Supporting Distributed Applications: Experience with Eden

Andrew P. Black
University of Washington

The Eden distributed system has been running at the University of Washington for over two years. Most of the principles and implementation ideas of Eden have been adequately discussed in the literature [4]. This paper presents some of the experience that has been gained from the implementation and use of Eden. Much of this experience is relevant to other distributed systems, even though they may be based on different assumptions.

1. Introduction

The Eden project started in September 1980 with the goal of experimentally investigating a particular paradigm for distributed computation. In its fundamentals, the Eden system of today has changed little from that described in the original design paper [20], although the details have evolved as we have moved from one prototype to the next [4]. What *have* changed – as a consequence of two years experience designing, building and using distributed applications – are our ideas about implementation, programming language support, and the relationship between experimental systems and the systems on which one does one's daily work.

We believe that much of this experience is applicable to other distributed systems, even if they are not object based. The purpose of this paper is to share our successes, our failures and the lessons we have learned with as wide a community as possible. The intention is not to preach a gospel. In some cases we started with a particular model, have stuck with it, and are able to report that it has served us well. However, one cannot infer that some other model might not have been as good or better.

For reasons of space, this paper is for the most part confined to lessons relating to distributed systems and distributed programming. Our scope is thus much more restricted than Lampson's *Hints for Computer System Design* [19], which presents excellent advice on system building in general.

The organisation of this paper is as follows. Section 2 sets the context by summarising the basic concepts of Eden and of the Eden programming language, and by giving some idea of the scale of the system. Section 3 describes some of our more significant experiences to date. Some conclusions and directions for further work are presented in Section 4.

2. An Overview of Eden

Eden represents a merging of three distinct threads in operating system design:

- Eden is a complete distributed operating system. In this sense it is rather like the Apollo DOMAIN [21] system, or the various distributed UNIX[1] systems that are now available, e.g., LOCUS [29] and The Newcastle Connection [10].
- Eden is an object-oriented system. Viewed in this way, Eden is a descendant of Hydra [30][31]. It offers the programmer a model of computation significantly different from that of conventional operating systems.

This work was supported in part by the National Science Foundation under Grant Number MCS-8004111. Computing Equipment was provided in part under a cooperative research agreement with Digital Equipment Corporation.

Author's Address: Department of Computer Science FR-35, University of Washington, Seattle, WA 98195. Electronic mail: Black@Washington.

[1] UNIX is a trademark of AT&T Bell Laboratories.

- A third way of viewing Eden is as an implementation of remote procedure call. In this sense it is rather like the RPC System pioneered at Xerox's Palo Alto Research Center [5]. The Eden Programming Language automatically generates stubs for both the caller and the callee; Eden is a very early example of a full-scale RPC system.

Thus, Eden has something in common with several of the distributed operating systems projects of the last few years. However, in combining their advances, Eden provides a unique set of facilities. It is distinguished from systems like LOCUS in being based on a contemporary object-oriented model. In contrast to the Apollo DOMAIN system, Eden's notion of object is definable and extensible by the user. It differs from the Xerox RPC system because objects are mobile, and the binding of a "client" to a "server" is performed potentially on every invocation, rather than in a separate initial phase.

It is important to observe that Eden is not a set of facilities provided on top of an existing operating system in an attempt to graft distribution onto some other model of computation. This is true despite the fact that the current prototype of Eden is implemented using the facilities of UNIX. Eden itself provides the user with a complete environment for program development and execution.

Eden is an integrated system with a single uniform system-wide namespace spanning multiple machines. Its space of objects is managed by the system, in the sense that the system takes care of obtaining resources for the creation of new objects and for garbage collecting objects when they are no longer accessible. Eden objects have the following characteristics:

- Objects are referenced by *capabilities*. Capabilities are not addresses. Rather, each object has a unique identifier. A capability consists of that unique identifier and a set of sixteen rights. The problem of locating an object given only its capability is handled by the system itself. Capabilities are protected from forgery; this ensures that only legitimate clients can access an object.
- *Invocation* is the means whereby one object obtains service from another. It may be thought of as a request message followed sometime later by a response message. Invocation is location independent: in order to invoke an object, the invokee needs to know its capability but not its location.
- Objects are *mobile*, i.e. their physical location within the system changes over time. Because invocation is location independent, clients of an object need not be aware of its migration.
- Conceptually, objects are *active* at all times. Each object has one or more processes within it, and can initiate activities as well as respond to invocations; Eden objects thus subsume the conventional notion of *program*. In practice, an object may voluntarily deactivate (to economise on resources) or be forced to crash (if its code is faulty or its machine crashes). However, if such an inactive object is invoked, the system will reactivate it automatically, so users of an object need not be concerned with these practicalities.
- Each object has a *concrete Edentype*, which may be regarded as a description of the state machine that defines the behaviour of the object, i.e. those patterns of invocation that it will accept and the effect of each invocation. In implementation terms the concrete Edentype is a piece of code in the Eden Programming Language.
- Each object has a *data part*, which includes long-term state representing the data encapsulated by the object and short-term state consisting of the local data of invocations currently in progress, hints, caches, and so on. This data is private to the object in question; other objects can access it only via invocation. Eden thus supports information hiding at the system level; if an object's data structure is found to violate its invariants then the code of only that object need be examined to find out why.
- An object may *checkpoint*, that is, it may atomically write its state to stable storage. Typically, all of the long-term state will be written, and as much of the short-term state as is necessary to achieve appropriate reliability. Only state information that has been checkpointed will be available when an object is reactivated.

The use of Eden has been made significantly easier by the design and implementation of the Eden Programming Language (EPL). This is a modest language based on Concurrent Euclid [14][15], a Pascal extension providing processes, modules and monitors. EPL provides direct support for the fundamental abstractions of Eden, that is, capabilities and invocation. Because EPL provides multiple, light-weight processes, it makes it possible to manage many synchronous remote invocations at one time. This is discussed further in Section 3.1.2.

Both the Eden programming language and the Eden system itself have been available since April 1983, first on a network of four VAX computers, and then on a network of sixteen Sun workstations. About fifty different Edentypes have been publicly released; there are almost certainly an equal number of "private" Edentypes used by small groups of people. A random selection of seventy public and private Edentypes were found to contain over three hundred thousand lines of code, and define about three hundred different invocations.

Our implementation is built on top of Berkeley 4.2 UNIX; we use UNIX to provide us with processes and virtual address spaces, to load code into those address spaces, and to provide access to the disk and the network. The Eden kernel is implemented as a UNIX user process. A local invocation on a Sun workstation currently takes about fifty milliseconds of wallclock time for a null operation and return; a remote invocation takes about seventy milliseconds. Of the fifty milliseconds, just over half is used by the UNIX IPC mechanism; the remainder represents the overhead imposed by Eden's parameter and result packaging, operation and type checking, lightweight process dispatching, object location, capability checking, and so on.

3. Lessons from Eden

The lessons are grouped into four divisions; within the divisions each lesson is introduced by a paragraph, and then motivated by our experience. The first two divisions comprise lessons relating to programming style and lessons relating to naming and distribution. We believe that these are applicable to a wide range of systems. The third category contains lessons that we have learned in the context of Eden objects. While the applicability of some of them may be restricted to systems that provide some notion of "object", several are relevant in other contexts. The final division is more managerial than technical; it discusses some of the difficulties we faced in running a major systems project.

3.1. Lessons relating to Programming Style

3.1.1. Type Abstractly

Objects should be typed abstractly (according to their behaviour) rather than concretely (according to their implementation). The purpose of type checking is to prevent the misinterpretation of values. Its purpose is *not* to prevent multiple implementations of a type from coexisting; neither is it to prevent new types from being added to an existing system, provided that they are interpreted appropriately. A good type system should give warning of misinterpretation, if possible at compile time, without preventing legitimate evolution of the system.

To misinterpret a value of a particular type is to mistakenly treat the collection of bits used to represent it as being a value of some completely different type, for example, to interpret a pointer as a character, or a boolean as an integer. Misinterpretation is a programming error that can be very difficult to track down if it is not detected explicitly; static type checking is thus preferable to dynamic type checking because misinterpretation errors are detected earlier and more completely. Static checking is also more efficient.

Unfortunately, it is clear that not all inter-object type checking can be performed at compile time, if only because the code for the object that one wishes to invoke may not have been written yet. For example, Eden mail message objects now invoke the *Deliver* operation on both mail boxes and mail distribution lists, even though mail distribution lists did not exist when the code for mail messages was written [3].

In an object-oriented language like Smalltalk [12], it is not necessary to perform type checking in order to prevent the misinterpretation of values; the object encapsulation mechanism achieves the same effect. If the // (integer division) message is sent to a boolean object, the result is a "Message not understood" error, not the result of attempting division on the representation of the boolean. However, object encapsulation does not provide early warning of such errors; the Smalltalk programmer who accidentally types

$7 > x // 3$

instead of

$7 > (x // 3)$

is not warned of his error until the code is executed.

In an object-oriented system like Hydra [31], objects are typed. When a procedure is called to act upon an object, it checks that the object is of a particular concrete type. There cannot be a *Deliver* procedure that operates on both mail boxes and mail distribution lists. In Hydra, all attempted misinterpretation of data can be detected at run-time, but at the expense of the flexibility and extensibility that we think are important in a distributed system.[2]

In the Eden programming language, capabilities are not typed; they are like PL/I pointers. There is no way to determine the concrete Edentype of the object to which a capability refers, unless the object chooses to export an invocation that reveals this information. For example, if one object thinks that it is reading from another object of type *Stream*, and issues *transfer* and *close* requests in order to read data, there is no check that the invokee is actually a *Stream*. There *are* checks that the invokee exports *transfer* and *close* invocations, and that the parameter lists of these invocations are correct. As a consequence, the invokee need not be a *Stream*, but can be any object prepared to present the appropriate interface [6][7].

In order to detect type errors at the invocation interface as soon as possible, the Eden Programming Language performs compile time type-checking in addition to the run-time checking described above. When a capability-valued identifier is declared, the programmer asserts that it is for a given Edentype; the compiler then checks that the invocations made on that capability are indeed exported by that Edentype, and that the parameter lists conform. However, the initial assertion is not checked, even if the capability is provided at compile time.

We call this *abstract typing* because the Edentype mentioned in the **for** clause of a capability declaration need not be implemented; it can simply be an invocation interface that abstracts from several implemented (or to-be-implemented) concrete Edentypes. For example, there is no concrete Edentype *Stream*; streams are an abstraction of the interface presented by the bytestore and terminal

[2] Flexibility can be regained through that universal panacea, an extra level of indirection [23]. The Hydra *TypeCall* mechanism is one way of obtaining this indirection, and solves this problem, although that was not its original purpose.

handler Edentypes. It is also sometimes the case that a particular concrete Edentype implements several abstractions; for example, bytestores implement a random access interface in addition to the stream interface.

There is currently no mechanism that allows us to tell whether an object of a particular concrete Edentype conforms to a certain abstract type specification. Given such a mechanism, it would be possible to check the assertion that accompanies capability declarations, and eliminate the checking that accompanies every invocation. This would help us to reduce run-time overhead, while preserving the open-endedness of the Eden type system. We are adopting this approach to type checking in the Emerald language [8].

3.1.2. Use Synchronous Communication

We have found that programmers prefer to use synchronous (procedure-call-like) communication primitives rather than asynchronous (send-receive) primitives. This observation was made in an environment which provided approximately equal syntactic support for synchronous and asynchronous invocation, but also provided multiple processes at low cost.

When the Eden Programming Language was designed, the most appropriate invocation semantics and programming style for Eden were still unknown. We therefore decided to provide a similar kind of language support for both synchronous invocation, which acts rather like a remote procedure call, and asynchronous invocation, where a request is sent and later a response is awaited. In fact, the asynchronous form has been used only rarely, even though we took care to provide equal support for both. (For example, initially both synchronous and asynchronous stubs were produced for every Edentype that was compiled.)

We believe that synchronous invocation was preferred because multiple local processes are directly supported by the constructs of the language. In contrast, multiple remote invocations managed by one local process are not so supported. For example, the monitors and condition variables of EPL make it relatively easy for an EPL process to wait for a condition involving other processes. In a system that does not support local concurrency, or supports it inefficiently, a different conclusion might be reached about the desirability of synchrony. For example, Jones, Rashid and Thompson report that a quarter of all calls that use the Accent stub generator are asynchronous [17]. However, of the various languages for which stubs may be generated, only Ada – not widely used on Accent at the present time – provides lightweight processes and concurrency control. This leads us to the next lesson, the importance of concurrency.

3.1.3. Provide Concurrency – Inexpensively

Our experience is that local processes and synchronisation primitives are powerful tools for the management of remote concurrency. Eden provides concurrency at two levels. Naturally, different objects running on different processors may execute in parallel. In addition, multiple lightweight processes and structures for concurrency control are available within each object.

The Eden Programming Language provides processes that communicate through monitors. We do not claim that monitors are better or worse than, for example, Ada tasks and rendezvous. In fact, our lesson is independent of the particular communication mechanism supported by the language. Because of the various compromises that were made in order to implement EPL rapidly, the model of concurrency provided by our language is very different from that provided by the Eden system as a whole. *In spite of this*, the processes and monitors of the Eden Programming Language have been a major success: the implementation overhead is small, and the EPL code of objects is cleanly structured and easy to understand.

One of the reasons for the importance of concurrent processes within an object is that one object will typically be managing many remote activities simultaneously. This is true both when one is using multiple objects in a computationally intensive task and when one is making calls on remote services. If one has five remote objects performing some computation on one's behalf, the simplest way of managing that situation is to have five local processes, each one of which initiates some remote computation and waits for it to complete, and then finally reports progress back to a coordinating process. In this way we separate the concerns of managing concurrency from those of managing remote operations. Conceptually, this is a much simpler arrangement than having one process that asynchronously starts the five remote processes and then waits for response from any of them, remembers which one has finished, and so on.

3.1.4. Make the Language Support the System

Ideally, a systems programming language and the system that it supports should be designed together. If this is not the case, it is inevitable that the language and the system will present differing computational models. However, each step that can be taken to narrow this difference is of major assistance to the programmer.

Two of the central features of Eden are capabilities and invocations. Capabilities may be thought of as location-independent object pointers; invocations may be thought of as remote operation requests. Both of these features are provided with substantial linguistic support.

Instead of introducing a special invocation construct, the Eden Programming Language packages invocations to look like procedure calls. This choice was made for implementation reasons, and because programmers are familiar with procedure call; nevertheless there are some significant semantic differences between invocation and true procedure call. One of them is that a var parameter to an invocation is strictly a result, whereas a var parameter to an ordinary procedure call is both an argument and a result. Another difference is in the set of types that are understood on both sides of the two kinds of call. A parameter to a procedure call can be of any type that is in scope both at the point of call and in the called procedure body. However, parameters to invocations are limited to a fixed set of system-defined types. This restriction was originally imposed partly to simplify the initial implementation, and partly because there are genuine semantic problems in trying to decide when two types declared in different pieces of code compiled on different machines at different times are equivalent. We had intended to remove the restriction, but the pragmatics of project management have always given other tasks higher priority. Furthermore, in many cases, being prohibited from passing a record structure as an invocation argument is not as bad as it sounds: the record can be implemented as an Eden object, and its capability passed instead.

Our experience with the first prototype of Eden (called Newark) helped motivate the development of EPL. Newark, constructed in the autumn of 1981, was programmed in Pascal and did not have any language support for invocation. Sending an invocation to another object involved fabricating a variant record out of the arguments (to circumvent the Pascal type system), calling the system-provided invocation primitive, and reconstructing the argument list in the remote object. This and other contortions convinced us of the need for an Eden programming language.

Apart from invocation, the other major Eden-specific feature of EPL is the incorporation of a capability type as a first class citizen. By this we mean that capabilities can be manipulated as freely as integers, without programmers having to concern themselves with "C-lists"; the programmer can declare both capability variables and initialised capability constants. Ideally, capability constant initialisation would be done by the editor at program composition time. Because we do not have an object-based program editor, we resort to a more complicated arrangement.

Capability constants are denoted by a string that represents a path through the Eden directory system. This path is interpreted at compile time, with the result that a capability for the appropriate object is bound into the executable image of the newly compiled Edentype. When objects of this type execute, the capability need no longer be present in the directory system at all, and certainly need not be accessible by the client of the object. This facility is useful when confidential information or privileged code is being handled. A simpler scheme would have been to provide a function that returned the capability for the root of the directory system and to perform all lookups at run time. Our arrangement encompasses this simpler scheme; we feel that the flexibility of being able to bind an arbitrary capability into the code of an object, as well as the conceptual simplicity of being able to treat capability constants in the same way as constants of other types, has made the extra implementation effort worthwhile.

3.2. Lessons relating to Naming and Distribution

3.2.1. Distribution: Hidden or Visible?

Various kinds of applications run on a distributed system; some are born to distribution, while others have distribution thrust upon them. In the former case, when it is appropriate for an application to exploit distribution, the system should make it easy to do so. In the later case, the programmer would prefer to be able to ignore distribution. Her task would be greatly simplified if the system were actually centralised, and the function of the operating system ought to be to simulate that happy state.

The Eden mail system and the translator for the Eden Programming Language are examples of applications that have distribution thrust upon them, i.e. they are forced to deal with distribution. Translating the EPL code of an object requires access to a large environment, including the interfaces of all the objects that it invokes. It is impractical to keep current copies of this environment on all machines; it is intolerable to insist that all translations take place on a particular "database" machine because of the excessive load this would place on it. This was the dilemma that faced us so long as we relied on our initial cross-compiler for EPL, which ran on UNIX. Once we ported the translator to Eden, the problem vanished. The various objects that make up the translator can be on any machine in the network, but there need be only a single copy of each file object that makes up the environment. The fact that some or all of these files may be remote need concern neither the writer nor the user of the translator.

By applications that are born to distribution, we mean those that can actually take advantage of the many machines in the network in order to work more effectively – perhaps faster, or with greater availability, or by exploiting particular hardware. If distribution is hidden competely, then the system itself must take the responsibility for all of these things. We believe this is presently infeasible, and also inappropriate in an experimental system.

One application that was implemented on Eden specifically to exploit distribution finds approximate solutions to queuing network models; this involves solving large systems of equations. A coordinator object partitions the equations among several "slave" objects; each slave finds an approximate solution to its part. The coordinator collects these partial solutions, calculates an approximate result, and then (possibly) asks the slaves to iterate. This process is computationally intensive, and the purpose of distributing it is to obtain faster results through parallelism; it is clearly important that each of the slaves run on a different machine. Thus, at some level, the programmer needs access to some location dependent primitives that let him examine the load on different machines in the network and explicitly create his slave objects at specific locations. Once this has been done, the remainder of the application, in which the coordinator object communicates with its subordinates, can reap the benefits of location independence.

The EPL translator mentioned above is also able to exploit distribution. Although the translator will work no matter where the various objects are located, it will clearly work *faster* if some reasonable policy is used to distribute the load over the available machines. However, the part of the translator that determines where to locate its various components is quite separate from the components themselves.

Replication for reliability is another example of an application that needs to take advantage of distribution rather than to have it hidden. Clearly the replicas need to reside on machines with independent failure modes. Our view of location independence is that the basic operations of the system can be carried out on the entities of the system without regard for their location. In terms of the Eden system this means that possessing a capability for an object and invoking that object neither require nor imply any knowledge of its location. Location independence does *not* imply that one cannot make location dependent inquiries and requests, such as asking whether two objects are on the same machine, or that one's checkpoint file should be created on a particular disk. Our work on replication [25] makes extensive use of Eden's location dependent primitives.

3.2.2. Name Uniformly

In Eden, all global system entities are named uniformly: by capabilities. This includes not only objects, but also Edentypes, nodes (machines), and checksites (disks).

The preceding discussion indicates that there is an occasional need to refer to nodes and to checksites as well as to objects. Rather than introducing new namespaces for these purposes, we have named nodes and checksites with capabilities. The concept supported by the location dependent primitives is that of *co-location*; one asks the system to co-locate a pair of capabilities. For example, the kernel call

```
CreateAt(Type : Capability, Place : Capability,
        var NewObject : Capability,
        var Status : Kernel.KStatus)
```

creates a new object with Edentype *Type*, and returns its capability in *NewObject*. If *Place* is a capability for a node, the *NewObject* will be placed there if that is possible. If *Place* is the capability for an active object, *NewObject* will be created on the node on which *Place* is running. Although we have had only limited experience with these primitives so far, it seems that naming machines with capabilities reduces the number of location dependent primitives required and provides a convenient conceptual unity.

3.2.3. Permit Mobility

The mobility of Eden objects is an advantage not shared by many other distributed systems. Mobility enables us to perform load balancing; it also simplifies the task of writing recoverable applications.

We have always intended that Eden objects not be tied to a particular machine but that they move around the system in response to fluctuations in load. This also enables a particular machine to be taken down for maintenance without interrupting operation of the system as a whole. As a consequence, none of our protocols for locating objects relies on an object being where it was last – although that is a useful heuristic. However, at the time of writing, objects cannot move while they are active. In order to move, an object must first request that its next reactivation occur on another node, and then deactivate itself.

We do not see any conceptual problem in moving an active object from one node to another; indeed, the DEMOS/MP system provides this function [24]. However, various practical difficulties have deterred us from completing this part of Eden's implementation. We foresee two classes of problem. The first class is caused by our prototyping environment: moving an object would involve creating a new UNIX process on a new machine with the same code and execution state as those of the old process. This would require performing appropriate translations on the various UNIX namespaces. Were Eden implemented on a bare machine there would be no translations to be performed, as all Eden names are location independent. The second class of problems has to do with invocation messages that are in progress when an object moves; these have to be stored and forwarded by one of the kernels involved.

As a result of the restriction that only passive objects can move, our experiments with load sharing in Eden have been limited to those applications where jobs are moved before they start; once they are running on a particular node they stay there. Being able to move active objects would naturally widen the range of load sharing experiments. In particular, it would let us investigate problems of stability, which seem not to have received much attention. Nevertheless, because all of the entities referred to by an object are named in a location-independent manner, there is still much more scope for load sharing in Eden than in, say, a network of Suns running UNIX.

Another advantage of mobility is that it simplifies recovery. Consider an Eden object executing on a single node, and checkpointing to a disk on that node. Should the node break, the object will be unavailable until it is repaired. Now suppose that we try to increase the availability of the object by replicating its checkpoint file on two additional nodes; the active form is not replicated. When the node on which it is running breaks, our object can be reactivated on another node, using the remotely checkpointed state. Such replication is currently being implemented by introducing a variant of Gifford's voting scheme [11] into the checkpoint mechanism. This is fairly simple to do, and introduces overhead only when the object checkpoints; moreover, the interface to Checkpoint is the same in the replicated and unreplicated cases.

Now consider what would happen if objects could not move. Replicating the checkpoint file would not help, because even though the data would still be available after the crash, the node to run the object would not be. Service could not be resumed until the node is repaired or replaced. In order to increase availability, it would be necessary to replicate the active form of the object, and to employ transaction techniques to ensure consistency between the replicas.

The above discussion also illustrates why we do not rely on forwarding pointers to locate objects that have moved; if the old location of the object is not available, neither is the forwarding pointer. In addition, there are difficulties in determining when the forwarding pointers can be garbage collected.

3.3. Lessons Relating to Objects

3.3.1. System-Supported Objects Improve the Structure of Applications

Our experience with objects has been very positive. This is true despite the inefficiencies of a prototype implementation constructed on top of a conventional operating system. Eden objects subsume the conventional notions of file, value of an abstract data type, and program. They provide a convenient answer to the question of what is distributed in a distributed system.

The major benefit of Eden objects is that they enable one to structure an application in what now seems to be a very natural way. For example, in the Eden mail system [2], not only are the mail boxes of users represented as objects, but so is each message. To people who have been brought up on file based operating systems, the notion of making each message a separate object may seem strange. In fact, it provides a straightforward way to state that messages are an abstract data type with operations that set and interrogate the various fields – the *To* field, the *Subject* field, and so on. Because objects are encapsulated, each message can enforce invariants on its data. For example, mail messages refuse to modify themselves after they have been delivered. A more complicated example arises when a new message is created in reply to an earlier message; the new message can ensure that its *Subject* field is derived from that of the earlier message (possibly by prefixing *Re:*), and that its *To* field contains the earlier message's *From* field.

Another benefit of representing data abstractions as Eden objects is that they can easily be shared. In the mail system, this enables us to achieve some economies; for example, a message sent to a large number of addressees need not be copied into each mail box. Instead, each mail box is given a capability for the message; this also reduces the task of duplicate suppression to a test of equality on capabilities[3].

[3] Strictly, equality of the unique identifiers in the capabilities.

A more significant use of sharing occurs in the Eden appointment calendar system [13]. When a meeting is scheduled for several participants, a single *Event* object is created to represent the details of the meeting. This event is shared by the various calendars, which necessarily hold consistent views of the state of the meeting. In this case Eden enables us to capture succinctly the distinction between a single meeting attended by four people and four separate meetings that happen to occur at the same time.

For objects to be used in such a free way, it is essential that the system allocates and deallocates them automatically. The Eden mail system would be quite useless without a garbage collector that frees the resources used by a message when there are no more capabilities for it in the system. Even if the mail system programmers had been willing to go to the trouble of maintaining reference counts to messages, this would have prevented other subsystems – like the directory system – from keeping capabilities for messages.

Eden objects are not cheap. In our prototype implementation, each Eden object is represented by a UNIX process – and a large one, as it contains a lot of library code that is logically part of the Eden kernel. For example, the code for the *ByteStore* object occupies approximately 122 kbytes on a Sun. Of this, 15 kbytes are code specific to *ByteStore*, 23 kbytes are from the UNIX library, and 84 kbytes are from the Eden library. Even though this code is shared by all the *ByteStores* that are active on a given machine, obtaining it from the disk initially, loading it into a UNIX address space, and subsequently paging it and swapping it, all impose a considerable overhead on the system. Because of this, there were places where we were dissuaded from using Eden objects as generously as we might have liked, and instead relied on the data structures and modules of the Eden Programming Language. Being able to choose between two different representations with different costs enabled us to use the more expensive, system-wide abstraction tool only when it was justified. In a "production" system, objects could be made relatively cheaper, but we still believe that some form of language-level abstraction tool is necessary. The Oz/Emerald project recently begun at the University of Washington aims to integrate the abstraction facilities of a language with those of a distributed system [8].

3.3.2. Eliminate Redundant Concepts

Each Eden object has an Edentype, which contains its code and therefore describes its behaviour. Edentypes are named by capabilities; moreover, they are themselves *ordinary* objects, without special privileges.

We initially imagined Eden to have a three-level type hierarchy, like the class-instance hierarchy of Smalltalk–76 [16]. Each object would be an instance of a concrete Edentype. Ordinary objects would be at the bottom of the hierarchy, and would be instances of Edentypes. Edentypes would be instances of a distinguished object called "typetype", and typetype would be an instance of itself[4].

We have now come to the conclusion that both "typetype" and the distinction between types and instances are unnecessary. Currently, every object is still an instance of a type. However, this type is another ordinary object, of type *typestore*. Instances of typestore are "ordinary" in the sense that the kernel treats them just like other objects; typestores are special only in that their representations contain executable code. Like other objects, typestores provide an invocation interface; indeed, the EPL translator writes them and the debugger reads them by using invocations to access their bytes and capabilities. The type of a typestore is simply the typestore that contains its code.

Apart from the conceptual simplification, the new scheme has a practical advantage. Typetype, as a distinguished object that was an instance of itself, had to be implemented by the kernel. In contrast, typestore is an ordinary Edentype, and is implemented outside the kernel. Moreover, because typestores are not distinguished by the kernel, several implementations of typestore can co-exist. This enables a programmer to create a new kind of typestore, and to test it, without the risk of breaking the translator.

3.3.3. Make Namespaces Large

The need for adequately sized global namespaces is obvious; what was not obvious to us was that one strategy for dealing with small namespaces – making them local – is rather hard to apply.

Unlike Hydra, the rights in an Eden capability are not interpreted by the system itself; that task is left to the object, which is thus free to interpret each bit as it sees fit. At first sight it might seem that sixteen different access rights for each object is ample. In fact, the rights space is inadequate; rights interact with the abstract type system so that in the most general case there are only sixteen different access rights for the whole system. Each abstract Edentype defines its own set of access rights, but since any two abstract types may be combined in a particular concrete type, one abstract type is not free to re-use a right that has been used by another type.

Returning to the mail system example, suppose that the *MailSink* abstract type chose to use right five to represent delivery privilege, while *DirectoryMap* required right five in order to add a new entry. So long as these two abstract Edentypes are unrelated, all is well, but as soon as we create the Mail Distribution List Edentype, which exports the interfaces of both *MailSink* and *DirectoryMap*, it becomes impossible to create a capability permitting delivery to a list without also permitting the addition of new entries.

[4] That is, the type of typetype would be typetype, whence the name.

We do not have a solution to this problem. It is clearly impractical to increase greatly the number of rights so long as sets of rights are part of capabilities. The lesson is that moving the responsibility for the interpretation of a variable from a global agent (the kernel) to a local agent (the object) does not necessarily make the variable's meaning local.

3.3.4. One Object, One Capability?

In Eden, as in most other object-oriented systems, each object is named by exactly one system-wide identifier. An alternative is to allow each object to be named by several identifiers. We have not been able to explore this alternative directly, but some of our experiences indicate that it might be a useful generalisation.

Consider a sequential file object that can be read simultaneously by several clients; for each of these clients the file maintains a "current position" pointer. A client cannot simply invoke the *read* operation on the file, because the file would not know which current position pointer should be used. There are two solutions to this problem. The first is for clients to provide some kind of channel identifier when making a *read* invocation. The second is for the file object to create a sub-object for each open channel, and for each client to *read* from these channel objects rather than from the file itself. Each of these solutions might be appropriate in different situations; however, the interfaces that they present to the client are different, so they cannot be freely mixed.

The Eden Transput protocol achieves a uniform interface to these two different styles of implementation by simulating multiple capabilities. Streams are always represented by *<capability, integer>* pairs; objects that implement a single stream ignore the integer.

Another application for multiple capabilities might be to solve the problem of the small rights space mentioned above. Suppose that instead of interpreting our sixteen rights bits as a set of rights, we interpret them simply as an integer counter that allows each object to have up to 2^{16} capabilities, and that sensitive operations are protected by requiring special capabilities. One of the consequences of this is that a client presently holding a single capability containing a set of rights would be required under the new scheme to hold a set of capabilities, each of which would confer slightly different rights. In practice this might turn out to be intractable. If the application wanted to treat the capability index as a set, it would have to provide invocations to subset it.

Allowing several capabilities for each object represents an interesting compromise between port addressing and the usual kind of object addressing. In systems like Accent [27], each process can receive messages at a number of *ports*. The identity of the receiving port is thus an implicit argument of every request, and ports subsume the uses of multiple capabilities. However, Accent ports are more general than multiple capabilities, because the right to read from a port can be passed from one process to another, whereas the object corresponding to a given capability remains fixed. Retaining the object (rather than the port) as the addressable entity, but allowing several capabilities for each object, may be a useful combination of facilities.

3.3.5. Checkpoint is Inadequate

The Eden *checkpoint* operation enables an object to write its state to disk atomically ; in doing so any previous checkpoint written by that object becomes inaccessible. This mechanism is conceptually very simple, and fulfills the goal of ensuring that the state of an object can survive a crash, thus enabling Eden objects to exist indefinitely. However, with the benefit of hindsight, it is easy to spot the deficiencies of checkpoint: for many objects, it is neither a primitive nor a solution.

The requirement that *checkpoint* be atomic with respect to failures was motivated by the need to write transaction systems, and other applications that recover a consistent state after a crash. However, this same requirement makes checkpoint very difficult to implement efficiently in our UNIX-based prototype. This makes it infeasible for applications to use checkpoint as a primitive, that is, to build more sophisticated recovery schemes from a collection of checkpointing objects. The requirement that the whole state be written at once means that checkpoint is not an appropriate solution when a small change is made to a large object; a logging facility would be more useful in such a case.

It may take as long as a second to perform a checkpoint operation[5]. Most of the blame for this can be attributed to our prototyping environment: updating the disk atomically is not something for which the UNIX file system is particularly well adapted. Indeed, in our initial implementation under Berkeley 4.1 UNIX, it was impossible. Berkeley 4.2 UNIX provides an atomic *rename* system call and an operation that flushes the disk cache for a particular file; these enable us to achieve atomicity, but it remains very expensive. Over eighty per cent of the CPU time of a checkpointing object is consumed by five UNIX file system calls – and that does not include the *write* calls! The overhead is in opening, creating, and especially in linking and unlinking files in the directory system. Similarly, over seventy per cent of the time spent by the kernel process at the checksite is consumed by link, unlink, and open. Database implementors often have similar experiences with the services of conventional operating systems, which turn out either to do the wrong thing, or to have severe performance problems [28].

Another way of looking at the deficiencies of our checkpoint operation is to say that it hides the power of the disk. Object programmers know that disks are capable of random access, and they resent being forced to treat the disk as if it were a magnetic tape. If the file is organised as a list of pages, then a small atomic change can be made simply by creating replacements for a few pages in the file and changing some of the page references in the index. In other words, the disk is capable of atomically changing a small part of a large file, but we do not take advantage of it.[6]

One other attribute of the checkpoint operation comes partially to our aid: it is private. By this we mean that only the object that checkpoints data can read it. It would be quite possible for Eden to provide more than one kind of operation that writes to stable storage. For example, in addition to our current "write everything" form of checkpointing, one could also provide a "logging" checkpoint that atomically adds records to the end of a log file. If an object used a combination of these two methods to maintain its data on stable storage, this complexity would be hidden from the clients of that object. One could envisage several alternative checkpoint implementations, rather like the provision of many access methods in conventional operating systems. But because of the encapsulation provided by Eden objects, this proliferation of methods would not be visible at the interfaces between objects. In contrast, for example, it would not be feasible for Eden to provide several different kinds of invocation.

3.4. Project Organisation

The previous sections have described some of the technical lessons of Eden; this section is concerned with project management and implementation rather than with research issues. Managing a large project is generally recognised to be difficult, and we do not make any claim to be experts. However, we feel that the sub-headings below capture experiences that are interesting and useful to other system builders.

3.4.1. Separate Research from Development

The first major lesson is that one should decide what research one wants to do and to concentrate on doing it; other research problems met along the way should be scrupulously avoided, however interesting they may be. This may sound rather obvious, but in practice it is easy to dissipate a limited amount of manpower amongst too many projects, or worse still, to have one piece of research depend on the successful outcome of another piece of research.

One of Eden's early mistakes was to base the development of its prototype machines on research being performed by the Intel Corporation on the iAPX-432

[5] If the amount of data is small (less than a few kilobytes), the time taken for a checkpoint operation is more or less independent of the size of the data.

[6] Many of these problems could be avoided by using the raw disk, rather than the UNIX file system, to implement checkpoint. But this in turn would cause problems in the implementation of object creation, when it is necessary to execute checkpointed data.

object-oriented processor. I suspect that neither we nor Intel realised that the 432 represented research rather than development. In retrospect, the non-performance of the 432 may have been fortunate: we were sometimes tempted to take advantage of some enticing feature of the 432's architecture to improve the functionality of Eden, and in so doing we were sacrificing portability for efficiency, a premature optimisation. In fact we have recovered well from the non-delivery of the 432, but it probably cost us nine months or more in lost momentum.

One place where we have learned and applied this lesson is in the area of the Eden programming language. After using the Newark prototype, it became clear that some programming language support was necessary to achieve our goal of making distributed programming easy. We made a conscious decision to avoid language design research by choosing an existing language and adding a minimal set of distribution dependent features. Of course, no existing language was entirely suitable. In the end we chose Concurrent Euclid, which was available on a large range of machines and had very good reports from users in industry and academia, but lacked debugging facilities, commercial support[7], and some basic language facilities like union types, procedures as arguments, and garbage collection.

In some places our goal of minimising changes to Concurrent Euclid and to its implementation has adversely affected the programmer's view of EPL. For example, the EPL programmer must explicitly enumerate those data structures that he wishes to checkpoint, and must explicitly manage the capability table to ensure that unused capabilities are purged. However, because of our limited objectives, we were able to implement EPL rapidly without diverting a large number of people from the implementation of Eden itself.

As a result of these compromises we were able to produce a language and an implementation that have been used heavily by Eden programmers. While EPL has its deficiencies, we now have a much better idea of the way programmers actually use a language in a distributed environment. Some of the things we originally regarded as research issues turn out to be unimportant, and some necessary features that we had not previously considered have been brought to our attention. Having gained this experience, we are in a much better position to do research in language design for distributed systems than we were at the inception of the Eden project. This is indeed something that we are now starting to do. However, over the last three years we have been free to concentrate our attention on the systems aspects of Eden.

[7] Concurrent Euclid is distributed by the University of Toronto. Although they were not committed to support the compiler, the response to bug reports was always fast and helpful.

3.4.2. Don't Hide Power

A now famous lesson listed by Butler Lampson [19] is "Don't hide power", and I don't think we did, although this lesson sometimes conflicts with the principles of information hiding and good modular structure.

The Eden kernel provides an asynchronous invocation interface, in which a message is sent and the sender is free to continue processing; the sender will receive an interrupt when the reply arrives. However, the interface that is normally used by a programmer is synchronous: each invocation looks like a procedure call, and the invoking process is blocked until the invocation completes. To make this model feasible, multiple processes are provided within each object. The synchronous interface is implemented out of the asynchronous one by a module called the Dispatcher, which consists of a process that waits for invocation messages and a monitor containing queues of waiting processes.

It was suggested to us during the design of the current implementation that perhaps we should hide the asynchronous interface and present a programmer with just the synchronous procedure-call interface. This of course would give us more freedom for reimplementing the underlying Eden kernel. In a production system this might be the right thing to do; however, in a prototype system like Eden, we thought it best to expose the details of the implementation so that programs could take advantage of them. As mentioned above, the asynchronous interface has rarely been used, and it would be reasonable to hide it in a reimplementation of Eden.

In contrast, there are quite a number of places where UNIX does hide power, and this has caused us a certain amount of grief. One of these places has already been alluded to above: the hiding of the page index in the UNIX file system, which makes the atomic checkpoint of a large file unnecessarily expensive. UNIX also hides the hardware page map and dirty bits, which would otherwise provide a most appropriate mechanism for automating the selection of the appropriate pages to checkpoint. Another example of power hiding occurs in the interprocess communication system of Berkeley 4.2 UNIX.

Eden was initially implemented on Berkeley 4.1 UNIX, which provides no real interprocess communication facilities. In order to build Eden, we adopted the UNIX implementation of Accent IPC [26]. This mechanism is restricted to a single machine, and is thus able to notify the caller if a message cannot be delivered because of congestion. In the absence of such notification the sender can be sure that the message will be delivered.

The interprocess communication system of Berkeley 4.2 UNIX works both locally and remotely. Because of the unreliability of the network, remote messages may be lost; because the IPC interface attempts to hide the difference between the local and remote cases, the message primitives do not guarantee delivery *in either case*. In other words, even in the local case, UNIX does not tell the caller whether an IPC message has been delivered.

Our prototype of Eden uses IPC messages to implement what would be simple kernel calls in a production system. By "kernel calls" we mean requests for service made from an Eden object to its Eden kernel, which is always on the same machine; examples are asking for the time of day, sending an invocation, or performing a checkpoint. These requests must be reliable, and yet the UNIX 4.2 IPC primitives are inherently unreliable. This is a clear example of hidden power: the UNIX kernel knows whether a local message can or cannot be delivered, but it refuses to tell the client process. Our solution here has been to incorporate the Accent IPC mechanism into our UNIX 4.2 kernel, and to use Accent IPC for all Eden "kernel call" communication. An alternative would be to layer an acknowledgement protocol on top of the bare IPC primitives, but the primitives are slow enough that this would be unrealistic.

3.4.3. Use Prototypes

As Fred Brooks said, when building a large system: "Plan to throw one away; you will, anyhow" [9]. We feel that in developing a system of the complexity of Eden it is essential to build prototypes that one can freely throw away. We learned a great deal from the Newark prototype, and the current implementation of Eden is much the better for it. Similarly, the current prototype built on top of UNIX has told us a lot about what is and what is not important in a system like Eden. It is now coming to the end of its useful life, in the sense that some of the improvements that clearly are necessary (like the checkpoint mechanism) need access to the machine at a level lower than that allowed by UNIX.

The use of prototypes also helps to reconcile the need to not hiding power with the need to hide information. In the earliest prototypes one should feel free to expose everything. Later, as one gets a better feeling for which implementation details should not be exposed, one can increase information hiding. A final production version of the system can be fully encapsulated.

3.4.4. Provide Migration Paths

It is easy to underestimate the amount of effort required to produce a system that other people would want to use for their daily work in the way they use UNIX or TOPS-20. Lampson and Sturgis [18] have claimed that a kernel is ten per cent of an operating system. We have created a lot more in Eden than the kernel: a file system, a mail system, a calendar system, a self-hosting compiler for our own programming language, a terminal handler, a command language and an interpreter for it. We are working on debuggers and performance monitors, a transaction system, replication facilities, load sharing tools and a more sophisticated screen-based user interface. In other words, we have built quite a number of the tools and utilities one would expect to find in a complete operating system. Nevertheless, we still have far fewer tools than one finds in a system like UNIX that has been around for fifteen years and has accumulated software from all over the world. It was naive for us to expect that people would clamour to use the Eden system in their daily work, even if distribution was an important part of that work.

We have mitigated this problem in two ways. First, building Eden on top of UNIX, despite the technical problems that have been hinted at above, had the great advantage of providing an easy path by which users could migrate from UNIX to Eden. It also provided an environment in which it was possible to use UNIX tools to accomplish Eden tasks. As an example, at a time when there were no Eden facilities for input from and output to the terminal, it was possible to demonstrate the Eden mail system by using the Emacs editor and a filter process to compose a mail message [2]. Similarly, we have an interface that allows one to use Emacs to edit Eden ByteStores as easily[8] as UNIX files. Second, we actively sought out researchers who could use Eden and gave them positive encouragement and support to do their experiments on it. This was in our own interest: without applications, there can be no experience with the use of the system. And as this paper has tried to show, there is no substitute for experience.

4. Conclusion

We have discussed many of the lessons arising from our experience with Eden, concentrating on those that relate to design choices in distributed systems and to the organisation and management of a large systems project.

Our most significant successes have been the use of abstract object typing and the design of the Eden programming language, which recognises the importance of lightweight processes, concurrency control primitives and parameter packaging for invocations. Neither abstract typing nor language support for distribution are unique to Eden; the Xerox RPC system, for example, has both programming language support (from Lupine) and a notion of "interface" inherited from the Cedar/Mesa language that is similar to an Abstract Edentype. However, our formulation of these ideas in terms of an object-oriented system seems to be novel, and presents research opportunities that we intend to pursue.

Eden's fundamental idea of basing a distributed architecture on objects has been adopted widely. The basic concepts of our particular notion of Eden object – location independent invocation, abstract typing, mobility and encapsulation – have proven to be remarkably powerful and flexible. Nevertheless, some parts of our object model need modification. In this paper I have discussed the need for better primitives for maintaining state on stable storage and the problems with access rights (see Sections 3.3.5 and 3.3.3), and have mentioned the possibility of allowing multiple capabilities for a single

[8] As easily, but not as rapidly.

object (Section 3.3.4).

One question that in our view remains open is the provision to be made for atomicity. Eden provides an atomic checkpoint primitive, but does not provide a general-purpose atomic action system. Thus, programmers who so desire can build robust and secure applications, but to do so they must construct their own transaction mechanism. This is a different approach from that taken by, for example, the Argus system [22] and the Clouds system [1], where atomic actions are among the basic building blocks provided at the system level. In the Eden Calendar system, which contains its own support for a particular kind of atomic action, we estimate that about ten per cent of the code is concerned with maintaining the consistency of the distributed database of calendars. On one hand, this is not a lot; on the other hand, the transaction code is probably some of the most complicated in the application. And yet, because transactions are built as part of the calendar system, we are able to take advantage of the semantics of the calendar and event objects so as to reduce the number of writes to stable storage, compared to the number that would be required by a general purpose transaction system.

An obvious remaining question is where the Eden project goes from here. We do not currently plan to re-implement Eden on a bare machine; the cost of doing so would be high, and the payoff in terms of new knowledge about supporting distributed applications would be low. Of course, it would be pleasant to have a "real" implementation of Eden that could be used for our daily work, but building one is probably a task for a company rather than a university.

We do plan to continue with our experiments in load sharing, replication, transactions, concurrency control, graphical interfaces, distributed programming languages, type checking and checkpointing. Some of the personnel involved with these projects are using the Eden system as it now exists, some and are making experimental modifications (for example to allow a limited form of replication), and one group is constructing a new distributed system [8] whose initial prototype is based on parts of the Eden implementation.

5. Acknowledgements

Over fifty faculty, staff and students contributed to Eden during its evolution. It is impractical to acknowledge the contributions of these people individually, but collectively they have been responsible for creating and using the Eden system, and thus providing the experience that this paper has tried to capture. Jan Sanislo deserves especial mention for overseeing the maintenance and improvement of the Eden implementation. Dick Watson provided constructive criticism of a previous version of this paper. Guy Almes and Ed Lazowska shared in and discussed some of the experiences mentioned herein; nevertheless, responsibility for the opinions expressed in this paper rests with the author.

References

[1] Allchin, J. E. and McKendry, M. S. "Synchronization and Recovery of Actions". *Proc. 2nd Symp. Principles Distributed Computing*, August 1983, pp31-44.

[2] Almes, G. T., Black, A. P., Bunge, C. and Wiebe, D. "Edmas: A Locally Distributed Mail System". *Procs 7th Int'l Conf Softw. Eng.*, March 1984, pp56-66.

[3] Almes, G. T. and Holman, C. "Edmas: An Object-Oriented, Locally Distributed Mail System". Technical Report 84-08-03, University of Washington, Department of Computer Science, August 1984.

[4] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D. "The Eden System: A Technical Review". *IEEE Trans. on Software Eng. SE-11, Nr 1* (January 1985), pp43-59.

[5] Birrell, A. D. and Nelson, B. J. "Implementing Remote Procedure Calls". *Trans. Computer Systems 2, Nr 1* (February 1984), pp39-59. Presented at 9th ACM Symp. on Operating System Prin..

[6] Black, A. P. "An Asymmetric Stream Communication System". *Proc. 9th ACM Symp. on Operating System Prin.*, October 1983, pp4-10.

[7] Black, A. P., Brower, J. P. and Korry, R. "The Abstract Type STREAM in Eden". Eden Project Internal Document, University of Washington, Computer Science Dept, Seattle, WA, July 1984.

[8] Black, A. P., Hutchinson, N., Jul, E., Levy, H. M. and Carter, L. "Distribution and Abstract Types in Emerald". Technical Report 85-08-05, University of Washington, Computer Science Dept, August 1985.

[9] Brooks, F.P.Jr. *The Mythical Man-Month, Essays on Software Engineering*. Addison-Wesley, 1975.

[10] Brownbridge, D. R., Marshall, L. F. and Randell, B. "The Newcastle Connection, or Unixes of the World Unite!". *Software—Practice & Experience 12* (1982), pp1147-1162.

[11] Gifford, D. K. "Weighted Voting for Replicated Data". *Proc. 7th ACM Symp. on Operating System Prin.*, December 1979, pp150-159.

[12] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, May 1983.

[13] Holman, C. and Almes, G. T. "The Eden Shared Calendar System". Tech. Rep. 85-05-02, University of Washington, Computer Science Dept, May 1985.

[14] Holt, R. C. "A Short Introduction to Concurrent Euclid". *SIGPLAN Notices 17, Nr 5* (May 1982), pp60-79.

[15] Holt, R. C. *Concurrent Euclid, The Unix System, and Tunis*. Addison-Wesley, 1983.

[16] Ingalls, D. "The Smalltalk-76 Programming System". *Conf. Rec 5th ACM Symp. on Prin. of Prog. Lang.*, January 1978, pp9-16.

[17] Jones, M. B., Rashid, R. F. and Thompson, M. R. "Matchmaker: An Interface Specification Language for Distributed Processing". *Conf. Rec. 12th ACM Symp. on Prin. of Prog. Lang.*, January 1985, pp225-235.

[18] Lampson, B. W. and Sturgis, H. E. "Reflections on an Operating System Design". *Comm. ACM 19, Nr 5* (May 1976), pp251-265.

[19] Lampson, B. W. "Hints for Computer System Design". *Proc. 9th ACM Symp. on Operating System Prin.*, October 1983, pp33-48.

[20] Lazowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R. and Vestal, S. "The Architecture of the Eden System". *Proc. 8th ACM Symp. on Operating System Prin.*, December 1981, pp148-159.

[21] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L. and Stumpf, B. L. "The Architecture of an Integrated Local Network". *IEEE J. Selected Areas Communications SAC-1, Nr 5* (November 1983), pp842-857.

[22] Liskov, B. and Scheiffer, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *Conf. Rec. 9th ACM Symp. on Prin. of Prog. Lang.*, 1982.

[23] Needham, R. Quoted on p 279 of [31].

[24] Powell, M. L. and Miller, B. P. "Process Migration in DEMOS/MP". *Proc. 9th ACM Symp. on Operating System Prin.*, October 1983, pp110-119.

[25] Pu, C., Noe, J. and Proudfoot, A. "Regeneration of Replicated Objects: A Technique for Increased Availability". Tech. Rep. 85-04-02, University of Washington, Computer Science Dept, April 1985.

[26] Rashid, R. F. "An Inter-process Communication Facility for Unix". CMU-CS-80-124, Computer Science Dept, Carnegie-Mellon University, Pittsburgh, PA, February 1980.

[27] Rashid, R. F. and Robertson, G. G. "Accent: A communication oriented network operating system kernel". *Proc. 8th ACM Symp. on Operating System Prin.*, December 1981, pp64-75.

[28] Stonebraker, M. "Operating System Support for Database Management". *Comm. of the ACM 24, Nr 7* (July 1981), pp412-418.

[29] Walker, B., Popek, G., English, R., Kline, C. and Thiel, G. "The LOCUS Distributed Operating System". *Proc. 9th ACM Symp. on Operating System Prin.*, October 1983, pp49-70.

[30] Wulf, W., Levin, R. and Pierson, C. "Overview of the Hydra Operating System Developement". *Proc. 5th ACM Symp. on Operating System Prin.*, 1975. Austin, TX.

[31] Wulf, W. A., Levin, R. and Harbison, S. P. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.