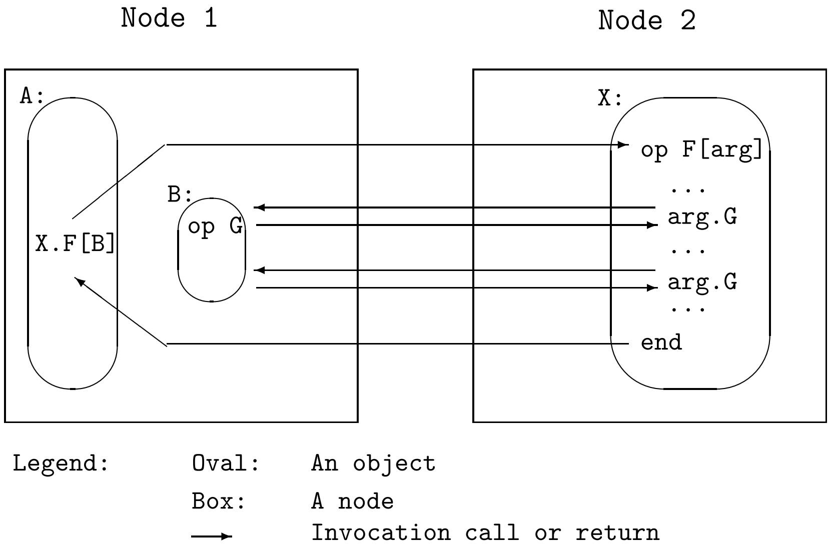

Figure 3.3: The Parameter Call-back Problem

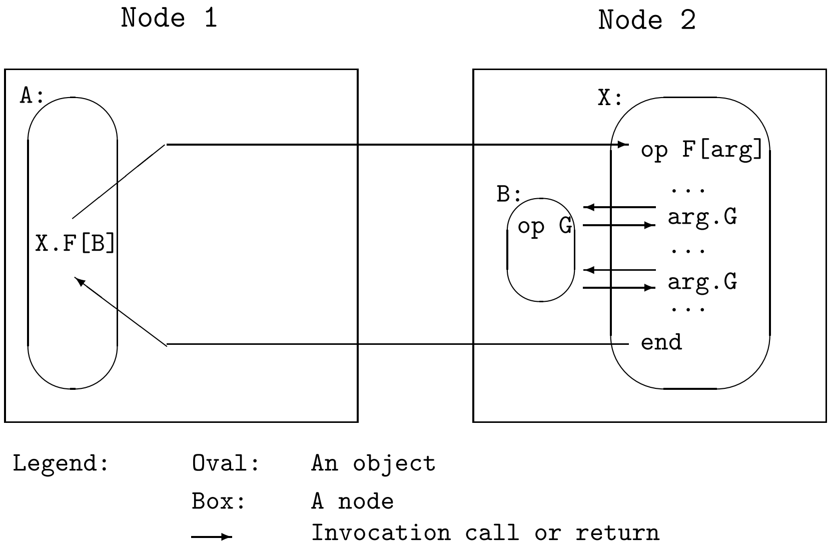

Figure 3.4: Avoiding the Call-back Problem

information to make decisions concerning parameter mobility. The Emerald programmer may decide that a parameter object should be moved based on knowledge about the application. The programmer is free to move parameters at any time before or after the call. In the above example, the caller could move the object B to node 2 before performing the call.

To make parameter mobility more efficient, Emerald provides the three parameter passing modes call-by-move, call-by-visit, and call-by-move-return (see 2.4.2). Move parameter modes are similar to attachment. Informally, we say that call-by-visit parameters are attached to the invocation. The modes are specified by prefixing the keyword **move** (or **visit**) to the actual parameter, e.g., in Figure 3.3 “$X.F[B]$” becomes “$X.F[$**move** $B]$”. Point-of-call specification has the advantage that the call-by-move modes can be used when invoking objects that are ignorant of distribution. Alternatively, the parameter passing mode could be specified with the formal parameter. This could require the run-time system to know the parameter passing mode before issuing the call. Because of abstract typing the run-time system does not know the parameter passing mode for the target object and must query it to obtain the mode. Obtaining the information from the target before making the call would not only nullify the performance advantage but would add considerable extra overhead to *every* invocation. This overhead could be avoided, if references to objects would carry the call-by-move information for each of the operations of the referenced object. This can be done but would require yet another kernel data structure to be passed along with references.

Another alternative is to specify the parameter passing mode in the abstract type. There are two reasons why we did not allow this. First, move is fundamentally a performance optimization and is entirely unrelated to abstraction. The decision to use call-by-move is usually based on the size of the parameter object and on its future use—information that is not available when defining the abstract type. Second, if the mode is specified in the abstract type then the decision whether or not to use call-by-move is static and cannot be made at run-time (unless each operation had two versions, one with call-by-move and one without).

Using point-of-call specification, the Emerald programmer can choose between parameter passing modes by executing an appropriate invocation. The choice can be made dynamically, for example, by using the following code:

```
if wantCallByMove then
    X.F[move B]
else
    X.F[B]
end if
```

The return-by-move parameter passing mode is specified at the callee’s node by adding the keyword **move** to the specification of the return parameter; we say that the result is attached to the return. A usable alternative would have been to specify it at the point-of-call.

Call-by-move is a convenience and a performance optimization. Both caller and callee are free to move parameter objects at any time, so the call-by-move modes are not necessary.

However, providing call-by-move as a parameter passing mode allows packaging of the parameter objects in the same network packet as the invocation message. The resulting performance gains are discussed in section 7.4.1. Furthermore, call-by-return and call-by-visit obviate the need for the invoked operation to know the location to which results should be returned. If a result is to be explicitly moved, such knowledge is necessary.

3.7 Mobility Summary

In this chapter, we have examined several earlier process migration systems and used them to shed light on the design issues concerning mobility. Emerald differs from these systems in two ways: It has fine-grained mobility and it supports mobility at the language level. We have discussed the inherent conflicts that arise when attempting to provide distribution transparency and visibility in the same system. We have presented the concept of attached variables to enable application programmers to specify groups of objects that are to migrate together. Our solution to the call-back problem leads us to introduce a new kind of parameter passing mode, namely call-by-move. In the next three chapters, we show how to implement a kernel supporting fine-grained object mobility.

Chapter 4

Implementation of a Kernel Supporting Mobile Objects

In this chapter, we describe the implementation of the Emerald kernel and how it supports mobile objects. In our implementation, we have emphasized three major areas: distribution, fine-grained mobility, and node-local performance. Before considering the implementation problems derived from each of these three areas, we discuss the main differences between previous distributed systems and Emerald. Thereafter we present the main implementation problems and distinguish which the Emerald kernel itself solves and which are solved by building the kernel on top of UNIX. The remainder of the chapter is devoted to detailed solutions to the most interesting of the kernel implementation problems. First, object implementation, including how objects are allocated, dynamic linking and loading of code, and how both intra-node and inter-node addressing is done. Second, we describe process implementation and how processes can span several nodes. Third, we describe how objects and processes are moved. We conclude with a number of minor implementation issues. Two problems are worthy of separate discussion and are relegated to later chapters. Chapter 5 discusses the problem of keeping track of mobile objects and Chapter 6 that of distributed garbage collection.

4.1 Implementation Problems

In the following, we discuss the main differences between previous systems and the Emerald prototype. The most important of these are fine-grained mobility and the emphasis on node-local performance.

4.1.1 Coarse-grained versus Fine-grained Systems

Previous object-oriented systems have fallen into one of two categories: single machine systems, such as Smalltalk, and distributed systems, such as Eden. Single machine systems provide fine-grained objects and have the potential to be reasonably efficient when all objects are implemented within a single address space. Distributed object-oriented systems provide coarse-grained objects that may be distributed across a network of machines. However, distribution is often bought at the expense of node-local performance. For example, Eden suffers from poor performance for objects that are on the same machine. We see two reasons for this. First, in Eden the full invocation mechanism is used even for local invocations. Second, the full mechanism requires multiple crossings of process protection boundaries. Emerald is an attempt to combine the advantages of these two types of systems by adding the fine-grained, fairly efficient object concept of Smalltalk[1] to the concept of distribution as present in Eden. Consequently, we derive implementation problems from both types of systems, in addition to the problems caused by merging their concepts. A major goal has been to provide light-weight distribution without sacrificing node-local performance. We achieve this by integrating distribution concepts into our object-oriented language and having the compiler generate efficient code for node-local operations.

In earlier distributed systems, the implementation of mobility is simplified by making the unit of mobility entire process address spaces. The advantage is that an address space encapsulates the code, the thread of control, and the data used by a process. When an entire address space is moved, it is not necessary to modify its contents; all addresses remain the same. Only references to entities outside the address space need be translated, e.g., IPC addresses. To avoid translating any addresses in the address space, most process migration systems restrict communication across address space boundaries to a well-defined IPC mechanism (e.g., message passing as in DEMOS/MP). Such mechanisms introduce a level of indirection in inter-process communication. Instead of giving a process the address of a communication partner, the kernel gives the process a *handle* that indirectly refers to the communication partner. For example, the handle can be an index into a kernel data structure containing the actual address, or it can be some form of network-wide identification. Such indirection in naming simplifies the implementation of mobility. When an object (or process) is moved, it is not necessary to change the object's references to other objects—only the destination kernel's handle data structures need be updated. Also, other objects' references to the moved object do not need to be changed—only the content of the handle data structure for the moved object must be modified. In a distributed object-based system, a similar encapsulation technique can be used by choosing the object as the unit of mobility. Objects cleanly define the boundaries of all system entities. Furthermore, because all resources are objects, addressing is standardized and object references can be used as handles. In Eden, the unit of mobility is the object, but objects are coarse-grained (each is a UNIX process).

Although choosing a large unit of mobility can simplify implementation, it has several disadvantages. First, mobility is coarse-grained so only entire address spaces can be moved.

[1] Not to say that Smalltalk is efficient; it has a number of efficiency problems of its own, see [Krasner 83]. The point is that Smalltalk is reasonably efficient because all objects reside in the same address space.

Second, there are efficiency problems because inter-process communication is forced to cross heavy-weight protection boundaries.

As opposed to other distributed object-oriented systems (e.g., Eden), Emerald objects do not have their own address space. Instead, Emerald is a single address space system; all objects—and the Emerald kernel—share an address space. The object is the unit of mobility as in Eden. Instead of moving an entire address space, Emerald moves only a small part of an address space. When the Emerald kernel moves an object, it is carved out of the kernel's address space and sent to another node's kernel, where it is integrated into that kernel's address space. There are two advantages over earlier systems. First, node-local invocation can be performed without using the full general object invocation mechanism. Second, only remote invocations must cross heavy-weight boundaries. Invocations of objects on the same node can therefore be performed directly by compiled code without kernel intervention, greatly increasing efficiency over earlier systems. A disadvantage of our approach is that it is necessary to perform translation of the addresses in a moved object because there is no common address space across nodes. An alternative being considered in the Amber system is to establish a common address space across all nodes, for example, by letting each node "own" a part of the address space [Levy 88]. Upon creation, objects would be allocated in the creating node's part of the address space and would reside at the same virtual address on all machines. Such an approach limits the maximum number of nodes (or the maximum number of objects) and requires the nodes to have compatible virtual memory systems. However, it may be a viable solution, especially if it is integrated with virtual memory management routines.

4.1.2 A Summary of the Main Implementation Problems

The Emerald prototype emphasizes three areas: distribution, fine-grained mobility, and node-local performance. Each of these presents a set of implementation problems which are discussed in the following.

Emerald is distributed and so we must provide a network-wide naming scheme for objects. We must also provide a suitable IPC mechanism which, due to our uniform object model, must be a transparent remote procedure call facility. To enable similar objects on different machines to have the same code, we need to provide for the distribution of code within the network. Distribution also complicates object storage. We also need facilities to handle failures caused by node crashes.

Emerald objects are fine-grained and mobile, so we must be able to extract their state from the kernel address space, move them, and subsequently integrate them into the destination kernel's address space. This causes several problems related to addressing objects efficiently and related to repairing the damage done by extracting an object from an address space. Such repair requires that all affected pointers can be found and modified. To reduce storage for the executable code in objects, we provide a code sharing facility. Because objects can move, we need to be able to locate them when they are accessed. Mobility also complicates garbage collection although simple and effective solutions are available.

We have emphasized node-local performance and have therefore put considerable effort into ensuring that the presence of mobility does not degrade local performance. In traditional remote procedure call systems (e.g., as described by Birrell and Nelson [Birrell 84]), the user calls a stub procedure that either performs a local or a remote call. Such indirection is not efficient when the call turns out to be local. We therefore investigate the problem of providing efficient local calls while retaining the ability to do remote calls. The most general object implementation provides flexibility for distributed operations but is not efficient for node-local operations, so we attempt to optimize the implementation of objects that are known to be local. We have also improved local performance by extensive use of the concept of immutable objects. When moving an object, the pointers in it must be identified. The standard Smalltalk-80 implementation [Goldberg 83] uses tag bits in the pointers themselves, which induces some run-time overhead. We investigate alternative solutions that have no run-time overhead.

Emerald supports failure handlers. In some systems, the presence of failure handlers introduces extra run-time overhead. We investigate how such overhead can be avoided.

We also discuss several minor implementation issues such as how to provide pseudo-concurrency, how to handle I/O, etc. We then preview the techniques used to attack these implementation problems.

4.1.3 Techniques for Performance

In our design we emphasize performance because we believe that good performance cannot successfully be retrofitted. The following list previews the main techniques used to achieve good intra-node performance in the Emerald prototype. As we discuss the implementation and its problems, we expand on each of these techniques as appropriate.

Shared address space

To meet our goal of building a distributed object-based system with efficient local execution, the Emerald prototype kernel relies heavily on shared memory. The Emerald kernel and all objects on a node are stored within a single address space. As a consequence, most inter-object invocations do not have to cross address space boundaries and kernel calls can be executed without the overhead of an address space switch. A disadvantage is the lack of protection.

Protection by the compiler

Because kernel code and data are in the same address space as user code, we rely extensively on the compiler to produce trustworthy code. The compiler is allowed to produce code that directly operates on kernel data structures, avoiding expensive kernel calls. Compile-time checks also eliminate much run-time overhead. As in Concurrent Pascal, the only run-time checks are for array bounds and for **nil** references. In our current prototype, explicit **nil** checks have been avoided by choosing **nil** to be an invalid virtual address and relying on the hardware to produce an invalid-address fault.

Hard pointers

The use of *hard pointers* (by which we mean virtual memory addresses that are directly interpretable by hardware) enables the compiler to generate more efficient code than, for example, the standard Smalltalk-80 implementation [Goldberg 83] because in most cases indirect references through an indexed object table can be avoided. The disadvantage is that hard pointers must be translated when crossing address space boundaries.

No call-time-binding

Because Emerald is strongly typed, we can generally use static binding and use dynamic binding only when explicitly requested to do so. We entirely avoid binding on invocations (dynamic binding is done when variables are assigned—not when they are used).

Multiple implementations

Although there is only a single model of an object at the language level, there are several possible implementations at the kernel level. By choosing the least expensive implementation at compile time, we avoid using the relatively expensive general object implementation technique when it is not necessary. Hutchinson describes several experiments showing that less than one fifth of all objects need be of the most general type [Hutchinson 87a].

Compiler tables

The compiler produces numerous tables enabling the kernel to obtain information that would otherwise need to be maintained dynamically. For example, the concept of a current exception handler is supported at zero run-time cost through the use of compiler generated tables.

4.1.4 The Kernel and UNIX

The Emerald kernel runs on top of ULTRIX-32 (Digital's version of Berkeley UNIX). We have used UNIX because it provides a convenient substrate upon which to build an experimental system. On each node in the network, there is a single UNIX process containing an Emerald kernel written in C. All Emerald activities on a single node take place within this process—the kernel process. Emerald processes are implemented by light-weight processes within the kernel's address space. The kernel time-multiplexes its own activities with the light-weight processes to achieve pseudo-concurrent execution of the Emerald processes. The integrity of the kernel and protection of objects is guaranteed by the compiler through type checking and run-time array bounds checks. The UNIX process that executes the Emerald kernel is considered to be a bare machine with some convenient features. This simplifies the implementation because many low-level problems are avoided. Furthermore, by using UNIX and programming the kernel and compiler in C, the system can easily be ported to other machines. For example, the Emerald kernel was ported to a SUN system in a day. (Porting the compiler took four weeks due to the different hardware instruction and register sets which required rewriting of the code generator.)

UNIX solves many low-level problems. Bootstrap is trivial because the kernel is merely a normal UNIX user program—the kernel is started as any other UNIX program. I/O is greatly simplified because the Emerald kernel does not have to deal with actual hardware devices. We use TCP/IP connections for communication with the kernel (more specifically for Emerald user I/O and for test output from the kernel) and UDP datagrams for inter-kernel communication. Code files for Emerald objects are stored in the UNIX file system in standard UNIX ".o" format and are loaded into the kernel address space using UNIX *read* system calls. We rely on UNIX to supply the Emerald kernel with a large chunk of memory. The presence of virtual memory is not vital; it merely allows the kernel to use more memory than is available as physical memory. Finally, building a prototype on top of UNIX is significantly eased by useful tools such as the interactive, symbolic debugger, *dbx*, the profiling facilities, *gprof*, and by the ability to reliably produce kernel test output (by calling UNIX I/O system calls).

The Emerald kernel itself implements a number of low-level facilities for its own use. These facilities are general in nature and are not specifically related to Emerald. Similar facilities are found in any UNIX program that supports light-weight processes, is time-dependent, uses asynchronous I/O, and is to be dynamically monitored. Several of the modules are modified versions of the corresponding modules in the Eden kernel.

- We use a memory allocator written by Guy Almes. It is a version of QuickFit developed by C. Weinstock at Carnegie-Mellon University. Memory deallocation is extensively discussed in Chapter 6.

- A simple task management system that allows kernel tasks to be scheduled for execution and to execute sequentially in a coroutine-like manner. These routines were developed by G. Mager and J. Sanislo for use in Eden.

- Time handling routines to schedule tasks at given times, e.g., timeout handlers for network operations. These routines were originally written by S. Cady for use in Eden.

- Asynchronous I/O management is accomplished by multiplexing input and output to and from UNIX sockets.

- Communication between kernels uses a sliding window protocol [Tanenbaum 81] that provides reliable message passing based on unreliable UDP datagrams [Jul 84]. The protocol includes facilities for declaring communication partners dead when they have not responded after a certain number of retransmissions.

- Extensive kernel trace and snapshot facilities allow the state of the kernel to be dynamically displayed [Jul 88a].

4.2 Object Implementation

This section describes how the Emerald kernel supports objects. First, we describe how objects are stored in memory and how they are referenced locally and remotely. Second, the implementation of processes is presented. Third, we show how objects, processes, monitors, and conditions are moved. Fourth, we discuss several minor implementation problems.

4.2.1 Object Representation

All objects are programmed using a single Emerald object definition mechanism. However, not all objects are implemented using the same mechanism. The reason is that many objects do not require the generality and flexibility allowed by our object model. Where the compiler can detect that an object does not require the full generality, it is free to choose whatever representation is best. For each object representation, the compiler chooses an appropriate addressing mechanism, storage strategy, and invocation protocol [Hutchinson 87a]. We use four different object representations:

- A *global* object can be moved independently, can be referenced globally in the network, and can be invoked by objects not known at compile time. Global objects are heap allocated. An invocation of such an object may require a remote procedure call.

- A *local* object is completely contained within another object; that is, a reference to the local object is never exported outside the boundary of the enclosing object. Such objects cannot move independently; they always move along with their enclosing object. Local objects are heap allocated. An invocation is implemented by a local procedure call or directly by inline code.

- An *immutable* object is handled almost identically to a local object. Immutable objects are defined to be omnipresent. When moved, their implementation is not moved; instead, a copy is made. This means that immutable objects are always present locally and can thus be implemented either as local objects or as direct objects.

- A *direct* object is an immutable object except that its data area is allocated directly in the representation of the enclosing object. Direct objects are used mainly for primitive built-in types, structures of primitive types, and other simple objects whose organization can be completely deduced at compile time.

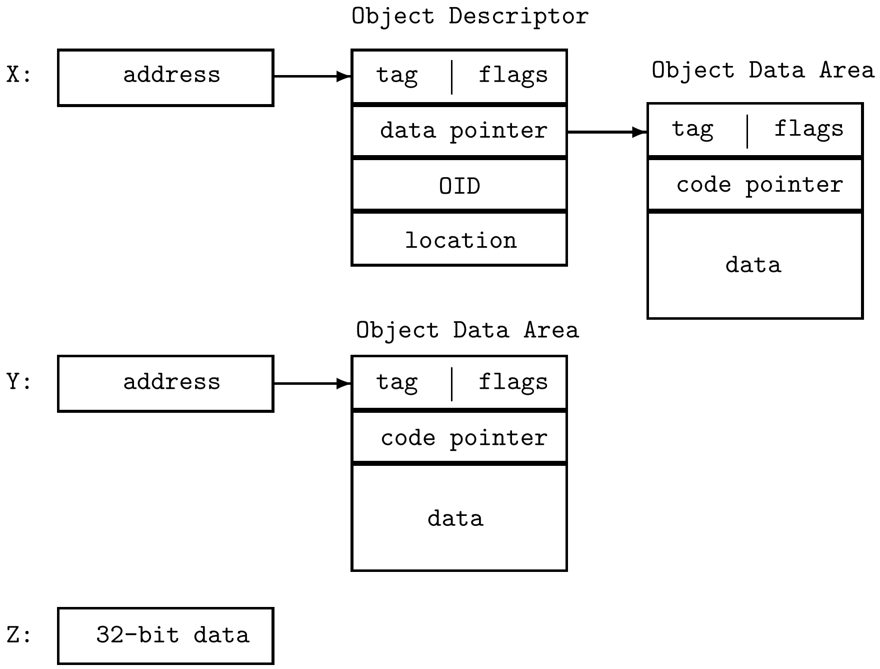

Figure 4.1: Emerald Storage and Addressing Structures

Figure 4.1 shows the various object implementations and addressing forms used. Variable X names a global object and the value stored in X is the address of a node-local *object descriptor*. Each node contains an object descriptor for every resident global object. In addition, there is a node-local object descriptor for every non-resident object that can be referenced from the node. Within a node, a global object is addressed by the node-local address of its descriptor. In principle, both object data areas and their descriptors live forever; in practice, when the last reference to a non-resident object W has been deleted from node A, A's object descriptor for W becomes a candidate for garbage collection. Both descriptors and object data areas can be garbage collected as described in Chapter 6.

An object descriptor contains information about the state and location of a global object. As shown in Figure 4.1, the first word of the object descriptor includes a tag field, which identifies it as a descriptor, and a number of flag bits, which indicate whether the object is local or global (the G bit) and whether or not the object is resident (the R bit). If the resident bit is set, the object descriptor contains the memory address of the object's data area; otherwise, the descriptor contains the current location of the object. The object descriptor also contains the object's object identifier (OID), a network-wide unique identification of the object. Object data areas also have a tag and flag bits in the first word. The second word contains a reference to the code for the object. The rest of the data area is the data for the object.

Variable Y in Figure 4.1 names a local object. The value stored in Y is the address of the object's data area. The first word of this data area, like the first word of an object descriptor, contains fields identifying the area as a data area for a local object. Finally, variable Z refers to a direct object that was allocated within the variable itself (e.g., an integer).

Notice that within a single node, all objects can be addressed directly without kernel intervention; most invocations are performed efficiently by compiled code. The generated code can readily tell whether an invoked object is local or global merely by inspecting the G bit and can therefore choose the appropriate invocation sequence. In many cases the compiler can determine that a variable will always refer to local objects and in these cases it will elide the G bit check.

In general, all references within a single node are location *dependent* and point to either a descriptor or a data area. For generality, the first word of all descriptors and data areas contains the previously mentioned tag and flag bits. We say that the data area for a local object acts as its own descriptor. In general, an object reference either points to a descriptor for a global object or to a data area for a local object—the G bit discriminating.

4.2.2 Inter-node Object Addressing

As described previously, a reference to a global object is represented as the address of the node-local object descriptor for the object. When passing references across node boundaries, these hard addresses must be translated, so each global object is assigned a unique network-wide 32-bit OID consisting of an 8-bit node identifier and a 24-bit sequence number (maintained on stable storage).[2] For efficiency, the OID is not assigned upon object creation; instead, the assignment is done when a reference to the object is passed outside the boundary of the creating node. Upon assignment of an OID, the object descriptor address is entered into a hashed *Access Table* that maps OIDs to object descriptor addresses. For flexibility, the access table is dynamically allocated and is automatically expanded should it become near-filled.

When a reference is sent to another node, the source kernel sends the unmodified virtual memory address of the object descriptor. It includes a translation record composed of the object descriptor address, the OID, the current location of the object, and a few bytes of control information. The destination kernel translates the unmodified virtual memory address by looking up the OID in the kernel's access table and translating the reference to the address of the object's descriptor on the destination node. If necessary, the kernel creates a node-local descriptor for the object. The current address information includes a time-stamp so that its age can be ascertained (the time-stamp and the exact format of the address is described in detail in section 5.2). The location information in the object's descriptor is updated if the

[2] In a production system, the OID would have to be larger than 32 bits.

newly arrived location information is newer. Finding and keeping track of mobile objects is the subject of Chapter 5.

An alternative scheme for inter-node addressing would be for references to contain the OID rather than the address of the object descriptor. References would be location-independent and it would not be necessary to translate them when crossing node boundaries. In Eden, the OID is encoded in the reference itself so that no translation information is required when sending a reference. The address of the object is found by looking up the OID in a hashed object table. Every reference to an Eden object incurs a high extra cost because the OID must be translated into the address of the object. This added overhead is especially a problem when invocations are performed by compiled code because the translation time can readily be greater than the time for performing the invocation. We chose to use location dependent addresses because it is the most efficient in the node-local case.

4.2.3 Code Sharing and Code Files

Conceptually every object has its own code which is specified in the object constructor that created the object (see 2.1.2). In the implementation we separate the code from the object so that all objects created by the same object constructor can share code. All objects created by an object constructor share the code specified in the constructor and contain a reference to the code in their data area. For each object constructor in a program, the compiler generates a separate assembler file. The UNIX assembler is then called upon to generate a code file in the UNIX executable *a.out* format. This code file contains the so-called *concrete type* of the object constructor. Code files are dynamically loaded and are treated internally as if they were immutable objects, called *concrete type objects*. They are assigned OIDs and are placed in dynamically allocated heap storage. The advantage of representing them as immutable objects is that they may be stored and accessed using the general object handling mechanisms, e.g., concrete types are located using the location scheme described in Chapter 5. Such objects are called pseudo-objects because they are represented as if they are objects but they are not Emerald language-level objects. The code file contains the executable code for all the operations (and the process, if any) of the object constructor and large amounts of control information needed by the kernel at run-time. The executable code for the operations is relocatable and the code file includes relocation information in the *a.out* format used by UNIX. The code for the Emerald operations is stored in the data area of the code file. It is native VAX code except that certain addresses are dynamically linked for efficiency. These addresses are pointers to other Emerald objects and are linked by the Emerald kernel when the code is read into the Emerald kernel's address space. The code file includes translation information enabling the kernel to translate from OID-based addresses generated by the compiler to the virtual memory addresses of the referenced objects.

Note that, as opposed to Smalltalk, the code for an object is statically defined at compile-time. In Smalltalk, the code can be dynamically changed. Bennett describes the problems that this causes in a distributed system [Bennett 87a]. The main problem is that it is difficult to keep code consistent across the network.

4.2.4 Dynamic Code Loading

The Emerald kernel dynamically reads in code files. When an Emerald kernel receives a request to start a program, it merely receives the OID for the relevant code file. If that file has not been loaded, the kernel will locate and read it. To locate the file, the kernel first looks in the directory where the compiler stores code files. If necessary, the kernel uses the location algorithm described in Chapter 5 to find the code file. When the code file has been located, it is read into the kernel address space and stored as an immutable object. In some cases, the code file is loaded from another node via the network.

After reading the code file, the Emerald kernel links the code into the kernel address space in two steps. First, the UNIX *a.out* format relocation information is used to link addresses of kernel variables and entry points for kernel routines. Second, the code file includes Emerald translation information which is used for inserting hard pointers into the code in place of OID-based references. For example, when the compiler emits code for the invocation of a local object of a well-known concrete type, it also emits a translation request containing the OID of the concrete type, the operation number for the operation to be invoked and the address of the point of call. The kernel then performs the translation *(Concrete type OID, Op number)* → *(hard address)* and inserts the hard address (the machine address) into the call instruction in the code. To perform the translation, the kernel looks up the virtual memory address of the concrete type, inspects the operation entry point table in the concrete type, and extracts the relative offset in the concrete type of the entry point. By adding the offset to the address of the concrete type the kernel obtains the hard address of the operation entry point. Note that when the kernel looks up the OID for the concrete type, it may not find it, and it may be necessary for the kernel to dynamically read the appropriate code file. Reading one code file into the kernel usually causes several other code files to be read also. Conceptually, code files remain in the kernel forever. In practice, they can be garbage collected when no longer needed.

As a concrete example of Emerald translation data, consider the program in Figure 2.2 on page 25. In line 12, compiled code calls the *getActiveNodes* operation on a node object. The compiler generates a jump to the entry point for this operation. However, the compiler cannot generate the actual virtual memory address of the entry point. Instead, it generates a translation request containing the OID for the concrete type for node objects and the operation number. When the kernel reads the code file, it will translate the pair

< *OID for Node concrete type*, *Operation number for getActiveNodes* >

into the appropriate virtual memory address. This scheme enables compiled code to directly reference other compiled code without an extra level of indirection. This increases the efficiency of the Emerald invocation mechanism.

4.2.5 Invocation Faulting

As previously mentioned, a major goal of Emerald is to provide good node-local performance. The compiler and kernel are closely coupled in that the compiler has knowledge of some kernel data structures and is allowed to operate on these without kernel complicity. Invocations of potentially non-resident objects are performed directly by compiled code: The code assumes that the object is resident and starts building a new activation record on the stack. Before jumping to the invoked operation, the compiled code checks the resident bit (the R bit) to see if the target object is resident. If so, then the invocation is executed directly, otherwise the kernel is called upon to perform the remote invocation. Using this optimistic approach called *invocation faulting*, global objects can be invoked locally in time comparable to a local procedure call, the only extra overhead being the check for residency of the object. An Emerald invocation of a potentially remote object that turns out to be locally resident can be performed in about 0.020 milliseconds on a MicroVAX II. This is a major improvement over, for example, Eden, where a comparable node-local call takes about 54 milliseconds on a SUN 2—three orders of magnitude longer.

There may be other reasons that compiled code cannot directly execute invocations on an object. For example, the object may not be fully instantiated (section 4.2.6). We have therefore introduced a single bit, called *frozen*, that is set if, for any reason, compiled code cannot directly execute the invocation but must take a fault. When handling a fault, the kernel finds the reason for the fault by checking the bits in the tag field of the object descriptor. If necessary, the process is suspended until the condition causing the fault has been resolved, e.g., until a remote invocation has completed.

The fault check is implemented as a test and jump around a kernel call. On a VAX, this check can be performed efficiently by a single Branch-On-Low-Bit-Set instruction, so very little overhead is introduced in the common case where the invocation can proceed. If we were to implement Emerald invocation in hardware, even this overhead could be almost entirely eliminated because the fault check could be performed in parallel. We could also put an illegal address in the descriptor and let the hardware fault (as for **nil** checking). Unfortunately, we do not have full control of hardware faults and the UNIX fault handling mechanism (signals) is not very efficient.

Alternatively, if we had full control over memory management, we could freeze objects by setting the page upon which the object resides to be non-accessible. However, it would be difficult to have more than one object on a page because they would be frozen together.

4.2.6 The Initially Problem

When an object that has an **initially** section is created, the creating process executes the **initially** as if it were an anonymous operation. The object is not considered to be fully instantiated until the **initially** completes. This gives rise to an esoteric synchronization problem when processes attempt to invoke the object before it is fully instantiated. Such invocations must be delayed. This situation can only arise when an object gives out a reference to itself in its **initially** section. To solve this problem, we use the faulting mechanism described in section 4.2.5. A new bit is introduced in the tag field of an object: *setUpDone* which is false until the object has been fully instantiated. To prevent invocations of the object, it is frozen (by setting the *frozen* bit in the object tag) until the object has been fully instantiated. This forces an invoking process to take an invocation fault. The kernel preempts the faulting process and puts it on a special queue (creating the queue, if necessary). When the object subsequently has been fully instantiated, the kernel empties the queue and schedules the processes found in it for execution. This *initially faulting* scheme does not add overhead in the normal case because the faulting mechanism was already in place due to the presence of distribution.

There is a minor problem with local objects because the faulting scheme is only in place for global objects. We chose a simple, straight-forward solution to this problem. The compiler can statically determine whether or not there is a possibility that the object can be invoked while being instantiated. (In almost all cases, there is no such possibility.) If there is, the compiler choses to implement the object as a global object.

Note that our semantics gives the **initially** section exclusive access to the object because we believe that invoking an uninitialized object cannot make sense. If statements in the **initially** section call another object which makes a call-back to the original object then a deadlock will occur. If the **initially** needs to be able to make such calls then those calls belong in the **process** section.

4.3 Process Implementation

Emerald processes are implemented as light-weight processes in a manner similar to the implementation of processes in Concurrent Pascal [Brinch Hansen 77] and Concurrent Euclid [Holt 83]. When an object with a **process** section is created, a new Emerald process is created.[3] It is implemented by a process descriptor and a stack which are dynamically allocated when the process is created. The stack contains one *activation record* for each ongoing

[3] For efficiency stacks are reused when possible.

invocation. A thread of control is created to execute the process. A process can invoke operations on its object or on any object that the process can reference. The thread of control of one object's process may pass through other objects; in the case in Figure 4.2, the process owned by object A invokes operations in objects A, B, and C.

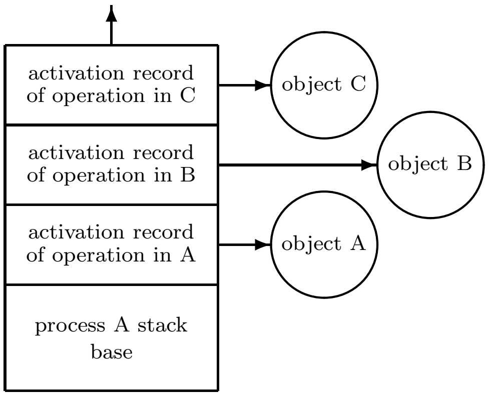
activation record of operation in C
object C
activation record of operation in B
object B
activation record of operation in A
object A
process A stack base

Figure 4.2: Process Stack and Activation Records

When a process is created the kernel builds an activation record on the stack by pushing the process parameters onto the stack and then simulating an invocation of the process block body as if it were an anonymous operation of the object. When the process invokes another object, compiled code creates an activation record without kernel complicity. If an invocation is for a non-resident object, the compiled code faults to the kernel, which removes the parameters from the stack and performs a remote invocation. Conceptually, it is the same process that continues to execute on the remote machine, so the remote kernel does not create a new process to handle the invocation. In practice, the remote kernel creates a new thread of control including a node-local stack upon which the thread of control can perform the invocation on behalf of the process. Thus, the stack for a process can be split over many machines; we call each part of the stack a *stack segment*.

We make a distinction between an Emerald process and a thread of control. The latter is node-local and executes node-local invocations on behalf of an Emerald process. When an Emerald process spans multiple nodes, there is a thread of control for each remote invocation.

Emerald stacks are small for two reasons. First, most activation records are small because they consist only of object references. Object data areas are allocated on the heap, e.g., as opposed to other programming languages, arrays are never stack allocated. Second, in object-oriented programming, variables are commonly encapsulated in objects and are therefore not allocated on the stack.

To save space, the kernel initially allocates a small stack segment to each process and dynamically expands the stack when necessary. A stack check is performed upon each operation entry. Compiled code checks that there is enough room on the stack for the next activation record. If not, then the kernel allocates an additional stack segment and links it to the previous stack segment using the same mechanism as for remote invocations. To prevent run-away processes from eating up all available memory, we set a maximum limit on the number of bytes allocated to a given process.

The kernel treats stack segments as objects. They are assigned OIDs and can be located and referenced in the same location-independent manner as other objects (as described in section 4.2.2 and Chapter 5). This is particularly useful when performing returns from remote invocations because the caller may have moved.

The process control information is stored in each stack segment and includes sufficient information to allow a return to any previous stack segment. The bottom activation record contains a return address pointing to a kernel routine. When a return is made from the bottom activation record, the kernel either terminates the process (if it has no more stack segments) or returns control to another (possibly remotely located) stack segment.

A single process can span a number of nodes and can have several stack segments on the same node. Conceptually, these stack segments represent the single Emerald process. Alternatively, seen from the kernel level, we may also say that each remote invocation is handled by a separate thread of control on each node. Each thread is implemented by a stack segment. When a process makes a remote invocation, the local thread is suspended and control is passed to the remote node where a new thread is created to handle the incoming invocation. From this implementation point of view, our remote invocation system is similar to Cedar RPC as described by Birrell and Nelson [Birrell 84].

4.3.1 Execution Environment

Initially, when an object with a process is created, its process executes in the context of that object. When the process invokes operations on other objects its thread of control passes to the operation and the process continues execution in the context of the invoked object. The object in which the process is currently executing is called the *current object* for the process. If the object is a local object then it is contained within some global object which we call the *current global object* for the process. While executing in an object, the process has direct access to its execution environment through processor registers as shown in Table 4.1. The register numbers are for the VAX implementation (the SUN implementation uses a similar layout). When a process is preempted, the current set of registers is stored in a separate register save area in the stack segment.

4.3.2 Activation Record Layout

The current object is addressed relative to the “g” register which points to the start address of the object data area. The local variables of the operation are allocated in the activation record

Table 4.1: Processor Register Allocation (on the VAX)

No.	Symbol	Name	Address of
r15	ip	Instruction ptr	Next instruction to execute
r14	sp	Stack ptr	Stack top
r13	l	Activation Record ptr	The current activation record
r12	g	Current Object	The data area for the current object
r11	b	Current Global Object	The descriptor for the current object
r10	ssp	Stack Segment Ptr	The current stack segment
r4 — r9	r4 — r9	Locals	Variables allocated in registers
r1 — r3	arg[123]	Kernel args	Kernel call parameters
r0	scratch	Scratch register	—

and are addressed relative to the "l" register. As an optimization, the compiler allocates some of the local variables in registers (on the VAX it uses r4 – r9). Upon entry to the operation, the previous contents of the registers are stored in a register save area in the activation record.

An activation record, as illustrated in Figure 4.3, is composed of the following parts (listed from the top of the stack toward the bottom):

Arguments for the next invocation (overlaps the next activation record).

Results to be returned from the next invocation (overlaps the next activation record).

Register save area for those registers that the operation uses for local variables. The size of the area depends upon the number of local variables allocated in registers.

Temporaries used during expression evaluation.

Local variables of the operation.

Invoke queue containing a special set of link fields used for handling process mobility as described in section 4.4.3.

Dynamic link composed of

- **ip** the instruction pointer for the return from the invocation.
- **b** the previous b pointing to the previous current global object.
- **g** the previous g pointing to the previous current object.
- **l** a pointer to the previous activation record on the stack.

Arguments for the current invocation (overlaps the previous activation record).

Results to be returned from the current invocation (overlaps the previous activation record).

Activation records overlap; the parameters for an invocation are pushed onto the stack and become the bottom part of the next activation record.

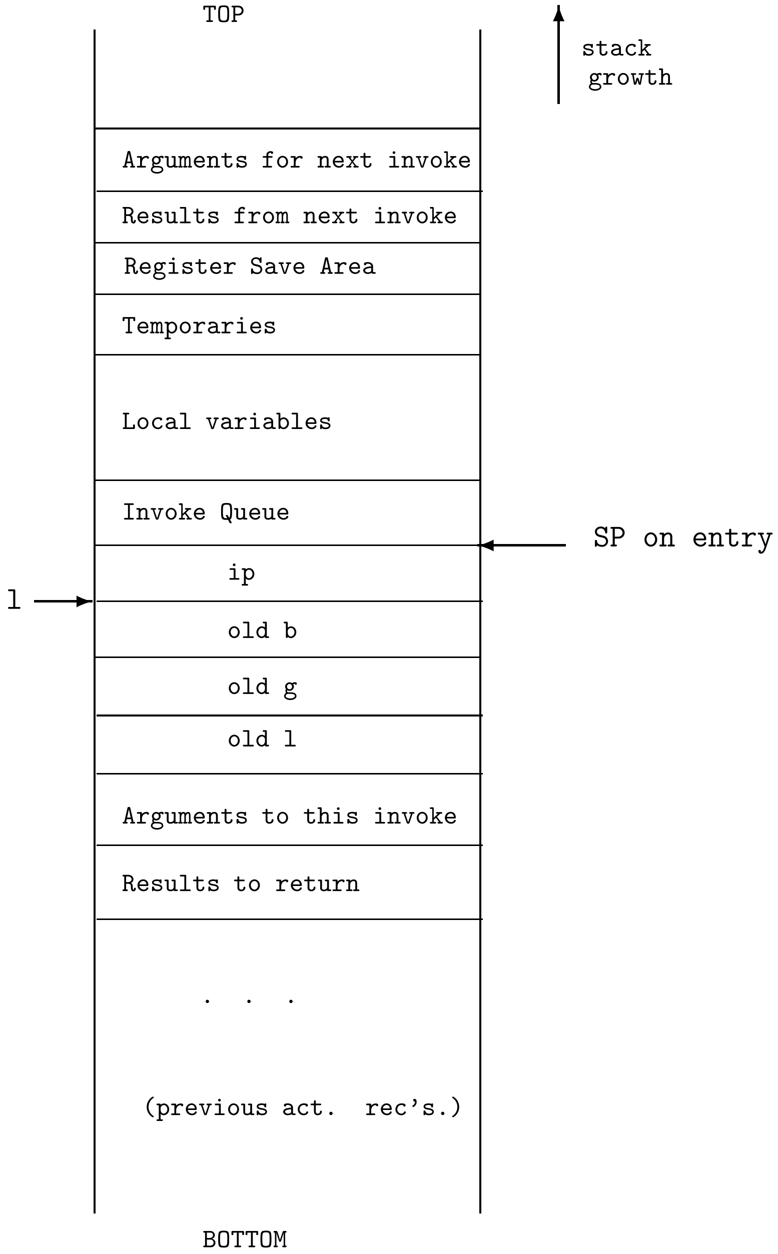

Figure 4.3: Activation Record Layout

4.3.3 Activation Record Templates

In addition to templates for data areas, the compiler produces templates for activation records so that the variables in an activation record that is moved to another machine can be found and translated. A template for an activation record describes four things: the parameters to the operation, the local variables used by the operation, the register save area, and the contents of the CPU registers. The invoke queue and the dynamic link are not described by the template because they are present in all activation records. Furthermore, the results and arguments to the next invocation are not described; instead, the kernel simply assumes that the top of the stack is a parameter area under construction. This allows the top part of the stack to vary in size during the execution of the operation; the stack pointer will always indicate what part of the stack is valid.

To simplify activation record templates, the Emerald kernel does not permit registers that contain variables to change their template-type during an operation. A register that contains a pointer must contain a pointer for the lifetime of the invocation. However, the pointer register can point to different objects of different types. This restriction is similar to the segregation of address and data registers in some architectures, but is more dynamic because the division is made for each specific operation. In practice, the restriction has no consequence for the current implementation because the compiler is still free to reallocate registers used for temporaries and, due to the type information available, the compiler can use pointer registers to hold direct objects. Without this restriction, we would need to have different templates at different points in an operation's execution—a design considered early in the project, but later abandoned as unnecessary.

4.3.4 Pseudo-concurrency and Critical Sections

The kernel time-multiplexes Emerald processes to achieve pseudo-concurrency. Stack segments that are willing to execute are kept in a ready queue and are allowed to execute for one time interval in a round-robin manner. Processes are not actually preempted when an interrupt[4] occurs. The problem is that during execution there are critical sections when compiled code inspects and manipulates kernel data structures and modifies the process state. For example, when performing an invocation, several registers must be saved on the stack and they must be assigned new values. The saves and assignments must be performed as one atomic operation. Unfortunately, no single machine instruction can perform the atomic action required. There are several solutions to this problem:

- We could provide explicit locking around the critical sections. This would add significant overhead to invocations, which is clearly undesirable.

[4] An ULTRIX signal is considered an interrupt in this context.

- We could write the kernel routines so that they could deal with the inconsistent data structures. This is essentially the same as defining a new set of weaker invariants for the data structures and keeping them up-to-date.

- The Owl/Trellis kernel [Schaffert 86] uses an ingenious solution to this problem. When an interrupt occurs, the kernel inspects the last instruction executed. If it is part of a critical section, the kernel simulates the execution of the remaining instructions of the critical section. Thus the kernel, in effect, lets the critical section execute to completion before handling the interrupt. This works well if the critical sections are well-known in advance and are few and short.

  On some machines there are instructions that allow the execution of a single instruction (e.g., the "EX" instruction on the UNIVAC 1100 computers). Such instructions can be used to efficiently execute the remaining instructions in a critical section. On VAX computers, the trace trap facility could also be used.

- The compiler can emit code that, at well-chosen points (such as at the bottom of loops), checks for interrupts. This introduces some overhead during normal execution, but this overhead can be held to a minimum.

We chose the last of these possibilities because we deem it to be simple, effective, and reliable. It has been successfully used in several earlier systems, most notably Concurrent Pascal and Concurrent Euclid. We have implemented the mechanism as follows. Processes can be preempted at two points: at the bottom of loops and at the start of every operation. Interrupt routines set a *preemptFlag* to indicate that the current process should preempt itself. At the bottom of loops, the compiler emits code that calls a preemption routine, if the *preemptFlag* is set. This check is inexpensive because in most cases it replaces an unconditional jump instruction with a conditional one. We also check the *preemptFlag* at the start of every operation. By using a simple coding trick we get the check for "free". We let interrupt routines reset the current stack limit for the executing process to such a value that the process is certain to fail the next stack check. Because compiled code must perform the stack check anyway when performing an invocation, this preemption check comes for free.

In theory, the loop check is sufficient to ensure that a process is preempted eventually. However, it is possible to write long-running programs without loops, for example, calculating Ackermann's function. Therefore we supplement the loop check with the check at the entry to operations. In practice, processes are preempted quickly because it is difficult to write long running programs that use neither loops nor invocations.

Note that the loop check is not necessary if the loop contains at least one invocation. By performing a flow analysis the compiler can attempt to determine whether or not an invocation always will be performed in the loop. If so, the compiler could elide the loop preemption check.

4.3.5 Finding and Translating Pointers

The use of location dependent addresses in Emerald increases the performance of local invocations. This improvement comes at a cost; movement of an object involves finding and modifying all of the pointers in the object, increasing the cost of mobility. We feel that this is reasonable because motion is less frequent than invocation. This design places the cost of mobility on those objects that use it.

There are many ways of finding and translating references. For example, Smalltalk-80 uses a tag bit in each word to indicate whether or not the word contains an object reference. Unfortunately, using tags increases the overhead of arithmetic operations and complicates the implementation in general. The Emerald compiler generates a so-called *template* for every object data area describing the layout of the area. The template is stored with the code in the concrete type object for the object. Each object data area contains a reference to the concrete type object so that the code and the template can be found given only the data area. In addition to their use for mobility, templates are used for garbage collection and debugging. As an example, consider the Emerald program shown in Figure 4.4 which defines a single object containing three variables inside a monitor. The variable *myself* contains a reference to the object *simpleobject*. The variable *name* is initialized to point to a local string object. Conceptually, the variable *i* points to an integer. Integers are implemented as direct objects, so the variable contains the two's complement representation of the integer 17. The corresponding object data area and template are shown in Figure 4.5. The data area for *simpleobject* contains:

```
const simpleobject == object simpleobject
    monitor
        var myself : Any ← simpleobject
        var name : String ← "Emerald"
        var i : Integer ← 17
        operation GetMyName → [n : String]
            n ← name
        end GetMyName
        ⋮
    end monitor
end simpleobject
```

Figure 4.4: Simple Emerald Object Definition

- 4 bytes of control information as described earlier.
- A pointer to the concrete type object for *simpleobject*.
- A lock for the monitor.

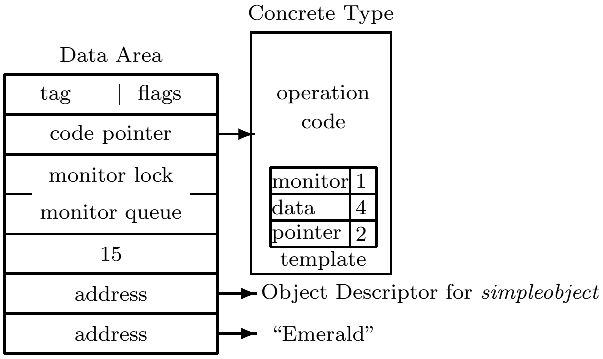

Figure 4.5: Data Area and Template Structure

- The variable *i* allocated as 4 bytes of data.
- The variables *myself* and *name*, each allocated as a pointer to an object.

The template does not describe the first two items because every data area contains them. Each template entry contains a count of the number of items described, a flag indicating whether or not the items are attached, and the so-called *template types* of the items. Template types indicate what the corresponding part of the data area contains. The template types include:

Pointer which is the address of an object; pointers must be translated if the object is moved. If the object is a global object, the address is that of the object's descriptor. If the object is a local object, the address is that of the object's data area. The kernel can discriminate by inspecting the tag bits in the first word of the descriptor or data area. (Pointers can also contain the address of the **nil** object represented as a special address.)

Data which is direct data (e.g., integers) stored as a number of bytes; these are not translated.

Vector which indicates that the object is a variably sized vector. The actual size of the vector is stored in the data area. The template further indicates the template-type of the vector elements so that any necessary translation of these can be performed.

MonitorLock which controls access to the object's monitor. Monitors are implemented as a Boolean and a queue of processes awaiting entry to the monitor. The monitor queue must be moved if the object is moved.

In principle, the compiler produces a template entry for each variable. However, a single entry can describe several variables that have the same template-type. Therefore, to save template space, the compiler contiguously allocates variables that can be described by identical template entries. The order in which variables are allocated in the object data area is therefore not the same as the order of appearance in the program. This space optimization has meant that the average template contains only two or three entries because there is only one template entry for each template-type.

4.3.6 Remote Invocations

When a process attempts to invoke a non-resident object, it will fault to the kernel as described in section 4.2.5. The kernel performs the remote invocation by packing the parameters into an invocation request message and sending it to the invokee's node. The kernel at that node sets up a stack segment to perform the remote invocation, copies the parameters onto the stack, and lets the process execute. When the process returns after performing the invocation, the kernel packs the results into a reply message and sends it back to the caller's node. There the kernel unpacks the results, places them on the process's stack, and lets the process continue execution. Parameters and results are sent unmodified but are translated at the receiving node as described in section 4.2.2.

If the kernel knows the location of the target object it optimistically sends the invocation to the target's node. Because we expect remote invocations to be much more frequent than moves, it is more efficient to send the invocation message to the target node, rather than sending a message first asking whether the object has moved. Asking first costs two extra network packets (one for asking and one for replying) and would essentially double the execution time for a remote invocation. If the object has moved, the target node simply forwards the invocation packet (including any call-by-move parameters) to the new node. Other forwarding protocols are discussed in Chapter 5.

4.4 Moving Objects

Using the addressing and implementation structure described previously, the actual moving of an object is rather straightforward. Moving an object consists of copying its data area to another machine and translating its pointers.

4.4.1 Moving Data Objects

Objects without active invocations are the simplest to move. The Emerald kernel builds a message to be transmitted to the destination node. At the head of this message is the data area of the object to be moved. As previously described, this data area is likely to contain pointers to both global and local objects. Following the data area is translation information enabling the destination kernel to translate location-dependent addresses (see section 4.2.2). A local object is always considered to be attached to its containing global object, so the object data area is sent along with the local object's virtual memory address.

On receipt of this information, the destination kernel allocates space for the moved object, copies the data area into the newly allocated space, and builds a translation table that maps the original addresses into addresses in the local address space. OIDs are used to locate object descriptors for existing global objects. The kernel then locates the template for the moved object, scans its data area, and replaces any pointers with their corresponding addresses found in the translation table. The underlying message passing system provides reliable delivery so there is no need for an acknowledgement of the move. For small objects (less than about 1500 bytes) the entire move can be handled in a single Ethernet datagram.

4.4.2 Moving Code

When an object is moved to another node, its code is not sent along. We assume that it is already present on the remote machine and only send a reference to the code in the form of an OID. If the other node does not have a copy of the code, it requests a copy from the sending node. There is one copy of the code on each node where an instance of the object constructor exists. Code can be garbage collected when no longer needed.

We believe that this lazy evaluation policy is reasonable because the code will have to be remotely loaded anyway, if it is not already present on the remote machine. The cost of using lazy evaluation when the code is not present is an additional message exchange (to request the code). This is a relatively minor cost compared to the actual transfer of code. The savings when the code is already present are substantial because the code is not transferred in that case.

4.4.3 Moving Process Activation Records

The activation records for processes executing an object's operations must also move when the object is moved. This presents a particularly difficult problem: Given an object to move, which activation records must move with it? Somehow we must be able to find the set of all activation records that represent invocations of the object. The problem is that this set changes rapidly; for every non-inlined invocation, there is an insertion into and a deletion from the set. Any per-invocation overhead will therefore have a severe performance impact. Several solutions are possible, but all have potentially serious performance implications. The simplest solution is to link each activation record to the object on invocation and unlink it when the invocation completes. This would unfortunately increase the invocation overhead in the current implementation by almost 60%. On the other hand, finding the activation records to move would require only a simple list traversal. Another solution is to create the list at move-time. This would avoid the invocation-time cost but would require a search of all node-resident activation records when moving an object. While we believe that mobility should not increase the cost of invocation, exhaustive search seems to be an unacceptable price to pay on every move. We have therefore adopted an intermediate solution. Our implementation runs on a uniprocessor so at any given time there is only one process executing. We maintain a linked list of all activation records executing in each object, as in the first solution above.

However, the activation record is not linked into this structure on invocation. Instead, space is left for the links and the activation record is marked as "not linked." When the (one and only) executing Emerald process is preempted, its activation stack is searched for "not linked" activation records, and these are linked to the object descriptors of their respective objects. The search stops as soon as an activation record is found that has been linked previously. In this way, the work is mainly done at preemption time and its cost is related to the *difference* between the stack height at the end of the execution interval and the lowest stack height during the interval. What is important is that the cost does *not* depend on the number of invocations performed—there is no per invocation overhead other than a single instruction to mark the activation record as "not linked." Profiling of executing kernels have shown that only 0.3 – 0.7% of the execution time of the kernel is spent in traversing and marking process activation records.

An operation must still unlink its activation record when it returns. Each return must check for a queued activation record and dequeue it before freeing the record. However, because many invocations are usually performed between successive preemptions, most returns will find a "not linked" activation record, in which case no work need be done. The overhead is in this case a single conditional jump instruction. When an object moves, the kernel performs the linking for the current process; thereafter the kernel can find all activation records that must move merely by traversing the linked list associated with the object.

Activation records are moved in a manner similar to that for moving data areas. If necessary, the activation records are removed from the stack segment containing them. This is accomplished by splitting the stack segment into (at most) three parts: the "bottom" part which remains on the source node, the "middle" part which is moved to the destination node, and the "top" part which is copied onto a new stack segment on the source node. The stack break points are found using the templates for the activation records. Note that when splitting a stack, the overlapping parameter area is duplicated because it is required by both activation records. At each of the two stack breaks, invocation frames are modified to appear as if remote invocations had been performed instead of local invocations. Figure 4.6 shows the structure that would exist if object B contained in Figure 4.2 were moved from node α to node β. This also illustrates the difference between an Emerald process and a thread of control. A new thread of control is created for each part of the stack. Thus a single Emerald process can have multiple threads of control on a single node: one for each outstanding invocation. Our concept of threads of control is similar to the concept of a process in Cedar RPC [Birrell 84].

4.4.4 Moving Attached Objects

As described in section 3.5.5 a variable can be attached. When moving an object (or process) that contains an attached variable, the object referenced by the attached variable must also

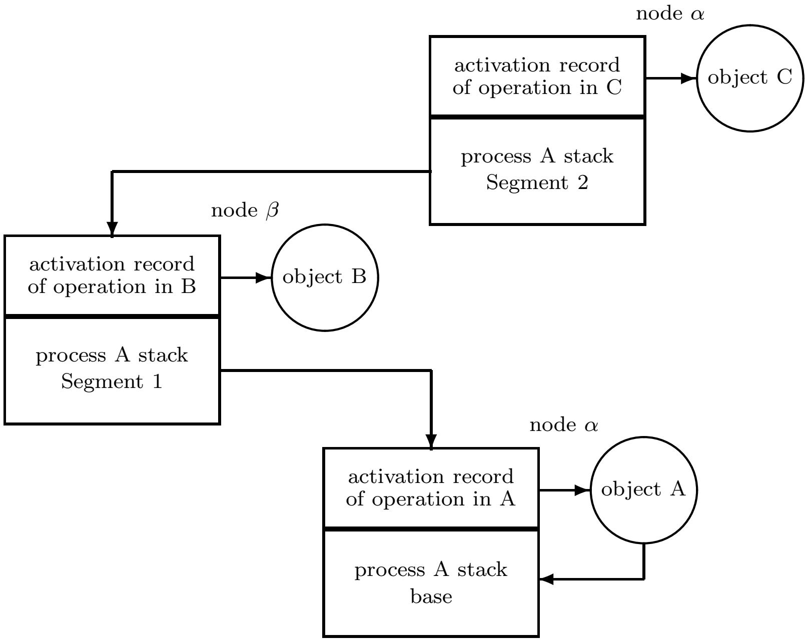
node α
activation record of operation in C
object C
process A stack Segment 2
node β
activation record of operation in B
object B
process A stack Segment 1
node α
activation record of operation in A
object A
process A stack base

Figure 4.6: Process Stack after Object Move

move. Implementing attachment is simple: we merely introduce an *attached* bit in the template entry for variables. If the bit is set then the variables corresponding to the template entry are attached. When moving an object (or process), the kernel scans the template to find references for which translation information must be sent to the target node. If the template entry indicates that a variable is attached then the kernel sends the object referenced rather than a reference to it. This simple method ensures that the presence of attached variables does not introduce additional overhead except when used.

4.4.5 Handling Processor Registers

An additional complexity in moving Emerald processes and activation records is the management of processor registers. The Emerald compiler attempts to optimize the addressing of objects by storing local variables in registers instead of in the activation record. In this way, some of the processor registers may contain machine-dependent pointers and these must be translated when the activation record moves. Unfortunately, the registers for a given activation record are not kept in one place. Each invocation saves a copy of those registers that will be modified.

Referring back to Figure 4.2, suppose that the first invocation (of A) and the third invocation (of C) both use register 5. In this case, a copy of A's register 5 is saved in *C's* activation record, as it would be in any conventional stack-based language implementation. If object A moves, its activation record will move with it. The stack will be segmented, and the rest of the stack will be left behind. Furthermore, the copy of A's register stored in C's activation record will be incorrect when C returns, because the data it refers to will be on a different node. To handle this situation, the kernel sends a copy of the registers used in an invocation along with the moving activation record. First, the kernel finds the template for the activation record in the concrete type object of the invoked object. Second, it determines which registers are used as pointers in the activation record by looking at its template. Templates for activation records have special entries for registers and for the area of the activation record where registers are saved. Third, the kernel scans the invocation stack, looking for the next activation record that has saved each of the registers. In this way, copies of the current values of the registers can be sent along with the record. On the destination node, the registers are modified using the translation table (as described in Section 4.4.1) and stored with the newly created stack segment.

For each stack segment of an Emerald process, there is a separate register save area containing the registers that are to be used when the stack segment resumes execution. These registers constitute the current environment for the active invocation, i.e., the invocation described by the activation record at the top of the stack. When an invocation return crosses a stack segment boundary, the registers used are those stored with the stack segment receiving control. These are the (possibly translated) values of registers that were computed when the stack was segmented.

4.4.6 Moving Monitors and Conditions

When moving an object containing a monitored section, there may be processes waiting to enter the monitor or waiting on conditions inside the monitor. These processes must also be moved when the object moves. There is no special problem here because the processes are found via the activation record link described above. The only additional work is to ensure that the monitor and condition queues are reestablished at the receiving node.

When traversing the template for a moving object, the sending kernel finds the queue of processes waiting for entry to the monitor. The kernel empties the queue and sends the processes to the target node along with an identification of the object. The processes are moved as described above (section 4.4.3). The sending kernel may also find one or more processes waiting on conditions. As for monitor queues, the condition queues are emptied and the processes sent to the target node along with an identification of the condition. The kernel at the target node reinserts the processes into their respective queues using the identification of the queue.

4.5 Implementation Summary

In this chapter, we have discussed most of the major problems related to implementing a kernel that supports mobile objects. Emerald provides efficient local execution for objects by placing all objects in the same address space and by close cooperation between compiler and kernel. We have introduced the concept of an invocation fault to efficiently handle potentially remote invocations that turn out to be local. We have shown how fine-grained objects can be efficiently represented in memory and how they may be extracted from one address space and moved to another. The next two chapters describe how to find objects within the network and how to garbage collect them.

Chapter 5

Locating Mobile Objects

In this chapter, we discuss the problem of locating an object within a local area network given a reference to it. Such location information is required, for example, when a remote object is invoked, because the kernel must know where to send the invocation request. Mobility increases the complexity of the problem because location information can become obsolete. The location problem can be handled either by keeping location information up-to-date or by providing a mechanism to update it when required.

In the following sections, we discuss the location problem and several alternative semantics for location operations. Second, we describe the concept of *forwarding addresses* that is used as a basis for location in our Emerald prototype. Third, we describe how location is done in our prototype. Fourth, we describe an algorithm to recover lost forwarding addresses.

5.1 The Location Problem

The location problem can be stated as:

> Given a reference to an object, find the node where the object resides.

Because objects are free to move at any time, location information is potentially obsolete the instant it is provided; at best it is a good hint. Therefore, it is impossible for any location procedure to provide a location result with semantics as strong as: "The location returned is the one at which the object currently resides." This impacts the kernel design because the kernel must be prepared to handle any situation that arises from using obsolete information.

At the implementation level, an object's location is represented merely by the node number of the node where the object resides. The location algorithm is embedded in a *locate* operation which, given a reference to an object, returns the location of the object (if it is known). Initially, we consider only the case where the desired object is available somewhere within the network; the case where the object cannot be found is discussed later. We consider two semantics for the result of a *locate* operation.

Weak semantics (old knowledge)

The location returned is one at which the object has resided at some point in the past. The location returned is merely a hint and may well be obsolete.

Strong semantics (recent knowledge)

The location returned is one at which the object has resided at some point *after* the initiation of the location request. This is stronger semantics because we know that the object was at the returned location at some time *after* we asked. If we also know that the object has not moved since we asked then we know its exact location.

Locating an object using weak semantics is called *weak location*; locating an object using strong semantics is called *strong location.* In theory, weak location is so weak that merely returning the location where the object was created suffices. In practice, it is more reasonable to return the latest known location. Computationally, this is very inexpensive because it merely involves inspection of a node-local data structure—no remote requests are involved. The location returned is potentially obsolete, but often it is accurate enough because objects tend to be accessed more frequently than they are moved. Implementing strong location requires that the kernel verifies the actual location of the object in question. Such verification can only be provided by the node where the object resides. This may require an expensive search of the network. Even if the object has not moved since it was last heard from, it is still necessary to query its node, thereby incurring the cost of (at least) two network messages.

Table 5.1 compares the characteristics of the two kinds of location. The main advantage of weak location is that it is fast and inexpensive; the main disadvantage is that it may fail to provide a usable answer. The main advantage of strong location is that the location provided is much more up-to-date; the main disadvantage is that it is slow and costly.

Table 5.1: Characteristics of Weak and Strong Location

Characteristic	**Weak location**	**Strong Location**
Semantics	Hint	Almost current
Implementation	Use node-local info	Verify hint or search
Efficiency	++	−−
Speed	++	−
Up-to-date	−	+

Strong location is necessary in distributed systems. When nodes crash, objects are lost or at least become temporarily inaccessible. A lesson from the Eden system was that it is crucial for a location protocol not to report an accessible object as being inaccessible. On occasion, the Eden system would report that a particular object was inaccessible when it was indeed accessible, causing applications to fail. This was especially frustrating when an Eden programmer had carefully orchestrated a test by placing different objects on a number of nodes, only to have the test fail because of location protocol problems. We therefore add the following to our definition of strong location semantics: When an object location request fails, the object was not accessible on any node that was accessible at the completion of the location request.

5.2 Forwarding Addresses

Forwarding addresses is a technique for keeping track of entities in a distributed system. The first distributed system to use forwarding addresses was the DEMOS/MP system (see section 3.1.1). In DEMOS/MP, interprocess communication is performed through *links* each of which contain a unique id and a location hint for the referenced process. When a process is moved from one machine to another, a forwarding address is left behind. Should another process subsequently attempt to use a link with the old location hint, the access request is forwarded to the new location (which may again forward it). Upon receiving a forwarded request for a resident process, the receiving kernel sends an update message back to the original sender of the request, so that the out-of-date link is updated. Next time a message is sent using the link, the path of forwarding addresses that must be followed is shorter; we say that the update message causes the path to be compressed. Fowler [Fowler 85, Fowler 86] studied forwarding address schemes in detail in the context of very large distributed systems. He treats the subject extensively and investigates several different methods for keeping forwarding addresses up-to-date. We have extended one of his methods to enable recovery of lost forwarding addresses. Our assumptions about the underlying system are different than Fowler's: we assume a local area network with a limited number of nodes and full connectivity. This allows us to employ exhaustive search if necessary—an option explicitly ruled out in Fowler's model.

5.3 Location in Emerald

In our Emerald prototype, location information is required by the kernel to:

- Invoke an object.
- Return a reply to an invocation.
- Return the location of an object when a **locate** statement is executed.
- Find the concrete type for an object when an object is moved to a new node.
- Move an object.
- Fix an object.

In general, strong location is preferable because it gives the most accurate information. However, it is slow and costly. In most cases the Emerald kernel therefore uses two protocols. First it optimistically uses weak location. If that fails, it uses strong location to ensure correct, albeit slow, operation of the system. 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 packet. If the hint turns out to be wrong then strong location is used to locate the object and the invocation message is forwarded. Using weak location works quite well in the common case where the object has not moved because in this case only two network messages are required to do a remote invocation. An advantage is that objects that do not move frequently can be invoked efficiently—their performance is not degraded by the presence of mobility.

Our strong location algorithm consists of a number of stages, each built on the previous stage, and each increasing the probability of success. For this reason, we call it the *Cascading Search* algorithm. For reliability, the algorithm progressively trades off performance for a higher probability of success and ultimately provides strong location semantics. At the heart of the algorithm is the concept of forwarding addresses as presented by [Fowler 85]. 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 although they cannot be found using an access path that includes the crashed node. The Cascading Search algorithm provides strong location in the face of node crashes by supplementing Fowler's forwarding address scheme with a set of stages that reliably recovers lost forwarding addresses.

5.3.1 The Use of Forwarding Addresses in Emerald

In this section, we describe forwarding addresses as they are used in the Emerald prototype. As described previously, each global object is assigned a unique network-wide object identifier (OID), and each node has a hashed access table mapping OIDs to object descriptors. The access table contains an entry for each resident object for which a remote reference exists and for each non-resident object for which a local reference exists. Every object descriptor contains a *forwarding address* as well as the object's OID. A forwarding address is defined as a tuple <*timestamp, node number*> where the *node number* is the node number for the latest known location of the object, and the *timestamp* specifies the age of the forwarding address. Given conflicting forwarding addresses for the same object, it is simple to determine which one is most recent merely by comparing the timestamps. As observed by Fowler, it suffices for the timestamp merely to be a count of the number of moves for the object. The reason is that new forwarding addresses are only created when we attempt to move an object, and forwarding addresses for different objects need never be compared. An alternative would be to use timestamps based on a global clock. This has several disadvantages. First, a globally consistent clock must be maintained, e.g., as described in [Lamport 78]. Such clock maintenance has a high computational cost. Second, clock based timestamps take up more storage space, because objects can be generated at a high rate, requiring a fine resolution clock. Every reference (to a global object) sent across a node boundary contains the OID of the referenced object and the latest available forwarding address. The kernel on the receiving node may then look up the descriptor for the object in the access table using the OID and update the forwarding address, if required. If the descriptor is not in the table, then one is created because it is the first time that this kernel has heard of the object.

If an object X is moved from node A to node B, both A and B will update their forwarding addresses for the object. No action is taken to inform other nodes. Should an object Y on node C try to invoke object X at A, A will forward the invocation message to B. When the invocation completes, B will send the reply to C with the new address piggybacked onto the reply message. This allows C to update its forwarding address and "short-circuit" the forwarding path to X—Fowler calls this updating strategy *Jacc*. (Jacc stands for "jump access" because future references using the new forwarding address "jump" directly to the object skipping the old path, i.e., the path has been compressed to length one.) Jacc has the advantage that it is very inexpensive to maintain the forwarding address information because no update messages are ever sent. It has the disadvantage that until the first invocation returns with the updated address, any other invocations sent from C directed at X will also have to be forwarded. On the other hand, because Emerald invocations are synchronous, a given Emerald process does not perform another invocation until the previous one returns. Therefore at most one invocation message need be forwarded for any given Emerald process. If many processes simultaneously invoke X then it can be necessary to forward a large number of invocation messages. Alternatively, when a kernel forwards a message, it could send back a forwarding address update message immediately. This would reduce the number of forwarded messages, but it would also require an extra control message for every forwarded message.

5.3.2 Discussion of the Forwarding Address Scheme

The forwarding address scheme is quite powerful. In the absence of node crashes, it can be used to provide strong location semantics by following forwarding addresses until the object is found. In other words, we can locate an object by following in its footsteps. This assumes that the object does not move faster than messages can be forwarded to it which might be a faulty assumption. The problem is that an object might "outrun" messages trying to reach it. Although we cannot rule out this problem, we argue that it is unlikely to occur in our current Emerald prototype. Messages are forwarded immediately at the kernel level while move requests are initiated by user processes and require more work than message forwarding; in essence, message forwarding is “faster” than moves. As a consequence, our current Emerald prototype ignores this problem.

One solution to the problem would be to introduce a *hop count* in each forwarded message indicating how many times the message has been forwarded. When a hop count in a forwarded message reaches some threshold (e.g., the number of nodes in the system) it would still be forwarded, but in addition the node-local descriptor for the target object would be specially marked. Eventually, the moving object will end up on a node where its node-local object descriptor has been marked. The kernel would then delay further moves until the message catches up with it. When this happens, the special marks would have to be cancelled in all affected object descriptors to allow the object to move again. Such cancellation involves sending a message to each node where a mark was left behind—a potentially expensive operation. A more *ad hoc* method would be to delay moving the object for a limited time only, in the hope that such a delay would be sufficient for the lagging messages to catch up. To ensure ultimate success, the length of the delay could be increased as a function of the hop count.

5.3.3 An Alternative Strategy for Forwarding Address Updating

An alternative strategy, which we did not adopt, would be to keep track of all nodes that have references to a particular object. Should the object move, update messages could be sent to those nodes. However, these extra messages could significantly increase the cost of move and of passing references. For example, when an object reference is passed to a node for the first time, that node would have to register with the node responsible for the object. The DEMOS/MP system uses a forwarding address update scheme, and updating forwarding addresses was shown to incur significant overhead. In addition, sending update messages on every move will not avoid the need for invocation forwarding, because update messages do not arrive immediately at all destinations. Our scheme places the cost of forwarding address maintenance on the current users of a forwarding address.

5.4 Recovering Lost Forwarding Addresses

Locating an object for which there is a valid forwarding address is easy: merely follow the forwarding address. Unfortunately, when a node crashes, its forwarding addresses are lost. If an object has moved to another node, access to the object is no longer possible using an access path that includes the crashed node. In the example used in section 5.3.1, consider the situation that arises when node A crashes before Y on node C invokes X using the obsolete forwarding address that points to node A. Node C will not be able to find object X even though X is available on node B.

We have developed a location algorithm (called *Cascading Search*) that includes reliable recovery of lost forwarding addresses The algorithm consists of a series of searches each of which is more expensive and time-consuming than the previous, but also more likely to produce an answer. The final stage ensures that the algorithm provides strong location semantics.

5.4.1 The Cascading Search Algorithm

The steps of the algorithm are as follows:

1. If a forwarding address is known, send a message to the node indicated in the forwarding address asking whether or not the object is there. The remote node replies by returning its forwarding address for the object. If the object has moved then this step is repeated with the new forwarding address. In this way, the path of forwarding addresses is iteratively followed until either the object is found, or the node it points to is down, or a queried node does not have a forwarding address for the object.

2. Broadcast a *Shout* message, requesting a response from the node where the object currently resides.

3. If a reply is received then the object has been located. The new location information is verified by restarting from step 1.

4. If there is no reply after a timeout period then broadcast a *Search* message, asking ALL nodes to respond with either a current forwarding address for the object or a message that no forwarding address is known.

5. If a new forwarding address is received then restart the algorithm at step 1.

6. After an additional timeout period has elapsed, send a point-to-point reliable message to every node that has *not* responded to the *Search* broadcast message.

7. When all nodes have denied hosting the object, or have been declared down by the reliable message delivery subsystem, the algorithm declares the object to be inaccessible and returns a **nil** location.

If the kernel has a forwarding address for the object then step 1 locates the object quickly by following the path of forwarding addresses. If the original forwarding address is up-to-date then the object is located at a cost of two network messages.

Steps 2–7 attempt to obtain a valid forwarding address for the object by exhaustively searching the network. Our experience with an Ethernet-based local area network has been that although broadcasts may be lost, the probability of losing one is very low. Therefore, the exhaustive search starts by merely broadcasting a Shout message—a relatively inexpensive operation compared to sending a point-to-point message to every node. Each node on the net must process the broadcast and thus the broadcast causes a significant amount of processing, albeit uniformly spread out over the nodes. The number of broadcasts that each node must handle per second is bounded by batching location broadcasts together and limiting the broadcast frequency (this is further discussed below).

If the shout broadcast does not produce a result within a short time (currently two seconds), it is assumed that either the object is not available or that its node did not receive the broadcast message. Step 4 begins an exhaustive search of all nodes. This could be performed by querying each node individually but that would require a point-to-point message to be sent to each node. Instead, the search is initiated by broadcasting a *Search* request message. Upon receipt, each node sends back its current forwarding address for the object in question. A list of responding nodes is maintained. Broadcast messages may be lost, so after a suitable timeout period (currently two seconds), step 6 checks the list of responding nodes, compares it to the list of nodes that are believed to be up, and sends out a point-to-point message to the non-responding nodes. Because the messages are sent reliably, the nodes will eventually receive and respond to the request—unless they crash. Node crashes are detected by the reliable message passing module and the kernel is informed of the node state change via an upcall. The upcall performs suitable cleanup actions including checking whether all nodes have responded to any outstanding search request. The exhaustive search requires processing on the order of the number of nodes in the system because a message from each node must be processed. Every other node is required to process two broadcast messages (in steps 2 and 4) and to send one reply message (in step 4). Should a node crash at any time after step 4, the algorithm is restarted at that step. This prevents an object from being declared inaccessible because it has recently moved away from a crashed node.

The algorithm uses exhaustive search; this would not be possible in a large distributed system because of the heavy load caused by every node being queried often. Even in an Ethernet-based system, the load caused by location requests can be significant. To prevent the system from being saturated with broadcasts, the location broadcast requests are not sent out immediately. Instead, they are queued and a daemon is scheduled for execution (if it is not already scheduled). The daemon periodically wakes up, batches the outstanding requests together, and sends out a single broadcast. Depending on the maximum size of a broadcast packet allowed by the underlying local area network, a broadcast packet can contain many location requests. Because our Emerald prototype uses an Ethernet, each broadcast packet can contain hundreds of location requests. Such batching places an upper limit on the number of broadcasts that a particular node can send out per time unit. This is especially useful when a node crashes and a number of forwarding addresses become invalid at once: both the number and the frequency of the broadcasts are limited. The timeout period and the frequency of running the broadcast daemon are currently fixed (although they can be set dynamically). Alternatively, these parameters could be dynamically adapted to the current conditions, for example, by scaling them inversely proportional to either the number of available nodes or the current frequency of location broadcasts.

5.5 Location Summary

We have discussed several different semantics for locating an object within a local area network. Weak location is fast and efficient and works well in most cases. When it does not work, we use strong location. We have shown how Fowler's forwarding address scheme can successfully be used in the context of a local area network and how to extend it with forwarding address recovery using the Cascading Search algorithm.

Chapter 6

Garbage Collection

In this chapter, we start by describing why garbage collection is appropriate in an object-oriented system. We thereafter describe the design decisions aimed at reducing the amount of garbage generated. These decisions have been effective enough to allow our prototype to operate without a garbage collector at all—our collector has been designed but not implemented. We proceed with a description of the two classic types of garbage collectors: reference counting and mark-and-sweep. We then describe our own mark-and-sweep based scheme. It introduces the concept of a *garbage collection fault* similar to a page fault in a virtual memory system. We show how this scheme can collect garbage concurrently with the execution of user processes, and how the scheme extends to the collection of distributed garbage even when faced with node crashes.

6.1 Why Garbage Collect?

Programming languages such as Pascal that use dynamic storage allocation have often left the problem of storage deallocation to the programmer. In these languages, the programmer must explicitly allocate and deallocate storage through calls to storage allocation routines. We see several reasons why such explicit memory management is not desirable:

Errors

Explicit storage allocation and deallocation is error prone (see [Rovner 86]). Common mistakes include neglecting to deallocate storage no longer in use (eventually causing a storage shortage) and deallocating storage still in use (leaving “dangling pointers” that often result in very peculiar and hard-to-find errors).

Transparency of memory management

Memory management is an implementation issue and as such it should be transparent to the programmer [Parnas 75].

Implementation freedom

Hiding memory management from the programmer gives the implementor the freedom to choose alternative (possibly more efficient) implementations (section 4.2.1). The memory management routines are also free to dynamically reorganize memory as called for by copying garbage collectors [Mohamed Ali 84].

Freeing of system resources

Certain system resources may have to be garbage collected. For example, in both the DEMOS/MP system and in our Emerald prototype there is no simple method for discovering that a given forwarding address is no longer required.

Memory management using automatic storage allocation and garbage collection takes care of these concerns but introduces a set of problems of its own.

Garbage collection should not significantly degrade the performance of the system. Performance can be affected in two ways. First, the overall performance may be degraded if too much time is spent garbage collecting. Second, response time may be increased because the application is delayed by garbage collection actions, e.g., classic mark-and-sweep collectors cause long and abrupt pauses.

In object-oriented systems, there is a need for garbage collection. For example, because of encapsulation, a programmer does not have direct access to (or even to know about) subobjects of an object and cannot free them when freeing the object.

Smalltalk [Goldberg 83] relies heavily on garbage collection. The standard Smalltalk implementation generates large amounts of garbage—usually more than one byte of garbage per executed bytecode.[1] As a consequence, considerable effort has been put into designing garbage collectors that exhibit high performance [Lieberman 83, Ungar 84]. However, these collectors were designed for non-distributed systems. A distributed garbage collector is somewhat more complex as it must fulfill additional requirements. It must be able to collect garbage spanning many nodes. It must operate *concurrently* with normal system operation. Because of the lack of shared memory, the collector does not have efficient access to all parts of the distributed object storage. In a distributed system, stopping the entire system while garbage collecting is not acceptable. It is also desirable that the collector be *incremental*, that is, collect smaller amounts of garbage at a time. For example, the collector should be able to collect garbage that does not span more than one node, and even to collect garbage while one or more nodes are down.

Emerald was designed with garbage collection in mind. Section 6.2 describes the design decisions made that affect garbage collection. Most of these are aimed at reducing the amount of garbage created. For example, as opposed to Smalltalk, activation records are not objects and can be stack allocated and deallocated rather than burdening the garbage collector.

[1] One reason that the standard Smalltalk implementation produces large amounts of garbage is that activation records (called contexts) are heap allocated. Thus almost every procedure call produces a large chunk of garbage.

Section 6.3 describes how garbage is collected in conventional systems using either reference counting or mark-and-sweep algorithms. Sections 6.4 and 6.5 describe the Emerald garbage collection scheme which includes a novel algorithm for the collection of distributed as well as non-distributed garbage. The scheme is based on the classic mark-and-sweep algorithm. We call our scheme *faulting garbage collection* because it includes the concept of a *garbage collection fault* similar to a page fault in a virtual memory system. An important property of our faulting scheme is that it can collect distributed garbage concurrently with the execution of user processes. In our design, we have two collectors: one that incrementally collects garbage entirely local to a single node and one that collects garbage spanning several nodes. The two collectors need only synchronize when reclaiming storage—they must not simultaneously reclaim the same object.

In this chapter, we use the term *object* to mean a separately allocated area of memory used to hold an Emerald language-level object (or a kernel pseudo-object). The term also covers certain other data structures (for example, concrete type objects and process stack segments) that are handled by the kernel as if they were objects.

6.2 Design Decisions Affecting Garbage Collection

We had three major goals with respect to garbage collection.

- To reduce the amount of garbage generated, thereby reducing the frequency with which garbage collection is required.

- To support distributed garbage collection in a distributed system with highly mobile objects.

- To support efficient *local* collection of objects within a single node.

As a result, we made a number of design decisions to assist in reaching these goals.

Transparent storage allocation

There is no explicit storage allocation at the language-level. The programmer does not have direct control over virtual memory addresses, e.g., as in the programming language C [Kernighan 78]. The run-time system is therefore free to change the virtual memory address of an object.

Multiple implementations

The compiler chooses a suitable implementation for every object. In some cases, dynamic allocation from a heap can be replaced by stack allocation or avoided entirely, thus reducing the need for garbage collection.

Typed data areas

All data objects contain type information. The compiler also generates type information for process activation records and for processor register sets. The garbage collector can therefore determine the usage of every data object and every variable.

Activation records are not objects

The majority of garbage generated by standard Smalltalk-80 systems is due to the treatment of activation records for methods as language level objects. Baden reports that in a Smalltalk-80 implementation 82% of all storage allocations and deallocations were of activation records [Baden 83]. In Emerald, activation records are not language level objects and are efficiently allocated and deallocated using a stack discipline.

Stacks are data objects

At the kernel level, processes are implemented as a sequence of stack segments each of which is handled as if it were an object (see section 4.3). This reduces the amount of special case code for processes and makes orphan process detection "free" because an orphan process is merely a non-reachable stack segment. As seen from the garbage collector, processes are merely objects; the only special code is the algorithm for finding pointers in the stack segment.

Faulting mechanism

The faulting mechanism developed for handling remote invocations described in section 4.2.5 can readily be used for garbage collection purposes as described below in section 6.4.

These measures have been effective in reducing the need for garbage collection in our Emerald prototype—so effective that the prototype is quite usable even without a garbage collector.

The Emerald garbage collector is based on one of the two classic algorithms. Both are described in the following section.

6.3 Conventional Garbage Collection

In this section, we formulate the Garbage Collection Problem in terms of graphs and describe the two conventional types of garbage collectors: reference counting and mark-and-sweep.

6.3.1 The Garbage Collection Problem

For the purpose of garbage collection, objects may be considered to consist of (zero or more) references to other objects. The set of all objects can be considered a directed graph where each node in the graph is an object and each arc represents a reference from one object to another. If there is an arc from one object to another then the second object is said to be *directly reachable* from the first. Notice that the graph can contain cycles; for example, if two objects have references to each other or if an object references itself. Some objects are considered to be accessible even if there is no reference to them from other objects. The set of such inherently reachable objects is called the *root set*.

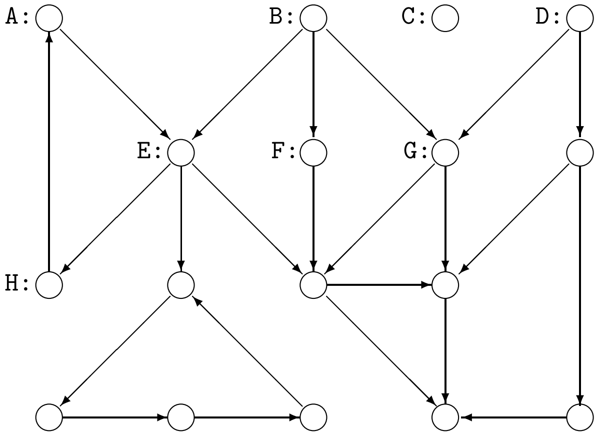

Figure 6.1: A Graphical Representation of Object Storage

Figure 6.1 shows an example of a graphical representation of a set of 17 objects. Object *A* references one object, namely the object *E*; object *B* references three objects: *E*, *F*, and *G*; object C does not reference any other object; objects *A*, *E*, and *H* form a cycle, etc.

The garbage collection problem may now be stated as a graph problem.

> Given a graph of objects and a root set, identify the set of objects that are *not* reachable from the root set.

The non-reachable objects are garbage and can be reclaimed by a garbage collector. In the following sections, we consider the two main algorithms for identifying garbage objects.

6.3.2 Reference Counting

Reference counting collectors operate by counting the number of direct references that exist for each object. The number is stored with the object and is called its *reference count*. It is incremented when a new direct reference to the object is created and decremented when a direct reference to the object is destroyed. If a counter is decremented to zero then the object is unreachable and can be reclaimed immediately. When reclaiming an object, the collector destroys the references stored in the object. This may cause the reference count of a another object to reach zero thus allowing the collector to reclaim that object recursively. The objects in the root set must be handled specially. Usually their reference counts are initialized to one to prevent them becoming zero. Figure 6.2 shows the graph from Figure 6.1 with reference counts added and a root set containing objects *A*, and *C*. The reference counts for the objects in the root set have been initialized to one. The objects with a reference count of zero are

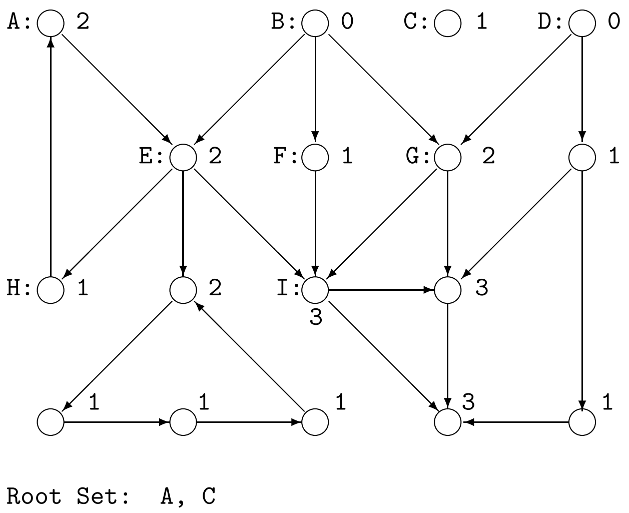

Figure 6.2: Graph Showing Reference Counts

garbage and can be collected, i.e., objects *B* and *D*. Freeing these two objects causes other object to become garbage, e.g., objects *F* and *G*.

Reference counting has the advantage that it is incremental; objects are reclaimed immediately upon becoming garbage and the garbage collection overhead is spread evenly over the computations being performed. This avoids abrupt pauses, which is particularly important in interactive systems such as Smalltalk-80 (see [Wirfs-Brock 83]). However, there are a number of disadvantages:

Cycles Cycles of garbage are not collected. Objects that contain references to each other may be garbage but their reference counts are non-zero. Figure 6.2 contains another example: if object *A* is removed from the root set and object *B* is deleted, then the cycle of unreachable objects *E*, *H*, and *A* remains, although it is garbage. In the Smalltalk-80 system, the problem with circular structures can be partially circumvented by unlinking them before references to them are deleted [Ballard 83]. Many reference counting systems periodically run a mark-and-sweep collector to collect cycles. Vestal proposes a novel scheme for locating and collecting cycles in a reference counting system [Vestal 87].

High overhead The overhead on every pointer operation is high. For example, for an imperative language (such as Emerald), two counters must be modified on every assignment,[2] thereby increasing the execution time for an Emerald assignment statement by a factor of at least four.

Extra storage space The reference counts take up space.

[2] The reference count for the assigned object must be incremented and the reference count for the object previously referenced by the assigned variable must be decremented.

Distribution problems Reference counting has to be modified to work properly in a distributed system because it relies on access to a shared counter. The counter must be distributed or it must be accessed remotely. Either way there are significant problems related to the high cost of network communication and the problem of keeping consistent counts across multiple nodes [Mohamed Ali 84]. Node crashes may cause reference counts to become incorrect.

Each of these problems seems to preclude the use of a reference counting collector for our purposes. For further discussion of these problems, see [Vestal 87].

6.3.3 Mark-and-sweep

The basic idea behind the classic mark-and-sweep algorithm is simple. First, all objects reachable from the root set are marked. Second, the remaining unmarked objects are reclaimed (they are garbage because they cannot be reached from the root set).

More specifically, the *marking* phase marks all reachable objects by following all pointers from the objects in the root set and marking the objects pointed to as reachable. The pointers in the marked objects are then followed and marked and so on until all reachable objects have been marked. Any remaining unmarked objects are garbage. In the *sweep* phase, the garbage collector traverses ("sweeps") the storage area and reclaims the unmarked objects.

The simplest mark-and-sweep collectors require that all nodes are up and that all other activity is stopped during garbage collection. Such abrupt pauses are undesirable in interactive systems. Furthermore, in distributed systems the pauses may be quite long due to the required network communication. For example, in the distributed Eden system, garbage collection of objects on stable storage was performed about once a month by stopping the system for an evening.

Mark-and-sweep collectors associate so-called *coloring bits* with each object:

White indicates that the object currently is not known to be reachable.

Gray indicates that the object is known to be reachable, but the objects that it references have not all been marked as reachable.

Black indicates that the object is known to be reachable and any object that it references has been marked gray or black.

The algorithm initially marks all objects as *white*. Then, the objects in the root set are marked gray and are entered into a *gray set*. The collector iteratively removes an object from the gray set and scans the objects directly referenced by the object. Any white object found is marked gray and entered into the gray set. When all references have been scanned, the object is marked black. A black object is known to be reachable and all its references have been followed. When all gray objects have been scanned, the storage area contains black objects that are reachable and white objects that are garbage. The collector makes one sweep of the entire storage area and collects the white objects. Figure 6.3 shows the graph from Figure 6.1 after it has been marked by the marking phase based on a root set consisting of the objects *A* and *C*. The black objects are reachable; the rest (the white objects) are not and can be reclaimed. For a detailed study of mark-and-sweep algorithm, see [Knuth 68].

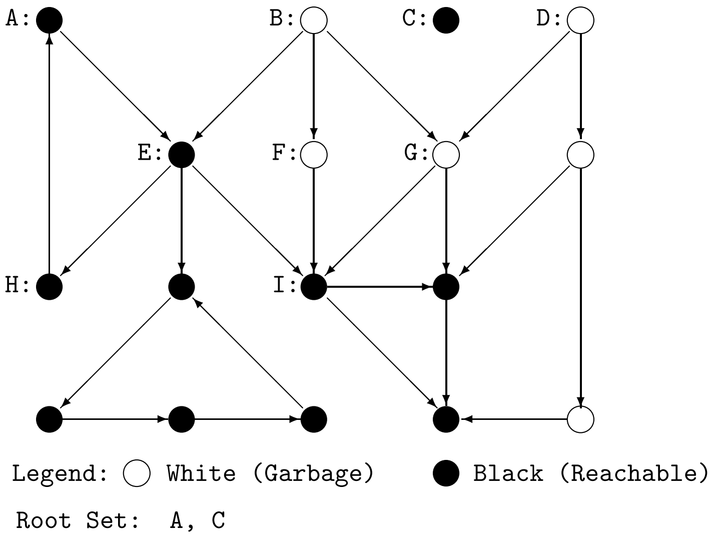

Figure 6.3: Graph after the Marking Phase

The main advantage of mark-and-sweep collectors is that they collect *all* garbage objects including unreachable circular structures. However, there are a number of disadvantages:

Non-concurrent The traditional algorithm is not concurrent; all processes modifying the references in the objects must be stopped during the collection. This is a major disadvantage especially in a distributed system where the classic mark-and-sweep takes much longer than on a uniprocessor due to the extensive network communication required.

Non-incremental All garbage is collected at one time at the end of the collection.

Visits all objects Every reachable object is visited and scanned for references to other objects. In the sweep phase, all objects are visited and garbage objects are returned to the pool of available storage. In a paging environment, this can lead to considerable performance problems [Ballard 83].

Various mark-and-sweep collectors have overcome some of these problems. There have been several suggestions for making mark-and-sweep collectors operate in parallel with the garbage generating processes [Dijkstra 78, Kung 77], some of which have been implemented [Almes 80, Chansler Jr. 82, Bennett 87a]. Dijkstra et al. present a modified mark-and-sweep algorithm that allows concurrency between the collector and processes. The algorithm has several drawbacks: First, it requires cooperation between executing processes (called *mutators*) and the collector. Second, the algorithm repeatedly scans the entire memory. Almes has implemented a version of that algorithm in the object-oriented Hydra system. Typically, a parallel mark-and-sweep collector requires processes to cooperate with the collector by setting color bits of referenced objects when performing assignments. Setting color bits is relatively expensive in Emerald because assignment is very efficient. Furthermore, setting color bits across the net is excessively costly and must be avoided.

Copying collectors (as proposed in [Bishop 77], [Lieberman 83], [Mohamed Ali 84], and [Ungar 84]) avoid visiting dead objects by copying the live objects into a new storage area and subsequently reclaiming all of the old storage area. [Ungar 84] has implemented a Generation Scavenging scheme where the storage area is subdivided; as objects age they are moved to "older" data areas. Incremental garbage collection can then be performed on the younger, more volatile areas.

A problem with the above mentioned collectors is that they require executing processes to cooperate with the collector every time they change a reference. This introduces extra overhead on every assignment and for every parameter passed. Vestal assumes that objects are large and that accessing them requires the intervention of the operating system; thus he assumes that it is acceptable to add a few instructions when accessing an object. This is not the case for our Emerald prototype. The extra overhead is relatively large because node-local operations are very efficient (they are performed directly by compiled code). Our garbage collection scheme, as described next, avoids this extra overhead.

6.4 Faulting Garbage Collection

We avoid extra work on assignment by using a scheme proposed by Baker and Hewitt [Baker 77]. At the start of the marking phase, the data areas accessible to each executable process are scanned and marked black before the process is allowed to run again. These data areas include the objects referenced in the activation records of the process and, transitively, any object reachable from such objects. After an individual process has been scanned and marked, it can proceed in parallel with the rest of the collection even though not all objects and processes have been marked. Should a process become executable (e.g., after waiting for entry to a monitor) then that process must be scanned and marked before being allowed to execute. The central idea is that, as far as an executing process is concerned, *the garbage collection has already finished.* This simplifies synchronization between the collector and processes immensely—indeed, executing processes need not synchronize with the collector at all. The need for special code on each assignment is eliminated. As Baker and Hewitt observe, "the notion that processes as well as storage must be marked may explain some of the trouble that Dijkstra and Lamport had when trying to prove their parallel garbage collection algorithm correct." As a result, the correct algorithm was troublesome to develop and its proof long and subtle.

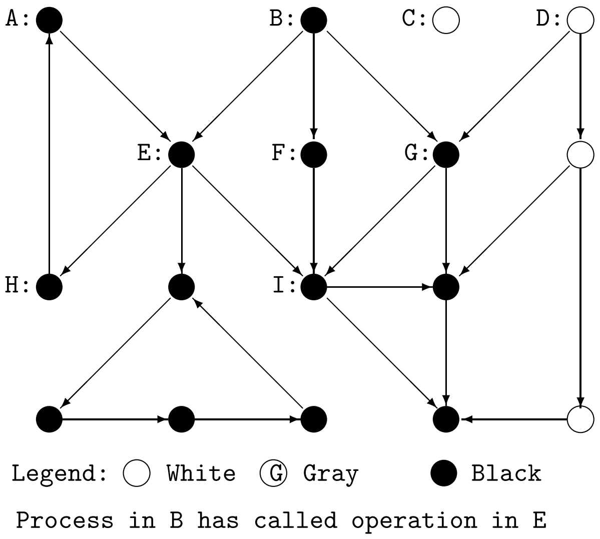
A:
B:
C:
D:
E:
F:
G:
H:
I:
Legend: ○ White Ⓖ Gray ● Black
Process in B has called operation in E

Figure 6.4: Graphed Marked According to Hewitt's Scheme

In the following, we illustrate Hewitt's scheme using Figure 6.4. It shows the graph of Figure 6.1 where we assume that a process started executing in the object *B* and has called an operation in the object *E*. The graph is marked according to Hewitt's scheme as described above, that is, any object reachable by the process executing in object *B* is marked black.

This scheme allows our collector to proceed in parallel with executing processes but there is a high initial cost when making a process executable: *all* objects reachable from the process must be marked. This may involve a significant number of objects. However, many of these objects will not be accessed by the process in the near future. We argue that, due to *locality of reference*, a process will only access, within a given amount of time, a small number of the objects that it can reach. We base this argument on previous work on virtual memory which has shown that the working set of a process is significantly smaller than the total memory of the process. Based on this observation, we have developed a *faulting garbage collection* scheme that significantly reduces the number of objects scanned and marked before a process can be restarted.

When a process is to be marked, instead of scanning and marking all reachable objects, we only scan and mark the objects that the process can currently directly access, namely the objects reachable from the process' top-most activation record. At any given time, a process can only manipulate the references in the current activation record and in the object in which the process is currently executing. This object and the activation record are therefore the only objects that we need to scan and mark as black. In doing so, we mark all directly reachable objects gray. Furthermore, these objects are frozen by setting the *Frozen* bit in their object descriptors thereby preventing them from being invoked. The process may now be allowed to execute. As before, so far as the process is concerned, *the garbage collection is finished* because all objects that it "sees" through its references have been marked. Figure 6.5 show

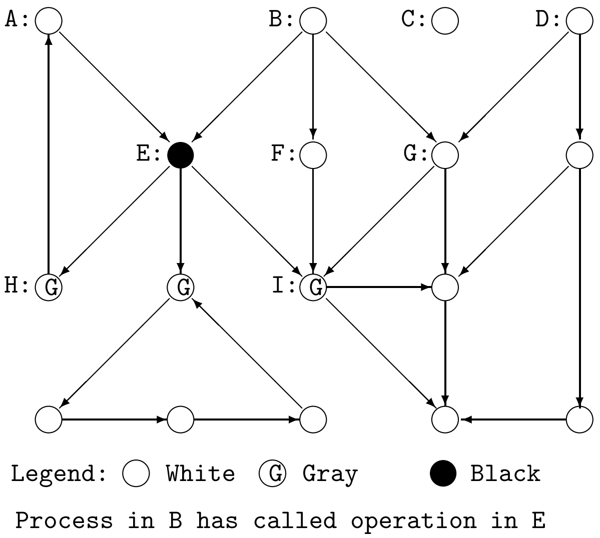

Figure 6.5: Graph after Marking a Process

the situation when a process in B has invoked an operation in E and the graph has been marked according to our scheme. Note the reduced number of marked objects in comparison to Figure 6.4. Any process that subsequently attempts to invoke a frozen object will fault to the kernel exactly as if the object had been remote (as described in section 4.2.5). The kernel makes the collector scan the object allowing it to be marked as black, unfreezes the object, and lets the invocation continue. For example, should the process in Figure 6.5 attempt to invoke the object *H*, then the process will fault to the kernel. The collector marks the object *A* gray and then marks *H* black. The resulting graph is shown in Figure 6.6.

The central idea is to only scan and mark the directly accessible objects in the process' execution environment and subsequently fault when crossing object boundaries. In this way, only a small number of the objects reachable from the process are scanned at a time and processes are allowed to execute after a short delay. Furthermore, the cost of faulting is relatively small because only a single object need be scanned and because the handling of the fault is inexpensive.

A background garbage collector scans the objects in the gray set and marks them black one at a time. Eventually it will have emptied the gray set. Thereafter the collector can enter the *sweep* phase and complete the collection by sweeping the remaining *white* objects. Note that the gray set will eventually become empty because new objects are created black. Consequently, the number of objects that have to be scanned is limited to the objects in existence at the start of the collection.

Our faulting scheme replaces one large delay at the start of the garbage collection with a number of smaller delays spread throughout a process' execution. A process is delayed every

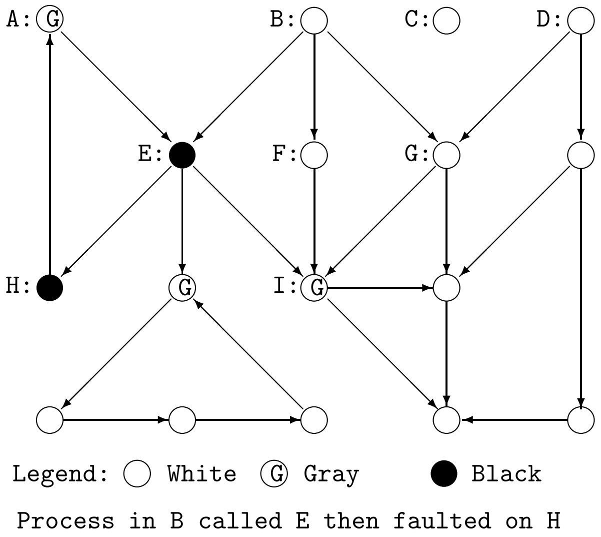
A:
G
B:
C:
D:
E:
F:
G:
H:
G
I:
G
Legend: ○ White Ⓖ Gray ● Black
Process in B called E then faulted on H

Figure 6.6: Markings after Invoke Fault on Object *H*

time it must fault. However, note that it will never fault on the same object twice, so the number of faults it can experience is limited to the number of different objects that it accesses. Furthermore, a fault in our system is relatively inexpensive. In addition, processes are slowed by the background collector.

As mentioned, our scheme essentially eliminates the need for processes to synchronize with the collector. The Emerald processes are free to execute in parallel with the collector; there is no interference between them because the object in which a process is executing has already been marked black. If the process attempts to invoke a gray object, the process faults to the collector which can then resolve the potential interference merely by marking the gray object black. Thus the necessary synchronization has been reduced to a fault. The faulting mechanism is already needed for mobility, so the garbage collection faulting scheme adds no extra overhead during normal non-collecting execution. In summary, our scheme essentially avoids all of the serious concurrency considerations that are found in other collectors [Dijkstra 78, Kung 77, Chansler Jr. 82]. We avoid complicated invariants and the use of a complicated coloring bit scheme as in [Bennett 87b].

There are two minor problems when using the mobility faulting mechanism for garbage collection faults. First, when invoking local objects there is no fault check. To solve this problem, we simply revert to Hewitt's scheme and scan all local objects immediately when scanning their enclosing global object. In the example in Figure 6.7, if object *H* were a local object then we would scan it immediately. This would result in object *H* being marked black and object *A* marked gray. Scanning all locals increases the number of objects that must be inspected before a process can continue. On the other hand, locals are a priori not used outside the object in which the process is executing, so we may hope that, due to locality of reference, local objects are more likely to be invoked than other objects. Thus it is possible that the extra time caused by scanning locals may be offset by a savings in the number of garbage collection faults. Alternatively, we could implement all local objects as global objects at a small extra cost in execution time. Had our invocation sequence been microcoded, we could have added the fault check with little or no extra overhead. Second, when a process returns after an invocation, it will return to an activation record that has not been marked. For example, when the process in Figure 6.5 returns from object E, it will continue executing in the object B which has *not* been marked. One solution is to introduce a fault check on returns for the purpose of garbage collection, but that would introduce extra overhead on *every* invocation return. Instead, we have chosen a solution that avoids extra overhead on returns by performing more work at the start of the collection. Before a process is initially allowed to execute after the start of the collection, we scan the top-most activation record on the stack, traverse down the stack and scan every activation record on it. Thereafter, any object that a process can return to will have been marked black. This increases the number of objects that must be scanned before a process is allowed to run. On the other hand, we avoid the check on every invocation return. Also, because of the stack segmentation scheme described in section 4.3, we only need to scan the activation records within a stack segment. If necessary, the kernel code that performs a return across a stack segment boundary calls the collector to scan the stack segment that the process is returning to. Because stack segments are small, the number of activation records that need be scanned is bounded. Figure 6.7 shows the graph from Figure 6.5 after we have marked all objects accessible from the process that originated in B. Finally, we expect to be able to reduce the number of garbage collection faults by using the following heuristic: when objects are entered into the gray set due to a faulting object, they should have priority when the background collector removes objects from the gray set. In this way, the background collector first marks that part of the graph where

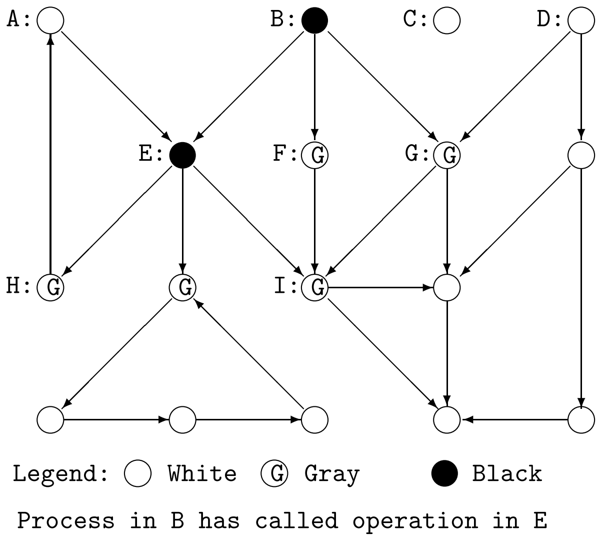
A:
B:
C:
D:
E:
F: G
G: G
H: G
G
I: G
Legend: ○ White Ⓖ Gray ● Black
Process in B has called operation in E

Figure 6.7: Actual Markings Done

there are executing processes, thus reducing the probability of a garbage collection fault for those processes. In a sense, our algorithm starts by marking objects that have been active recently while deferring marking of inactive objects.

6.5 The Emerald Two-collector Design

The Emerald design calls for two collectors. A node-local collector that can be run independently of other nodes and can collect garbage that does not span more than one node, and a distributed collector that requires the nodes to cooperate to collect garbage spanning multiple nodes. In a system with N nodes, there may be up to 2N collector processes running simultaneously—a local and a distributed collector on each node.

In the following the collectors are described as mark-and-sweep collectors but it is worth noting that they could readily be modified to be copying collectors. The modifications for doing so are straight-forward but would only serve to complicate the presentation of the basic ideas behind our collectors.

6.5.1 The Node-Local Collector

We expect most garbage to consist of objects that are created and disposed of on a single node, with no reference ever leaving that node. This is inherently the case for local objects. Hutchinson has shown that a significant number of objects are local not only to a given node but to a specific object [Hutchinson 87a]. Global objects, however, may be referenced across node boundaries. To know which global objects can be collected by the node-local collector, each object descriptor has a flag called the *Reference-Given-Out* bit. The kernel sets this bit in an object's descriptor whenever a reference to the object is passed to another node. The kernel also sets this bit in the descriptor for arriving global objects because the source node retains a reference to the object. By doing so, the kernel ensures that the *Reference-Given-Out* bit is set for every object for which an inter-node reference has existed at some previous point in time.

The node-local collector proceeds as the classic mark-and-sweep algorithm with the following modifications. For a given global object with the *Reference-Given-Out* bit set, it is not possible locally to tell whether or not a reference still exists on some other node and whether or not the object is reachable via such a reference. The collector must be safe (i.e., not delete any reachable object) so it must assume that the object is reachable. The node-local collector initially adds all objects with the *Reference-Given-Out* bit set to the root set. This conservative approach means that objects are marked as reachable even though they may only be reachable through non-resident garbage.

The collector ignores every reference to a non-resident object. We claim that any node-local object reachable through a non-resident object is also reachable through a node-local object with the *Reference-Given-Out* bit set. This is illustrated in Figure 6.8 by an object X resident on node A, an object W_1 resident on node B ($A \neq B$), and a path leading from W_1 to X passing through a chain of objects (W_1, …, W_{i-1}, W_i, …, W_n) where $W_n = X$. Following the path from the first object W_1, we must eventually cross onto X's node (because we end there). Let i be the index of the object W_i where we cross X's node boundary for the last time. The non-resident object W_{i-1} has a reference to W_i, so the *Reference-Given-Out* bit is set for object W_i. Because all objects with the *Reference-Given-Out* bit set are in the root set, the node-local collector will mark W_i as reachable and consequently it will also mark the objects W_{i+1}, …, $W_n = X$ as reachable.

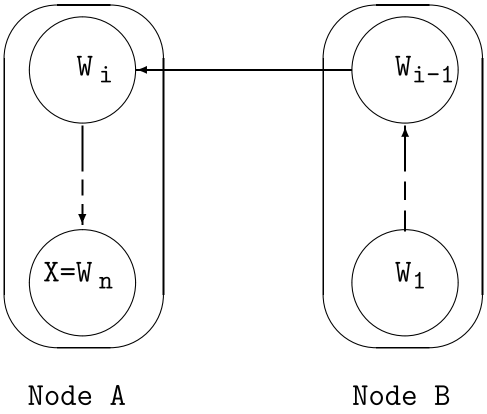

Figure 6.8: A Reference Crossing a Node Boundary

6.5.2 The Distributed Garbage Collector

Distributed collection is performed using a modified mark-and-sweep collection algorithm. By having two sets of marking bits in each object, we allow the distributed and the local collectors to run concurrently. Initially, a collecting process is started on each node, and all global collectors proceed in parallel. All objects are first marked as being unreachable, then each global collector marks all of its explicitly reachable objects gray as in the traditional mark-and-sweep scheme. When attempting to scan a gray object, a global collector may find that the object resides on another node. In that case it sends a mark-gray message to the node where the object resides.[3] The collector on the receiving node marks the object gray (if it is not already gray or black), adds it to the gray set and sends back an object-is-black message when the object has been scanned and marked. Upon receiving an object-is-black message, a collector removes the object from its gray set and marks it black. The marking phase is complete when all nodes have exhausted their gray sets. The collection can then be completed by sweeping all the data areas.

[3] Actually, garbage collection messages need not be sent individually; they may readily be batched together as indicated in the algorithms included in this thesis.

To prevent an object from "outrunning" scan-and-mark requests by moving often, objects are scanned and marked black when moved. This is done even for objects currently marked white[4] because any moved object is a priori reachable and the object would eventually be marked anyway.

If a node is currently unavailable (e.g., has crashed) when a mark-gray message is sent to it then the reference is ignored for the moment. Eventually the only gray references left are to objects on unreachable nodes. At this point, the collectors exchange information about the remaining gray objects so that every collector knows which objects still need to be scanned. When an unavailable node becomes available again, the collectors continue marking gray objects until either the collection is done or there is a gray reference to an object on an unavailable node. The collectors again exchange gray sets and wait for a node to become available. This process is repeated until the collection completes, at which point garbage objects and object descriptors can be collected. Note that it is not necessary for all nodes to be up simultaneously—it is only necessary for each node to be available long enough for the collection to make progress over time.

The algorithm for distributed garbage collection is included in this thesis as Appendix A. It has not been implemented.

6.6 Comparison to Other Work

Bennett's distributed collector [Bennett 87b] is a variant of the collectors presented in [Kung 77] and implemented by Chansler [Chansler Jr. 82]. Bennett's distributed collector prevents objects that are (or have been) referenced remotely from being collected by the node-local collector. It calls upon the node-local collector to provide reference counts and does not actually collect garbage. Instead, references to distributed objects that are to be considered garbage are simply deleted causing them to be collected by the node-local collector. Bennett calls this garbage collection by *prevention*.

The Emerald distributed garbage collector differs from Bennett's collector in that it performs a complete mark-and-sweep collection independently of the node-local collector. The distributed collector can operate concurrently with the node-local collector and collects the garbage it detects.

Ellis et al. [Ellis 88] have developed a copying mark-and-sweep garbage collector based on Baker's real-time algorithm [Baker Jr. 78]. Their collector uses virtual memory hardware to prevent access to unscanned pages of memory. It is similar to the Emerald collector in that it uses a faulting mechanism to allow processes to execute before the collection is finished. Their

[4] It is possible for objects to be moved while still white, for example in Figure 6.7, if object H is attached to E and A attached to object H then moving E would result in moving a white object, namely A.

scheme has processes faulting when accessing object storage while the Emerald scheme has processes faulting when performing invocations. This approach assumes a hardware viewpoint while the Emerald approach assumes a language viewpoint.

6.7 Garbage Collection Summary

We present a novel variation of the classic mark-and-sweep garbage collection algorithm that allows concurrent collection of garbage in a distributed system. It is based on marking of processes as well as objects. We introduce the concept of a garbage collection fault which we use to obtain a concurrent collector. An additional benefit of our scheme is that we get orphan process detection "free."

Chapter 7

Performance

Early object-oriented systems often emphasized the development of object concepts rather than efficient implementation, and therefore suffered from various performance problems. Emerald is in many ways a second generation object-oriented system and is designed with both concepts *and* efficiency in mind. We believe that efficiency cannot be retrofitted—it must be part of the design. Many decisions have significant impact on performance. For example, much effort has gone into alleviating Smalltalk's performance problems (see [Conroy 83] in [Krasner 83]) but ultimately there are inherent problems that no coding trick can solve. As one instance, the lack of a type system in the Smalltalk language implies the need for dynamic binding and type checking (method lookup in Smalltalk terminology). Emerald's unique type system allows the programmer to state nothing, something, or a great deal about the use of a variable; in general, the more information the compiler has, the better its opportunities for performing static binding and generating efficient code.

In the following, we present performance data for specific features of the Emerald system along with comments on the design decisions that affect performance. This data is compared to data for similar features in Eden and in the language C. Besides basic timing data, we have also performed several small experiments to illustrate the performance of the system. The effect of using mobility on network message traffic is illustrated by means of a small (2000 lines of code) application, namely the Emerald mail system. The performance of the mail system is measured when driven by a synthetic workload running on four nodes. Process migration is illustrated by a simple load leveling experiment where compute-intensive processes are migrated to idle nodes to increase throughput.

The Emerald prototype is implemented on top of ULTRIX (DEC's version of Berkeley UNIX). The prototype runs on any VAX or SUN computer that is Berkeley 4.2bsd UNIX compatible. Building on top of ULTRIX has an impact on performance, especially on network operations. Our measurements of the Emerald kernel show that for a remote invocation 88% of the time is spent either in ULTRIX or in routines that send (or receive) Ethernet messages.

The performance data presented in this chapter were obtained from timings done at the

University of Washington and the University of Copenhagen using ULTRIX-32w Version 2.0 running on MicroVAX II or VAXstation 2000 workstations. The VAXstation 2000 workstation has the same CPU as the MicroVAX II but has a different architecture which results in slightly longer execution times (usually about 6–8%). This explains the difference between the timings presented here and those presented in [Jul 88b]. Unless otherwise noted, the timings presented here were done on VAXstation 2000 workstations.

The next three sections present performance data for three of the most important language features: invocation, object creation, and mobility. Thereafter follow two sections describing a mobility experiment using the Emerald mail system, and a process migration experiment.

7.1 Object Invocation Performance

In object-oriented languages such as Smalltalk and Emerald a large percentage of execution time is spent performing invocations. The cost of invocation can therefore be considered the single most important performance number. As a consequence, we have spent considerable effort optimizing object invocation. As described in section 4.2.1, objects can be implemented using one of four different representations. The compiler generates the most efficient invocation sequence possible for each object representation. In the following, we consider node-local invocations separately from remote invocations.

7.1.1 Node-local Invocation

Table 7.1 shows the performance of several local Emerald operations. Integers and reals are implemented as direct objects (i.e., no separate data area is allocated for them). The timings for primitive integer and real operations are exactly the same as for comparable operations in C—which is not surprising because the instructions generated are the same.

Table 7.1: Local Emerald Invocation Timing

Emerald Operation	**Example**	**Time/μs**
primitive integer invocation	$i \leftarrow i + 23$	0.4
primitive real invocation	$x \leftarrow x + 23.0$	3.4
local invocation	localobject.no-op	16.6
resident global invocation	globalobject.no-op	19.4

MicroVAX II timings

For comparison with procedural languages, a C procedure call[1] takes 13.4 microseconds, a Concurrent Euclid procedure call takes 16.4 microseconds, and an Emerald local invocation takes 16.6 microseconds (i.e., 23% longer than a C procedure call). Concurrent Euclid and Emerald are slower because they must make an explicit stack overflow check on each call. C

[1] For a MicroVAX II using the portable C compiler from Berkeley.

avoids this overhead because UNIX uses virtual memory hardware to perform stack overflow checks at no additional cost. If the Emerald kernel had access to virtual memory hardware, it could also avoid the stack check overhead—and the local object invocation time would be the same as in C. Compared to Smalltalk procedure calls, the most notable difference is that Emerald avoids dynamic binding (called method lookup in Smalltalk).

The "resident global invocation" time is for a global object (i.e., one that potentially can move around the network) to be invoked by another object resident on the same node. The 2.8 microseconds above the time for a local invocation are due to the potential for distribution and are devoted to checking whether or not the invoked object is resident. More details on the compiler and its performance can be found in [Hutchinson 87a].

7.1.2 A Local Procedure Call Benchmark

In this section, we focus on the local procedure call mechanism itself. We use Ackermann's function as a benchmark program because most of its execution time is due to procedure calls; the only other operations performed are tests against zero and integer decrement (for details, see [Mendelson 64]). We have written Ackermann's function in Emerald and in C. The Emerald version appears below:

```
function Ackermann[n: Integer, m: Integer] → [result: Integer]
        if (m == 0) then
                result ← n+1
        elseif (n == 0) then
                result ← self.Ackermann[1,m−1]
        else
                result ← self.Ackermann[self.Ackermann[n−1, m], m−1]
        end if
end Ackermann
```

The C version has been written twice: a straight-forward C version and a hand-optimized version. The straight-forward version has been timed both when compiled with and without the C optimizer. We have compared execution times for two sets of parameter values, namely (6,3) and (7,3). Table 7.2 shows the timings for the different versions of Ackermann's function along with the relative difference in execution times normalized to the optimized C version. The Emerald version runs about 50% slower than the C version. When the C optimizer is used, the C timings improve by 12–13%. When carefully hand-coding the C version, the timings improve by an additional 10%.

An analysis of the code generated by the C and the Emerald compilers reveals that the Emerald version is slower than the C version for three reasons. First, as mentioned earlier, Emerald invocations are 23% slower than C procedure calls. Second, Emerald's parameter passing mechanism is more expensive than C's because Emerald also transfers type information. Third, in Emerald all variables are initialized.

Table 7.2: Ackermann's Function Benchmark (in seconds)

Version	(6,3)	Δ%	(7,3)	Δ%
C hand optimized	3.7	−10%	14.9	−10%
C with optimizer	4.1	0%	16.6	0%
C version	4.6	+12%	18.7	+13%
Emerald version	6.6	+61%	27.7	+67%

7.2 Remote Invocation Performance

Table 7.3 shows the elapsed time for various Emerald operations. The times shown are averages of repeated measurements and include the delays caused by periodically scheduled UNIX tasks such as the periodic flushing of the so-called superblock. For historical reasons, we currently use a set of network communications routines that provide reliable, flow-controlled message passing on top of UDP datagrams. These routines are slow: 24.5 milliseconds to transmit 128 bytes of data and receive a reply. The total elapsed time to send the invocation message and receive the reply is 27.9 milliseconds.[2] This means that the actual handling of the remote invocation uses only 3.4 milliseconds (12%) of the 27.9 milliseconds.

For comparison, a remote invocation in the Eden system took 54 milliseconds on a SUN 2. The major difference between the Eden implementation and the Emerald prototype is that the Emerald prototype uses only two network messages to perform a remote invocation while Eden uses four. Bennett's Distributed Smalltalk implementation built on top of Smalltalk uses 136 milliseconds for an "empty" remote invocation [Bennett 87b].

Table 7.3: Remote Operation Timing

Operation Type	Time/ms
local invocation	0.019
elapsed time, remote invocation	27.9
underlying message exchange	24.5
invocation handling time	3.4

MicroVAX II timings

7.2.1 A Breakdown of Remote Invocation Timing

To see where the time goes when performing a remote invocation, we have profiled the Emerald kernel using the *gprof* profiling facility. Table 7.4 shows a breakdown of where the Emerald kernel spends its time when doing one hundred thousand identical empty remote invocations. The breakdown consists of two parts: one for the caller's Emerald kernel and one for the callee's Emerald kernel. In general, the kernel spends only 25% of its time processing the

[2] On VAXstations 2000, the invocation time is 30.3 or about 8% higher, see Table 7.6.

call itself (call, return, and managing light-weight threads). Most of the time, 60%, is spent in message handling while the remaining 15% is spent multiplexing ULTRIX signals. The message module has not been optimized—there is ample room for improvement. For example, the interface to the message module specifies that the sender of a message retains the data area occupied by the message. This forces the message module to make a copy of the message for retransmission purposes.

Table 7.4: Breakdown of Remote Invocation Costs (in percent)

Action	**Caller**	**Callee**
Emerald object code	0.0	0.0
Call	9.0	12.2
Return	9.9	4.4
Thread management	4.3	5.8
Message receiving	20.6	22.4
Message sending	40.0	40.0
Signal multiplexing	16.2	15.2
Total	100%	100%

7.3 Object Creation

Table 7.5 shows creation times for various global and local Emerald objects. The numbers were obtained by repeated timings and are median values—averages do not make sense because the timings vary considerably (up to +50%) when storage allocation causes paging activity. The numbers are correct to about two digits (an error of 1–2%).

Table 7.5: Object Creation Timing (in milliseconds)

Creation of	**Global**	**Δ**	**Local**	**Δ**
Empty object	1.06	+0	0.76	+0
with an **initially**	1.34	+0.28	0.92	+0.16
with a monitor	1.34	+0.28	0.96	+0.20
with one integer variable	1.36	+0.30	0.96	+0.20
with one **Any** variable	1.37	+0.31	0.96	+0.20
with 100 Integer variables	2.41	+1.26	2.01	+1.25
with 100 **Any** variables	3.49	+2.43	3.07	+2.31
with one process	2.17	+1.11	1.85	+1.09

In general, it takes about 0.3 – 0.4 milliseconds longer to create an empty global object than an empty local object because of the additional allocation of an object descriptor. An **initially** construct adds another 0.2 milliseconds as does the presence of a monitor (because the monitor causes an implicit **initially** to be added within which the monitor is initialized). Creating an object containing 100 integer variables costs an extra 1.2 milliseconds of which

0.2 milliseconds are for the implicit **initially** that is generated to initialize the variables and the remaining 1 millisecond is for initializing the integers and additional time for allocating storage. The kernel acquires storage from ULTRIX in large chunks using the (relatively expensive) *sbrk* system call. Allocating larger data areas causes the Emerald kernel to request storage from ULTRIX more often thus inducing additional overhead. This extra overhead can be reduced by increasing the chunk size.

Creating an object containing 100 **Any** variables takes about 2.4 milliseconds longer than an empty object because **Any** variables are 8 bytes long while integers are 4 bytes, so twice as much storage must be allocated and initialized. Much of the object creation time is spent in the memory allocation routines which are designed not for efficiency but for ease of debugging, so there is room for improvement. *gprof* data shows that debugging code in the allocator accounts for $\frac{2}{3}$ of its execution time.

Creating a process takes an additional 1.1 millisecond independently of the object representation and object size. We have measured two processes that take turns calling a monitor. It took 710 microseconds for one process switch, one blocking monitor entry, one unblocking monitor exit, and two kernel calls. By executing two CPU-bound processes and varying the size of the time slice, we have estimated the process switching time to be about 300 microseconds (± 5%) including the necessary ULTRIX calls to set a timer.

7.4 Mobility

Moving a simple data object, such as *simpleobject* in Figure 4.4 (page 69), takes about 12 milliseconds. This time is less than the round-trip message time because reply messages are "piggybacked" on other messages (i.e., each move does not require a unique reply). Moving an object with a process is more complex; as previously stated, while Emerald does not need to move an entire address space, it must send translation data so that the object can be linked into the address space on the destination node. The time to move an object containing a small process with 6 variables is 40 milliseconds. In this case, the Emerald kernel constructs a message consisting of about 600 bytes of information, including object references, immediate data for replicated objects, a stack segment, and general process control information. The process control information and stack segment together consume about 180 bytes.

7.4.1 Parameter Passing Performance

In the following, we compare the incremental cost of call-by-move and call-by-visit with the incremental cost of call-by-object-reference. The following experiment was performed. A remote object was invoked passing it a node-local argument by reference. The remote object then invokes the argument object once. Without call-by-move, this causes a second remote invocation to the original call node (see Figure 3.3 on page 47). When using call-by-move or call-by-visit, the extra remote invocation is avoided because the parameter object is moved to the caller's node (see Figure 3.4 on page 47). The resulting timings are shown in Figure 7.6.

Table 7.6: Remote Parameter Timing

Operation Type	**Time/ms**	**Δ/ms**
remote invocation, no parameter	30.3	+0.0
remote invocation, one integer parameter	31.5	+1.2
remote invocation, local reference parameter	32.5	+2.2
remote invocation, two local reference parameters	34.7	+4.4
remote invocation, call-by-move parameter	35.9	+5.6
remote invocation, call-by-visit parameter	40.3	+10.0
remote invocation, with one call-back parameter	63.5	+30.2

Elapsed time in milliseconds.

Table 7.7 shows the benefit of call-by-move for a simple argument object containing only a few variables (e.g., *simpleobject* in Figure 4.4). The additional cost of call-by-move over call-by-reference is 3.4 milliseconds while call-by-visit costs 7.8 milliseconds extra. The call-by-visit time includes sending the invocation message and the argument object, performing the remote invocation (which then invokes its argument), and returning the argument object with the reply. One would expect the call-by-visit time to be approximately twice the call-by-move time. It is actually slightly higher due to dynamic memory allocation of data structures to hold the call-by-visit control information. Had the argument been a reference to a remote object (i.e., had the object not been moved), the incremental cost would have been 31.0 milliseconds. These measurements are a lower bound because the cost of moving an object depends on the complexity of the object and the types of objects it names.

Table 7.7: Incremental Cost of Remote Invocation Parameters

Parameter Passing Mode	**Time/ms**
empty remote invocation	30.3
call-by-move	+3.4
call-by-visit	+7.8
call-by-reference, one call-back	+31.0

Elapsed times beyond a call-by-reference invocation

The advantage of using call-by-visit depends on the size of the argument object (n), the number of invocations of the argument object (c), the local and remote invocation costs (r and R respectively), and the call-by-visit cost as a function of the argument size ($R_v(n)$). Call-by-visit is worthwhile when

$$R_v(n) + c \times r \leq R + c \times R$$

or, by solving for $R_v(n)$,

$$R_v(n) \leq R + c \times (R - r)$$

R and r are fixed and can only be improved by optimizing the implementation. In the Emerald prototype, the term $(R - r)$ can be simplified to R because $r \ll R$ (about 20 μs versus 30 ms).

To find $R_v(n)$ and to investigate when call-by-visit is worthwhile, we performed the following experiment. Consider the situation in Figure 3.4 on page 47 where one object invokes another object passing it an argument object of size n using call-by-visit. We assume that for the duration of the call-by-visit invocation, the argument object is invoked only by the callee. (We have not analyzed the more complex situation where there are other objects invoking the argument.) The questions is: how many call-backs, c, must be performed before call-by-visit is worthwhile? We assume that the call-backs merely access the data structure and return a single, simple parameter, and that this can be done in approximately the same time as a remote invocation with one parameter, so using Table 7.6 we let $R = 32.5$ ms. Call-by-visit is worthwhile when

$$R_v(n) \leq 32.5 + c \times 32.5$$

or solving for c we get the number of call-backs required to break-even

$$c \geq \frac{R_v(n)}{32.5} - 1$$

Table 7.8 shows the calculated break-even point and the cost in milliseconds of a call-by-visit with one argument object of size n. For objects in the range 20,000-400,000 bytes, call-by-visit costs about 20μs/byte.

Figure 7.1 graphically depicts the same data as Table 7.8 for object sizes up to ten thousand bytes. The figure shows definite "knees" at multiples of about 1800 bytes, roughly occurring when an additional Ethernet packet is required to move the data. These knees also appear when an additional call-back is needed to break-even. The reason for this is simple: When the object size increases enough to require an additional packet each way to move the object, then those two packets could as well have been used to perform an additional remote call-back. In general the performance of call-by-visit is proportional to the number of network packets required, which is directly related to the size of the data moved. We propose the following rule-of-thumb: For objects that can fit in k Ethernet packets, call-by-visit is worthwhile if there are at least k callbacks.

Let us now see what happens if we use faster network software, e.g., by replacing ULTRIX by Distributed V. The remote invocation times R and R_v can be divided into a network component N and an invocation component I:

$$\begin{aligned} R &= N + I \\ R_v(n) &= N(n) + I_v(n) \end{aligned}$$

Using Table 7.7, we get (for small n)

$$R = 30.3, I = 3.4, I_v = 7.8$$

and

$$N = 26.9, N(n) \approx N$$

We use the equation for the break-even point, but this time we calculate the minimum time N for which call-by-visit pays when there is only one call-back invocation ($c = 1$).

$$\begin{aligned} c &\geq \frac{N(n) + I_v(n)}{N + I} - 1 \\ 1 &\geq \frac{N + 7.8}{N + 3.4} - 1 \end{aligned}$$

From this we calculate $N \geq 1.0$ ms, therefore, only if the network component improves by a factor of 25 will call-by-visit not be worthwhile for small objects. For two call-backs, call-by-visit turns out to be worthwhile even for an infinitely fast network ($N = 0$). In summary, our overall conclusion about call-by-visit appears to hold. An improvement in communication speed benefits both the time for a call-back and the time for call-by-visit.

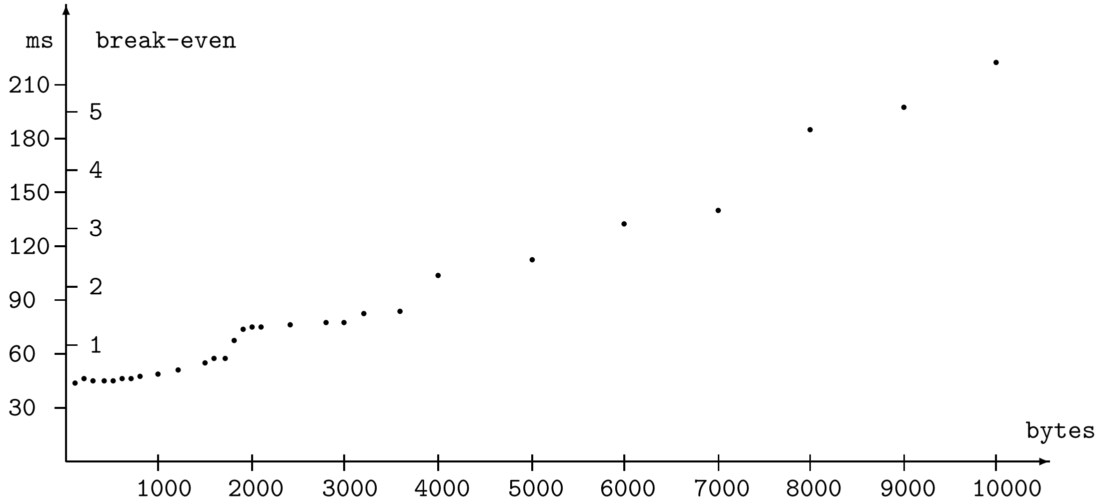

Figure 7.1: The Cost of Call-by-Visit as a Function of Object Size

For argument objects up to 1800 bytes, call-by-visit is worthwhile for even a single call-back invocation, for objects up to 3600 bytes, it is worthwhile for two call-back invocations, and so on as illustrated in Figure 7.1. For larger objects (size > 10000), it appears from Table 7.8 that call-by-visit is worthwhile if there is at least one call-back for every 1600 bytes of data.

7.5 Using Mobility in the Emerald Mail System

We have investigated the performance impact of using mobility in the Emerald mail system, an experimental application modeled after the Eden mail system [Almes 84]. The elapsed time

Table 7.8: Call-by-Visit Timings

Size	$\boldsymbol{R_v(n)}$/s	**Break-even**
100	0.0439	0.35
200	0.0464	0.42
300	0.0450	0.38
400	0.0450	0.38
500	0.0450	0.38
600	0.0458	0.40
700	0.0459	0.41
800	0.0478	0.47
1000	0.0490	0.50
1200	0.0514	0.58
1500	0.0550	0.69
1600	0.0570	0.75
1700	0.0570	0.75
1800	0.0670	1.06
1900	0.0740	1.27
2000	0.0750	1.30
2100	0.0750	1.30
2400	0.0760	1.33
2800	0.0777	1.39
3000	0.0780	1.40
3200	0.0822	1.53
3600	0.0843	1.59
4000	0.104	2.20
5000	0.112	2.45
6000	0.133	3.09
7000	0.140	3.30
8000	0.185	4.69
9000	0.197	5.06
10000	0.222	5.83
12000	0.239	6.35
16000	0.328	9.09
20000	0.413	12
24000	0.472	14
28000	0.565	17
30000	0.614	18
36000	0.722	22
40000	0.818	25
50000	1.06	32
80000	1.66	50
100000	2.13	65
120000	2.50	76
160000	3.35	103
200000	4.21	129
400000	8.58	263

benefit of call-by-move, as shown in Table 7.7, is due primarily to the reduction in network message traffic. We have measured the effect of this traffic reduction in the Emerald mail system. The experiment consists of developing two versions of the mail system—one with mobility and one without—and running both systems using a synthetic workload.

In the Emerald mail system, mailboxes and mail messages are implemented as Emerald objects. In contrast to traditional mail systems, a message addressed to multiple recipients is not copied into each mailbox. Rather, the single mail message is shared between the multiple mailboxes to which it is addressed. In a workstation environment, we would normally expect a mailbox to remain on its owner's private workstation. Only when a person changes workstations—temporarily or permanently—would the mailbox be moved. However, we expect mail messages to be more mobile. When a message is composed, it will be invoked heavily by the sender (in order to define the contents of its fields) and should reside on the sender's node. Upon delivery, mail messages may utilize call-by-move to co-locate themselves with a single destination mailbox. If there are multiple destinations, it is reasonable for the message to stay at the sender's node, but when the message is read it may be profitable to co-locate the message with the reader's mailbox.

To measure the impact of mobility in the mail system, we have implemented two versions:

one which does not use mobility and one which uses mobility in an attempt to decrease message traffic. In the Emerald mail system, the reading of a mail message requires five invocations: one to get the mail message from a mailbox and four to read the four fields. If the mail message is remote then reading the message will take four remote invocations. By moving the mail message, these four remote invocations are replaced by a move followed by four local invocations. However, additional effort may be required when other mailboxes access the message after it has moved.

To facilitate comparison, a synthetic workload was used to drive each of the mail system implementations. Ten short messages (about one hundred bytes) and ten long messages (several thousand bytes) were sent from a user on each of four nodes to various combinations of users on other nodes; the recipients then read the mail that they received.

Table 7.9: Mail System Traffic

Traffic	**Without mobility**	**With mobility**
Total Elapsed Time	71 s	55 s
Remote Invocations	1386	666
Network Messages Sent	2772	1312
Network Packets Sent	2940	1954
Total Bytes Transferred	568716	528696
Total Bytes Moved	0	382848

Table 7.9 shows some of the measurement data collected by the Emerald kernel. As the table shows, the use of mobility more than halved the number of remote invocations, reduced the number of network packets by 34 percent, and cut the total elapsed time by 22 percent. The number of network messages sent is exactly twice the number of invocations; each invocation requires a send and a reply. The number of packets is slightly higher than the number of network messages because the long mail messages require two packets. Note that the number of packets required per invocation is higher with mobility because mobile mail messages cause subsequent message readers to follow forwarding addresses.

Moving the mail messages reduces the total number of bytes transferred slightly (by 7%). Although the same data must eventually arrive at the remote node, whether by remote invocation or by move, the per-byte overhead of move is slightly less than that of invocation. In applications where only a small portion of the data in an object is required at the remote node, invocation might still be more efficient than move.

Finally, it is interesting to note that the 22 percent improvement in execution time was achieved simply by adding the word “move” in two places in the application.

7.6 Process Migration Experiment

Emerald supports full on-the-fly process migration. To investigate the performance of this we performed a simple experiment. Two compute-intensive tasks were executed concurrently and the elapsed time measured. Then the tasks were rerun but this time, after starting the tasks, one was moved to an idle node. The experiment was performed for two types of tasks: a very simple numerical application requiring less than 100 bytes of storage, and a more complicated matrix calculation requiring 100,000 bytes of storage. The elapsed times appear in Table 7.10. The first line shows the time for a single task executed twice and is lower than the second line which shows the time for the same task executed twice in parallel. Executing in parallel on a single node is slower because of CPU-contention and the resulting time-slicing overhead. Off-loading one of the tasks to another node cuts the total execution time almost in half because process migration costs are small—even when the tasks contain 100k bytes of data. From the table, we can estimate the move time for a process of 100k to be 1.45 seconds, including the time spent by the load balancing manager. This means that migrating the process is worthwhile if its remaining execution time is greater than 3 seconds.

Table 7.10: Process Migration Timing

Task	**< 100 bytes of data**		**100k bytes of data**	
	Time/s	**Normalized**	**Time/s**	**Normalized**
One task twice, one node	27.7	1	27.7	1
Two tasks, one node	28.0	1.01	28.0	1.01
Two tasks, two nodes	14.0	0.505	15.3	0.55

7.7 Performance Summary

We have shown how a distributed object-oriented system can provide both efficient node-local operation and fairly efficient remote operations. We believe that the current prototype meets our design goal of local invocation performance commensurate with procedural languages. An important difference between Emerald and previous systems is that we provide excellent performance for co-located, but potentially distributed objects.

Both mobility and remote invocation performance is dominated by the cost of sending messages using the underlying network. To handle a remote invocation the Emerald kernel uses only 12% of the total elapsed time—the remaining 88% is spent passing messages. If we ported the Emerald prototype to an operating system with faster message passing (e.g., Distributed V) then remote invocation performance could be improved significantly.

Fine-grained mobility is inexpensive—especially when combined with invocation using call-by-move. For small objects (up to 1800 bytes) call-by-visit *always* pays if the argument object is accessed at all. For larger objects call-by-visit is worthwhile if there is at least one access to the argument per 1600 bytes of data.

Using the Emerald mail system as an example, we show that call-by-move can yield significant performance improvements. Emerald also provides fine-grained process migration that can improve performance even for processes that execute for a few seconds.

7.7.1 Design Choices That Worked

In the design of the Emerald kernel, we have made a number of design decisions that appear to work well:

- The extensive cooperation between the compiler and the kernel have made many optimizations possible. These optimizations often involved moving work from run-time to compile-time. For example, the compiler generates numerous tables that the kernel can access efficiently at run-time. One such table translates instruction addresses into the address of the failure handler for the instructions. The concept of a failure handler is therefore maintained at zero run-time cost. Another example of compiler-kernel cooperation is that the compiler performs most operation binding. Even when run-time binding is required, it is performed once when a variable is assigned instead of when a variable is used for invoking the object it references. This allows us to avoid the call-time binding (method lookup) that slows Smalltalk.

- Many kernel data structures are implemented as pseudo-objects. This includes threads of control, stacks, and code files. They are handled using many of the same mechanisms as for real objects. For example, code files are referenced and located using the object location algorithm described in Chapter 5. Another example is that by allocating threads of control as objects, orphan process detection is obviated by the distributed garbage collector.

- Performance is improved by using many small caches.

- Dynamic storage allocation is optimized by keeping hot standby lists of recycled, preinitialized data structures.

- The faulting mechanism was originally developed for detecting non-resident objects. However, the mechanism turned out to be useful for solving several other problems, including the initially problem (section 4.2.6) and the problem of concurrent garbage collection.

7.7.2 What Might Have Been Done Differently?

Not all parts of the Emerald kernel have been optimized. Object creation is relatively slow because the storage allocation routines are outfitted with expensive validity checking code.

Another part of the kernel that could be optimized is message passing. As shown in Table 7.4, the message passing routines alone use 60% of the kernel's CPU time when performing remote invocations. They could be improved by changing both their interface and their implementation. For example, these routines were hierarchically implemented and therefore include many levels of procedure calls and much data is repeatedly copied. One reason that these routines have not been optimized is that the network is accessed using UDP datagrams. UNIX's implementation of UDP yields poor performance regardless of how fast the datagrams are composed. Thus faster Emerald remote invocation could be achieved by rewriting the message module and by porting the kernel to an operating system that has fast IPC, for example, Amoeba, which has RPC times of 1.4 milliseconds on SUN 3/50 computers [van Renesse 88].

We provide mechanisms for mobility but do little to aid the programmer who has to make mobility decisions. A problem is that abstract typing hides implementation details that are required when deciding whether or not to use mobility. We have implemented three different parameter modes involving mobility. The decisions of which modes to implement and which to leave alone were somewhat arbitrary; other possiblities may be better.

Chapter 8

Conclusion

We have designed and implemented Emerald, an object-based language and system for distributed programming. The goals of Emerald included:

- support for fine-grained object mobility,
- efficient local execution, and
- a single object model, suitable for programming both small, local, data-only objects and active, mobile, distributed objects.

This thesis has described the language and run-time mechanisms that support fine-grained mobility. While *process* mobility (i.e., the movement of complete address spaces) has been previously demonstrated in distributed systems, we believe that *object* mobility, as implemented in Emerald, has additional benefits. Because the overhead of an Emerald object is commensurate with its complexity, mobility provides a relatively efficient way to transfer fine-grained data from node to node.

The need for semantic support for mobility, distribution, and abstract types led us to design a new language, and language support is a crucial part of mobility in Emerald. While invocation is location-independent, language primitives can be used to find and manipulate the location of objects. The programmer can declare "attached" variables; the objects named by attached variables move along with the objects to which they are attached. More importantly, on remote invocations a parameter passing mode called call-by-move permits an invocation's argument objects to be moved along with the invocation request. Our measurements demonstrate the potential of this facility to improve performance while retaining the advantages of call-by-reference semantics.

Implementing fine-grained mobility while minimizing its impact on local performance presents significant problems. In Emerald, all objects on a node share a single address space and objects are addressed directly. Invocations are implemented through procedure calls or in-line code where possible. The result is that pointers must be translated when an object is moved. Addresses can appear in an object's representation, in activation records, and in registers. The Emerald run-time system relies on compiler-produced templates to describe the format of these structures. A combination of compiled invocation code and run-time support is responsible for maintaining data structures linking activation records to the objects they invoke. A lazy evaluation of this structure helps to reduce the cost of its maintenance.

Our Emerald prototype is operational on a small network of MicroVAX workstations. Through the use of language support and a tightly-coupled compiler and kernel, we believe that our design has been successful in providing generalized mobility without much degradation of local performance.

8.1 Contributions

The major research contributions presented in this thesis are described in Chapter 1 and are summarized here:

- Fine-grained mobility
- Move groups and the concept of attachment
- Efficient object implementation
- Support for multiple object implementations
- The use of strong location semantics
- Faulting garbage collection

8.2 Future Research

Our Emerald prototype provides many opportunities for further research:

Viability of Emerald

We have proven the viability of many of our ideas by construction. However, the real viability test is not in the construction of Emerald but rather in its use. To really prove our ideas, it is necessary to implement a number of applications in Emerald.

Garbage collection

Work is under way to implement our garbage collection design. It remains to be seen whether or not faulting garbage collection is superior to other methods.

Emerald on other architectures

Work is under way to port Emerald to a transputer system. It will be interesting to see how well our design adapts to more closely coupled CPU's.

Protection issues

In Emerald we have largely ignored protection issues. Our only protection scheme is that an object can only be accessed by someone with a reference to it. We also provide protection in the form of restricted object references (the **restrict** operator). However, mobility itself is entirely unrestricted. Anyone with a reference to an object can move the object. Seen from an operating system viewpoint, our Emerald prototype lacks the concept of object ownership—making it difficult to attribute errors to a particular owner.

Language design

There are some language design problems related to immutable objects. According to our definition, immutable objects can have some mutable data structures. This leads to considerable problems when a process is moved while executing in such a structure.

Implementation tuning

As mentioned in the previous section, there are parts of our prototype that need tuning. The message passing routines should be re-engineered.

Mobility policies

An interesting area of research is that of finding suitable migration policies. For example, the kernel could be instrumented to keep track of what objects communicate frequently and possibly migrate them so that they would be co-located. Korry discusses several criteria that could be used by a load balancer [Korry 86].

Message forwarding protocols

Fowler describes several alternative message forwarding policies [Fowler 85]. It would be interesting to implement and measure several of them.

Bibliography

[Almes 80] Guy T. Almes. *Garbage Collection in an Object-Oriented System.* PhD thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1980. Techincal Report CMU-CS-80-128.

[Almes 84] Guy T. Almes, Andrew P. Black, Carl Bunje, and Douglas Wiebe. Edmas: a locally distributed mail system. In *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, Florida, March 1984.

[Almes 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[Baden 83] Scott B. Baden. Low-overhead storage reclamation in the Smalltalk-80 virtual machine. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 19, pages 331–342, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Baker Jr. 78] Henry G. Baker Jr. *Actor Systems for Real-Time Computation.* PhD thesis, M.I.T., March 1978. Technical Report MIT/LCS/TR-197.

[Baker 77] Henry G. Baker, Jr. and Carl Hewitt. *The Incremental Garbage Collection of Processes.* Technical Report AI Memo 454, Massachusetts Institute of Technology Artificial Intelligence Laboratory, December 1977.

[Ballard 83] Stoney Ballard and Stephen Shirron. The design and implementation of VAX/Smalltalk-80. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 8, pages 127–150, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Banawan 87] Sayed Atef Banawan. *An Evaluation of Load Sharing in Locally Distributed Systems.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, 1987.

[Baskett 77] Forest Baskett, John H. Howard, and John T. Montague. Task communication in DEMOS. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 23–31, Association for Computing Machinery, November 1977.

[Bennett 87a] John Bennett. The design and implementation of Distributed Smalltalk. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–330, Association for Computing Machinery, October 1987. ACM SIGPLAN Notices 22(12), December 1987.

[Bennett 87b] John Bennett. *Distributed Smalltalk: Inheritance and Reactiveness in Distributed Systems.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, December 1987. Technical Report 87-12-04.

[Birrell 84] Andrew D. Birrell and Bruce J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Birtwistle 73] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristian Nygaard. *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden, 1973. Published in the U.S.A. by Auerbach Publishers Inc., Philadelphia, PA.

[Bishop 77] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1977.

[Black 85] Andrew P. Black. Supporting distributed applications: experience with Eden. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 181–193, Association for Computing Machinery, December 1985.

[Black 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 78–86, October 1986. ACM SIGPLAN Notices, 21(11):78-86, November 1986.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Brinch Hansen 70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[Brinch Hansen 73] Per Brinch Hansen. *Operating System Principles*. Prentice Hall Inc., Englewood Cliffs, New Jersey, 1973.

[Brinch Hansen 75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[Brinch Hansen 77] Per Brinch Hansen. *The Architecture of Concurrent Programming. Prentice Hall Series in Automatic Computation*, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1977.

[Campbell 74] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions in operating systems. In *Lecture Notes in Computer Science*, Springer-Verlag, 1974.

[Chansler Jr. 82] Robert J. Chansler Jr. *Coupling in Systems with Many Processors*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1982. Technical Report CMU-CS-82-131.

[Cheriton 88] David R. Cheriton. The V distributed system. *Communication of the ACM*, 31(3):314–333, March 1988.

[Clark 85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180, Association for Computing Machinery, December 1985.

[Conroy 83] Thomas J. Conroy and Eduardo Peligri-Llopart. As assessment of method-lookup caches for Smalltalk-80 implementations. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 13, pages 239–247, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Dijkstra 78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[Douglis 87] Fred Douglis. *Process Migration in the Sprite Operating System.* Technical Report UCB/CSD 87/343, Computer Science Division, University of California, Berkeley, February 1987. A revised version of this paper appeared in the 7th International Conference on Distributed Computing Systems.

[Ellis 88] John R. Ellis, Kai Li, and Andrew W. Appel. *Real-time Concurrent Collection on Stock Multiprocessors.* Technical Report 25, Digital Systems Research Center, Palo Alto, California, February 1988.

[Fowler 85] Robert J. Fowler. *Decentralized Object Finding Using Forwarding Addresses.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, December 1985. Technical Report 85-12-1.

[Fowler 86] Robert J. Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proceedings of the Fifth ACM SIGACT/SIGOPS Conference on the Principles of Distributed Computing*, Calgary, Alberta, Canada, August 1986.

[Goldberg 83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Greif 86] Irene Greif, Robert Seliger, and William Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the Thirteenth ACM Symposium on Principles of Programming Languages*, January 1986.

[Herlihy 82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[Hewitt 80] Carl Hewitt. The Apiary network architecture for knowledgeable systems. In *Conference Record of the 1980 Lisp Conference*, pages 107–118, Stanford University, Palo Alto, California, August 1980.

[Hoare 74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoare 78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Holt 83] Richard C. Holt. *Concurrent Euclid, The Unix System, and Tunis.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, January 1987. Technical Report 87-01-01.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language Report.* Technical Report 87-10-07, Dept. of Computer Science, University of Washington, Seattle, Washington, October 1987. Also available as DIKU Report no. 87/22, Dept. of Computer Science, University of Copenhagen, Copenhagen, Denmark and as TR no. 87-29, Dept. of Computer Science, University of Arizona, Tucson, Arizona.

[Ichbiah et al. 79] J.D. Ichbiah et al. Preliminary Ada reference manual. *ACM SIGPLAN Notices*, 14(6A), June 1979.

[Jul 80] Eric Jul. *Structuring of Dedicated Concurrent Programs using Adaptable I/O Interfaces*. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, December 1980. Technical Report no. 82/3.

[Jul 84] Eric Jul. Reliable message passing in Eden; a sliding window protocol for Eden. February 1984. Internal Eden Project Memorandum.

[Jul 88a] Eric Jul. Emerald user's guide. October 1988. Available from DIKU.

[Jul 88b] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions of Computer Systems*, 6(1), February 1988.

[Kernighan 78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Knuth 68] Donald E. Knuth. *The Art of Computer Programming: Volume 1 / Fundamental Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1968.

[Korry 86] Richard Korry. *A Load Sharing Algorithm for a Workstation Environment*. Master's thesis, Department of Computer Science, University of Washington, Seattle, Washington, October 1986. Technical Report 86-10-03.

[Krasner 83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Kung 77] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *Proceedings of the Eighth Annual Symposium on the Foundations of Computer Science*, pages 120–131, October 1977.

[Lamport 78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson 83] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 33–48, Association for Computing Machinery, October 1983.

[Lazowska 81] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eigth ACM Symposium on Operating Systems Principles*, pages 148–159, Association for Computing Machinery, December 1981.

[Levy 88] Henry M. Levy and Edward D. Lazowska. Amber: programming support for networks of multiprocessors. November 1988. Proposal to National Science Foundation.

[Lieberman 83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Liskov 77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[Liskov 79] Barbara Liskov, Russ Atkinson, Toby Bloom, Eliot Moss, Craig Schaffert, Bob Scheifler, and Alan Snyder. *CLU Reference Manual.* Technical Report MIT/LCS/TR-225, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1979.

[Liskov 84] Barbara Liskov. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, Massachusetts Institute of Technology, Laboratory for Computer Science, February 1984.

[Liskov 88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–313, March 1988.

[Mendelson 64] Elliott Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand Company, 1964.

[Mohamed Ali 84] Khayri Abdel-Hamid Mohamed Ali. *Object-oriented Storage Management and Garbage Collection in Distributed Processing Systems.* PhD thesis, The Royal Institute of Technology, S-100 44 Stockholm, Sweden, 1984. Technical Report TRITA-CA-8406.

[Naur 60] Peter Naur. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[Nelson 81] Bruce Jay Nelson. *Remote Procedure Call.* Technical Report CSL-81-9, Xerox Palo Alto Research Center, May 1981.

[Nygaard 70] Kristen Nygaard. *System description by SIMULA An introduction.* Technical Report S-35, Norsk Regnesentral / Norwegian Computing Center, November 1970.

[Parnas 72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Parnas 75] D.L. Parnas and D.P. Siewiorek. Use of the concept of transparency in the design of hierarchically structured systems. *Communications of the ACM*, 18(7):401–408, July 1975.

[Powell 83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 110–119, Association for Computing Machinery, October 1983.

[Pu 86] Calton Pu. *Replication and Nested Transactions in the Eden Distributed System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, Aug 1986. Technical Report no. 86-08-02.

[Pu 88] Calton Pu, Jerre D. Noe, and Andrew Proudfoot. Regeneration of replicated objects: a technique and its Eden implementation. *IEEE Transactions on Software Engineering*, 14(7), July 1988.

[Ravn 80] A.P. Ravn. Device monitors. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.

[Rovner 86] Paul Rovner. Extending Modula-2 to build large integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[Sanislo 84] J. Sanislo. Location dependent primitives user's guide. June 1984. Internal Eden Project Memorandum.

[Schaffert 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–29, October 1986. ACM SIGPLAN Notices 21(11):9-16, November 1986.

[Sollins 79] Karen R. Sollins. *Copying complex structures in a distributed system.* Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1979. Technical Report MIT/LCS/TR-219.

[Spafford 86] Eugene H. Spafford. *Kernel Structures for a Distributed Operating System.* PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, May 1986. Technical Report GIT-ICS-86/16.

[Tanenbaum 81] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Theimer 85] 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 Systems Principles*, pages 2–12, Association for Computing Machinery, December 1985.

[Ungar 84] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–67, Association for Computing Machinery, April 1984. ACM SIGPLAN Notices 19(5), May 1984.

[van Renesse 88] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the world's fastest distributed operating system. *ACM Operating Systems Review*, 22(4), October 1988.

[Vestal 87] Stephen Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, January 1987. Technical Report 87-01-03.

[Wirfs-Brock 83] Allen Wirfs-Brock. Design decisions for Smalltalk-80 implementors. In Glenn Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 4, pages 41–56, Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Wulf 74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[Zayas 87a] Edward R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 13–24, Association for Computing Machinery, November 1987.

[Zayas 87b] Edward R. Zayas. *The Use of Copy-On-Reference in a Process Migration system.* PhD thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1987. Technical Report CMU-CS-87-121.

Appendix A

The Garbage Collection Algorithm

This appendix presents pseudo-C code for the distributed garbage collector algorithm. The algorithm starts on any node which thereafter is the coordinator node. We ignore the problem of electing a new coordinator node should the initial one fail. The algorithm for communicating gray sets when a node crashes is not included.

```
1.      Clearing Phase.
        This marks all objects as not accessible and prevents all processes
        from executing in them.  The distributed part may be delayed
        (see below).  All processes are frozen during this phase.

        graySet = NULLSET;
        forall m: machines do
            wait for m to come up;
            forall x: objects do {
                x.color = white;
                x.gcfrozen = true;
                x.frozen = true;
                /* If it is an executing process then preempt it */
                for all x in readyQ: putQ(GCreadyQ, removeQ(readyQ, x));
        }

2.      Marking Phase:
        During this phase processes are allowed to execute.
        If a process attempts to invoke an object that has not been marked
        the process will be delayed until the object has been marked.
        To get the process running asap, the object is marked immediately.

2.1     Unfreeze the executable processes by removing them from the
        temporary ready queue, marking them, and entering them in the
real ready queue.

        for x in GCreadyQ do {
            traverseAndShade(x);     /* (Expensive) */
            putQ(readyQ, removeQ(GCreadyQ, x));
        }

        Note: as soon as a process has reentered the ready queue it may be
        allowed to run.

        Hereafter the collector proceeds in parallel with the executing
        processes.  The collector is run in the ‘‘background’’ and is
        time-sliced along with the other processes.
```

```
2.2     Shade the root set.
        forall x: object in the root set do
            Shade(x)

2.3     Traverse and mark all gray objects on this node.

        while graySet != NULLSET do {
2.3.1     while graySet contains a resident object do {
            x = graySet.GetOneOf;
            traverseAndShade(x);
            x.color = black;
            x.gcfrozen = false;
            /* Check for other reasons to freeze object */
            x.frozen = calculateFrozenBit(x);
          }
2.3.2     /* At this point there are no more resident gray objects */
          /* Cause a remote shading to take place. */
          forall remoteObject: graySet do
          if (remoteObject.non-resident) then {
            remoteShadeRequest(remoteObject);
          }
2.3.3     Get some more work to do.
          forall m: machines do
            additionalGraySet = m.exchangeGraySet(graySet);
            graySet = graySet UNION additionalGraySet;
        }

2.4     The algorithm completes when no one has any more gray objects.
        The coordinator can check for this using a 2-phase commit protocol:
2.4.1   forall m: machines do {
          ask m to return cardinality of grayset and if zero set
          a timestamp locally.
        }
        /* Second phase */
        forall m: machines do {
          ask m if grayset has remained empty since last time
        }

        If any machine still has a gray object then the algorithm is not
        done yet.
        Otherwise, the coordinator can shift to phase 3, Hunting.

3       Hunting Phase.
        The marking has now been completed.
        All that remains is to hunt down the garbage and reclaim it.

        forall m: machines do
            request m to do
                forall x: object do if x.color == white then reclaim(x);
```

Here are the additional procedures required:

```
Shade(x)            /* Ensure that the object is marked as reachable.*/
        if x.color = white then {
            x.color = gray;
            AddToSet(x, graySet);
        };


traverseAndShade(x) /*   Go thru the object x and shade the objects */
                        that it refers to.
        x.color = black;
        forall ref: References in x do {
            Shade(ref);
            if (ref.isLocal) then {
                traverseAndShade(ref);
            };
        Remove(x, graySet)


remoteShadeRequest(remoteObject)
        send a TraverseAndShadeRequest for remoteObject to its location


CreateObject(concreteType)
        /* Code to be added to the object allocation algorithm.*/
        newObject.color = (if ClearingPhase then white else black);
        return(newObject)

InvokeGCFrozen(x)
        /* Fault handler to be executed when a process experiences a */
        /* garbage collection fault. */

        if x.isResident then traverseAndShade(x)
        else {
            At x.location do the invocation (it might be black already).
        }
        if x.color == gray then Remove(x, graySet);
        x.color = black;  /* Because the invoke has forced it black */

RemoteShadingRequest(x)
        traverseAndShade(x);
```

Vita

Eric Jul was born in Roskilde, Denmark on September 24, 1955. He attended Tuckahoe High School in Tuckahoe, New York from 1969 to 1971 and Frederiksberg Gymnasium, Frederiksberg, Denmark from 1971 to 1973. In 1980 he received a cand. scient. degree (M.S.) from the University of Copenhagen in Computer Science with a minor in Mathematics. The title of his Master's thesis is "Structuring of Dedicated Concurrent Programs Using Adaptable I/O Interfaces." He completed his Ph.D. degree in 1988 and has become a faculty member at the Department of Computer Science, University of Copenhagen, Denmark. He married in 1983 and has three children.