The Eden Project: Overview and Experiences

Andrew P. Black

Department of Computer Science
University of Washington
Seattle, Washington (USA)

ABSTRACT

The Eden project is a five year experiment in designing, building and using an integrated distributed computing system. The Eden system attempts to combine the benefits of integration and distribution by supporting an object-based style of programming on a number of node machines interconnected by a local network. The hypothesis of the Eden project was that this environment would be conducive to building distributed applications; at the scientific level, the goal of the project was to test this hypothesis experimentally, which involved building a prototype Eden system and implementing a variety of applications on it. Additionally, Eden has been used as a tool in other research related to distribution [17] [15], and has motivated research in programming languages for distributed applications [7] [8].

This paper presents an overview of the Eden project and some of the Eden applications. It also describes some of our experience as builders of a large distributed system. The bibliography directs interested readers to more detailed papers on specific topics; a more detailed but less current introduction to the structure of Eden itself will be found in reference 4.

1. An Overview of the Eden System

Eden represents a merging of three distinct threads in operating system design. First, Eden is a state-of-the-art object-oriented system. Viewed in this way, Eden is a descendant of Hydra [19]. Second, Eden is a complete distributed operating system. In this sense it is rather like the Apollo DOMAIN [12] system or UCLA's LOCUS system [18]. Third, Eden includes a full-scale implementation of Remote Procedure Call (RPC). In this sense it is rather like the RPC system pioneered by Xerox [5]. Eden thus has something in common with a number of the advanced operating system projects of the last few years. However, in combining the advances of these systems, Eden provides a unique set of facilities. It is distinguished from systems such as LOCUS in being based on a contemporary object-oriented model. It is distinguished from the contemporary implementation of the Apollo DOMAIN system by having a notion of object that is definable and extensible by the user. It is distinguished from the Xerox system by the fact that objects are mobile and that the binding of a client to a server is performed upon every invocation rather than just once. And of course it is a significant advance over the Hydra system in that distribution is an integral part.

It is important to observe that Eden is not a collection of facilities provided on top of an existing operating system in an attempt to add distribution to a conventional style of computation.

This work is supported in part by the National Science Foundation under Grants MCS-8004111 and DCR-8420945. Computing equipment was provided in part under a cooperative research agreement with Digital Equipment Corporation.

This is true despite the fact that the current prototype implementation of Eden is built using the facilities of UNIX†. UNIX is merely an implementation vehicle; Eden itself provides the user with a complete, advanced environment for the development and execution of distributed applications.

Eden objects, the basic building blocks of distributed applications in Eden, exhibit the following characteristics:

- *Invocation* is the means whereby one object obtains service from another. It may be thought of as a request message followed some time later by a response message.
- Invocation is *location independent*; one object does not need to know the location of another object in order to invoke it.
- Objects are addressed by *capabilities*. Capabilities are not addresses. Rather, each object has an unique identifier. A capability consists of that unique identifier and a set of rights. The problem of locating an object given only its capability is handled by the system itself. Capabilities are protected from forgery by the system.
- Objects are *mobile*.
- Objects are *active*. Each object has one or more processes within it. This stands in contrast to Smalltalk objects, where threads of control enter an object when a request is made but leave it when the response is completed. Eden objects can perform activities on their own behalf, as well as in response to invocations.
- 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 invocations that it will accept and what their effect will be. 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.
- An object may *checkpoint*. This is an atomic way of writing its state to stable storage. The data is written under the control of the concrete Edentype of the object. Typically all of the long-term state will be written, and as much of the short-term state as is necessary to achieve the reliability specification of the object.
- Finally, objects are *activated automatically* when they are invoked, if that proves to be necessary. Conceptually we wish to regard objects as active at all times.

These characteristics of Eden objects lend the following unique characteristics to the Eden system as a whole:

- First, it is an integrated system with a single uniform system-wide namespace. Its space of objects is managed by the system, in the sense that the system takes care of obtaining resources for the creation of a new object and for garbage collecting objects when they are no longer accessible.
- Eden supports the notion of abstract Edentypes. Several different pieces of concrete code can implement services that at some level of abstraction may be considered as identical. For example, there may be two concrete directory types which support the same set of operations but which exhibit different reliability and performance characteristics. When invoking a directory the invoker need not be concerned with which concrete type is actually used. This simple idea permits us to support multiple inheritance hierarchies in the sense of Smalltalk, the asymmetric stream concept, and many other novel and promising ideas. This

† UNIX is a Trademark of AT&T Bell Laboratories.

idea is discussed further in section 5.

- The third consequence is that data encapsulation (information hiding) is supported and enforced by the system. The only code that accesses the representation of an object is the code that makes up that object's concrete Edentype. If the object's data structure is found to violate its invariants then only the object's own code need be examined to find out why. Similarly, if a change in use requires that a data structure be modified, all the code that needs to be modified is within the object itself.
- Fourthly, objects are secure. The system ensures that only through possession of a capability can an object be accessed. The system also ensures that capabilities cannot be forged. The access rights in a capability enable us to provide fine-grained restrictions on access.
- Another consequence of our design is that Eden does not offer automatic insulation from crashes. A general-purpose atomic action system is not one of the primitives that Eden provides. We *do* provide the atomic checkpoint primitive, whereby programmers who so desire can build robust and secure applications. This is a different approach from that taken by, for example, the Argus system [14] [13] and the Clouds system [1], where atomic actions are among the basic building blocks provided at the system level. In Eden, it is possible to experiment with different approaches to providing transactions [17].

2. Implementation

The Eden system has been operating on a collection of Sun workstations since autumn 1984, and on a network of VAX systems since April 1983. The Eden implementation is in two parts: the Eden Programming Language, and the Eden kernel. Eden coexists with UNIX, in the sense that an individual can make simultaneous use of UNIX and Eden services. This coexistence was crucial in minimising the software effort required to make Eden usable, and was the main motivation for our choice of prototyping environment.

The set of UNIX facilities that Eden uses is small: processes and address spaces, a minimal flat file system, and the ability to load code into those address spaces from a file. We attempted to minimise the changes made to the UNIX kernel. Starting with 4.1 bsd UNIX, we added an ethernet driver and an inter-process communication mechanism, and decreased the granularity of the timer; other changes were limited to system parameters.

Each Eden object is implemented as a UNIX process. The Eden kernel operates as an additional UNIX process on each node; this process is called an Eden host. Both the host and the object processes operate in user-mode, and do not require any special privileges. The rôle of the host is to create object processes, to maintain part of an object's state, to maintain caches of object locations, passive representations and code, and of course to implement the set of system calls that characterise Eden, including the calls that send an invocation, create an object, and checkpoint a passive representation.

In addition to the host, every node with a disk runs a second kernel process called a POD, for *Permanent Object Database*. The POD's function is to manage the passive representations of those objects that have checkpointed onto its disk. When a checkpoint occurs, the POD arranges that the old passive representation is replaced atomically by the new data; this may involve communicating with other PODs if the checksite has moved. The POD also manages the executable code that makes up an Edentype: this code is simply the passive representation of another object of type *Typestore*.

When one object wishes to invoke another, it calls the kernel-provided *AsynchInvoke* primitive, passing the capability of the target object and the appropriate data as parameters. If the target object happens to be located on the same node as the invoker, the kernel will discover this

by examining its tables, and will deliver the invocation message directly. To reply to the invocation, the target makes the *ReplyMsg* kernel call, and the kernel routes the reply message to the invoker. If the information in the kernel's location tables indicate that the target object is on another node, the kernel will send the invocation message to the kernel process on that node, which will deliver it to the target object. If there is no entry in the kernel's tables for the target object, the kernel engages in a multi-layered *location protocol*, which will eventually return with either the location of the active form of the target object, or with an indication that the object is not active but that its passive form is located on a particular POD. In the latter case, the object is automatically activated on an appropriate host, to which the POD makes the executable code and the passive representation available. Once the target object is actived, invocation proceeds as normal.

3. The Eden Programming Language

A major achievement of the Eden project has been the design and implementation of the Eden Programming Language. EPL is based on Concurrent Euclid, a Pascal extension providing processes, modules and monitors; it provides direct support for the fundamental abstractions of Eden, that is, capabilities and invocation. Capabilities are first-class citizens, even to the extent of having source-language denotations. Syntax exists both for sending and receiving invocations, making invocations as easy to use as conventional procedure calls. We feel that this has been a key factor contributing to the ease of use of Eden.

The significance of EPL is not in the way it extends the state of the art of language design, but in the careful matching between the concepts of Eden and the structures of EPL. The Eden invocation is a simple concept; the challenge was to make its realisation in EPL equally simple. On the invoking side, the programmer sees an ordinary procedure call with an additional *status* parameter (see figure 1). On the invoked side, the programmer writes a procedure body, again quite ordinary except that it is designated an *invocation procedure* and certain parameters must be present. In addition, some process on the invoked side must receive the invocation and call the invocation procedure. The declaration of the invocation procedure is a way of stating that the object is willing to handle a particular invocation; it also defines the parameter list for that invocation, and thus provides the information needed to perform type-checking.

The figures illustrate how invocation support is implemented. The invoking routine actually calls a stub procedure (in the rectangular box, figure 1), which has been generated by a program from a description of the invocation interface. The receipt of an invocation is shown in figure 2. The invocation is received by a user-written process (in the circle in figure 2), but typically the only action of this process is to call the automatically generated *CallInvocationProcedure* which unpacks the arguments, calls the appropriate procedure in the target object, and packages up and sends the results.

The other main contribution of EPL is the provision of intra-object concurrency. A run-time kernel provides multiple light-weight processes and monitors within the UNIX process that supports the object. Thus, when a client process makes a remote invocation, it is possible to suspend just that process pending the receipt of a response; other client processes are free to continue. It is therefore possible to write a library module that implements the abstraction of synchronous communication, using the *AsynchInvoke* kernel call and EPL processes and monitors. One such module, the *Dispatcher*, is the standard way of sending and receiving invocations in Eden; programmers prefer not to use the asynchronous primitives, even though they are available. The matter of synchronous *vs* asynchronous communication and the availability of concurrency is discussed in greater depth in reference 6.

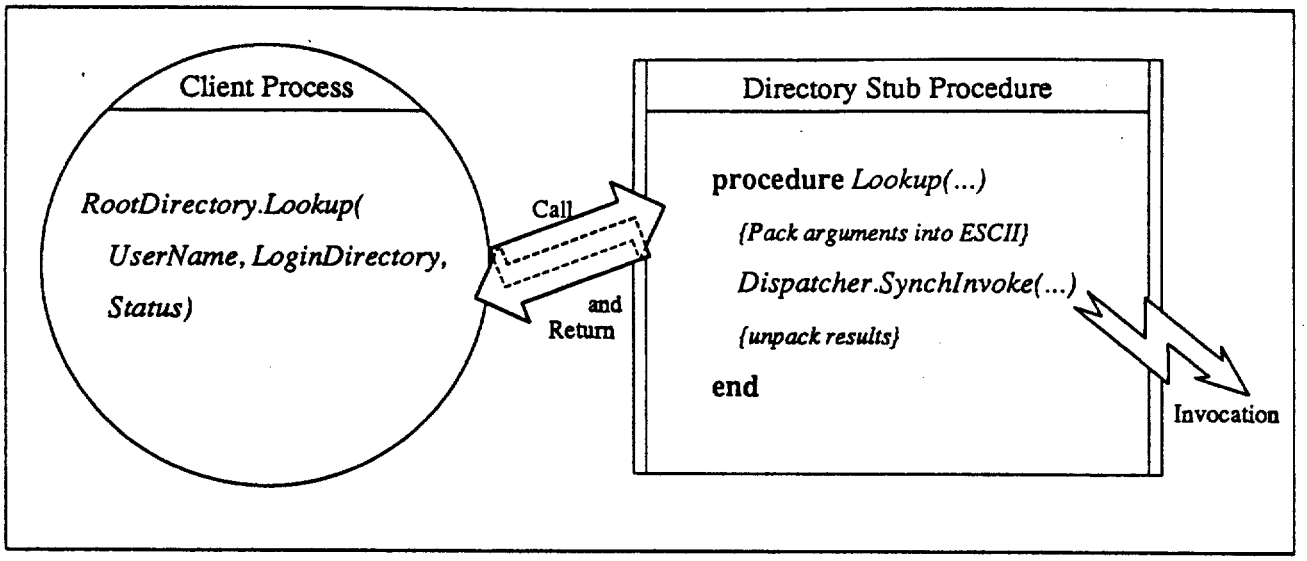

Figure 1: Sending a *Lookup* invocation

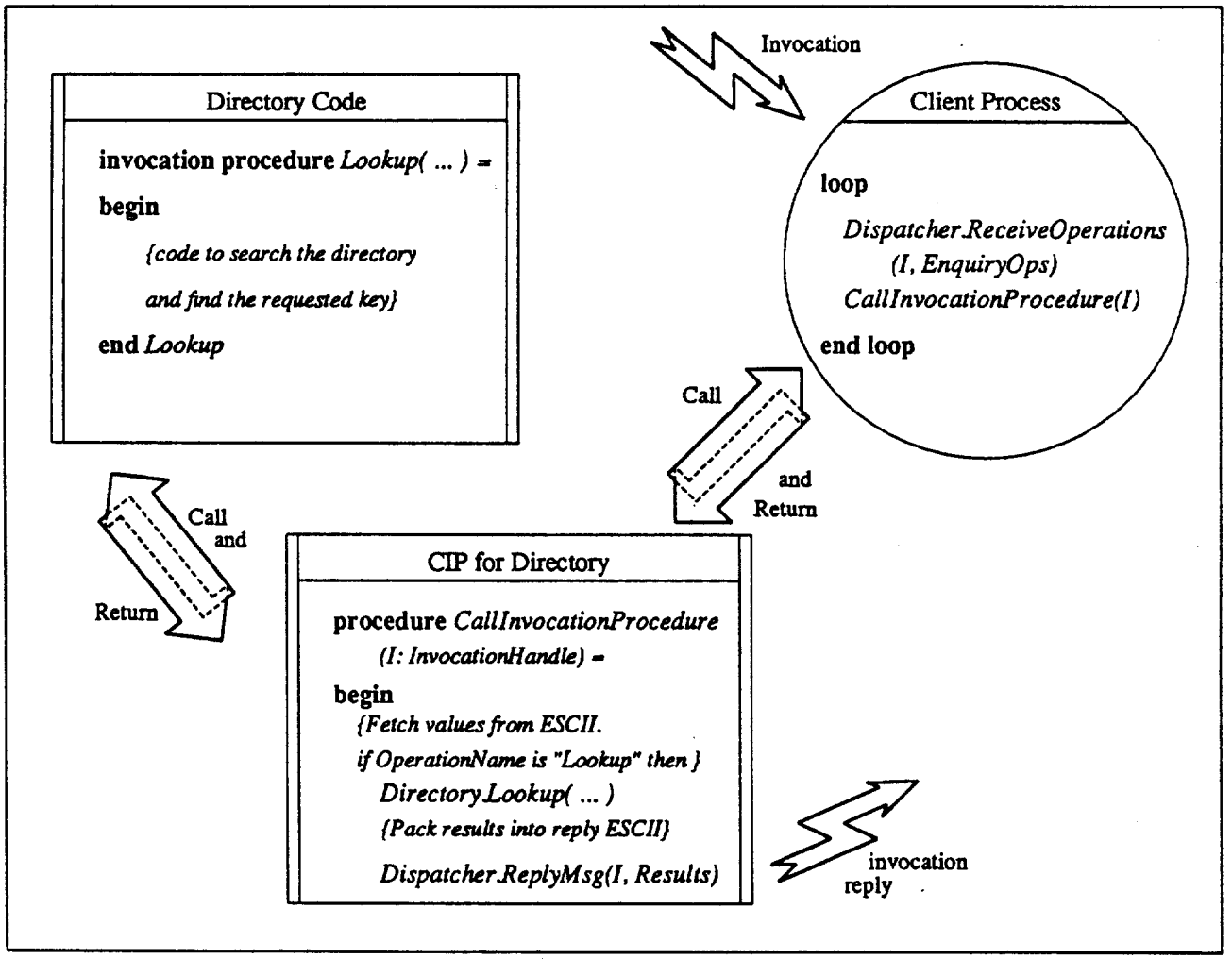

Figure 2: Receiving a *Lookup* invocation

4. Some Eden Applications

Each application that has been built in Eden has three objectives, although the relative emphasis on these objectives differs among the applications:

- *to evaluate the system*

  The ideas of Eden, i.e., the hospitability of the system for distributed applications, can only be assessed through use.

- *to make Eden a complete system rather than a kernel*

  In their evaluation of Cal-TSS, Lampson and Sturgis note that a kernel constitutes perhaps ten per cent of an operating system [11]. A key advantage of the object-based approach used by Eden is that many traditional operating system components can be built as applications.

- *to conduct research into the applications themselves*

  The Eden system provides a laboratory for exploring the design and structure of distributed applications.

To consider a few examples: the Eden Mail System (Edmas) [3], our first application, was written largely with the first objective in mind – testing the support that the Eden system provides for programming distributed applications. It was not the intention to interconnect Edmas with other mail systems to provide a general mail utility, nor was it the intention to study mail systems in general (although considerably more has been learned about this than was anticipated). The same is true of the basic file system (a subset of the system described in reference 10).

The reader may be surprised to see us refer to the file system as an application, but conventional sequential files are not part of the Eden kernel. A sequential file is simply an object that allows its contents to be read and written using the Eden transput (input/output) system. The *Checkpoint* facility allows such an object to maintain its state on the disk, and thus to exist permanently. However, sequential files are not the only form of long-term storage in Eden. All objects, once checkpointed, are equally permanent. A means is required for keeping capabilities for them; the Edentype *Directory* does just this. A Capability for any object can be associated with a mnemonic string and stored in a directory, and can later be retrieved by performing a *Lookup* operation on the directory with the string as argument. Since Directories can be entered into other directories, an arbitrary directed graph of Directories can be built up from a single root.

The Eden Calendar System [9] was written largely with the first and third objectives in mind: to further evaluate the Eden system's hospitability, and to explore the use of transactions in multi-user calendar systems. The Eden Terminal Handler and the Eden Command Language Interpreter were built primarily because they were necessary for the use of the system – the second objective. Finally, moving the translator for the Eden Programming Language into Eden contributed to all three objectives: the translator exercises virtually every aspect of the system, while exploiting the Eden architecture to solve specific practical problems.

Eden has been and continues to be used as a basis for research in distribution, replication, and concurrency control. An experimental version of Eden provides replicated passive representations, kept consistent in the face of failures by a voting scheme [15]. Crash-resistant resources have been implemented on top of the normal Eden system by using multiple objects to represent a single logical resource; to update the resource despite some of the copies being unavailable, we *regenerate* the inaccessible copies elsewhere in the network [16]. We have also implemented a general purpose nested transaction mechanism out of Eden objects; each transaction is characterised by its own transaction manager object that is responsible for the concurrency control and crash recovery of its sub-transactions [17].

Representation	
Start Time:	Monday, 22nd Sept 1986 at 13:30
End Time:	Monday, 22nd Sept 1986 at 15:00
Description:	EUUG Conference Session
Event Status:	Definitely Scheduled
Participants:	Black <capability> ; Donner <capability> ; Harper<capability>

Operations
AppendParticipant()
CancelEvent()
ConfirmEvent()
GetNextParticipant()
LookupParticipant()
RejectEvent()
ScheduleEvent()
SetDescription()
Show()

Figure 3: An Event Object

To give the flavour of the way applications are constructed in Eden, I will briefly review the design of the Eden Calendar System †. More details can be found in reference 9. The Calendar system is designed to serve both as a personal appointments book, in which one can schedule, list and cancel appointments, and as a shared calendar system that assists one to schedule events with one or more other users, while guaranteeing consistency of the calendars and detecting conflicts. I will first discuss some design considerations, and then the implementation in terms of objects. The way that the objects interact to achive atomicity will then be described; although this is atypical of Eden applications (most do not have very stringent atomicity requirements), it is interesting to see how transactions can be built that take advantage of the semantics of the particular operations available on the objects concerned.

A fundamental feature of such a system is that information about appointments needs to be shared by all of the participants. There are two obvious ways of doing this: centralising the information in one place, and letting each participant refer to it, or replicating the data so that each participant has a copy. Centralisation has the advantage that consistency is assured, but performance and availability may suffer. Replication of the data means that one has to take special care to ensure consistency; this also impacts performance, and may also adversely affect availability, depending on the techniques used. The compromise reached in the calendar system is partial replication. The system contains two basic types of Eden objects: *Events* and *Calendars*. Events are the repositories of the true information about an engagement; Calendars contain hints and caches to improve performance, and use a transaction mechanism to ensure

† Although technically appropriate, this choice of subject illustrates the difficulties of writing for a trans-atlantic audience. A calendar, to an American, is what the English call a diary: a list of personal engagements made for fixed dates and times. There seems to be no word that is satisfactory in both languages, so I will trouble my English audience by following the system's designers and using the word calendar.

Representation		
Owner's Name:	Andrew P. Black	
Last Read:	Monday, 8th September 1986 at 09:30	
Last Updated:	Monday, 8th September 1986 at 09:05	
Appointments:		
Start Time:	22 ix 1986 13:30	23 ix 1985 12:00
End Time:	22 ix 1986 15:00	23 ix 1985 13:00
Status:	definite	tentative
Owner Action:	accepted	no action
Capability:	<figure 3>	<lunch event>
History:		
Cancelled and past events		

Operations
AddEvent(), Remove()
Confirm()
FindFreeBlock()
GetLock(), FreeLock()
ListNext()
SetName()
SetStatus()
Show(), ShowEventTotals()

Figure 4: A Calendar Object

consistency where necessary.

Figures 3 and 4 illustrate typical Event and Calendar objects. They may be thought of as abstract machines that encapsulate the relevant data representation and provides access to it through a set of operations. The representation of the event is a record structure containing the relevant data; note that the list of participants is represented as a list of capabilities for their calendars. The given operations can be applied to the event to add participants, interrogate the data, and so on. The *ScheduleEvent* operation provides automatic scheduling; it takes as arguments an interval of time and a duration, and attempts to find a free slot on all of the participants' calendars of the required duration within the interval. The calendar object (figure 4) has a similar structure; the appointments field is represented as a list of events. As well as Capabilities for event objects, some of the information about each event is replicated in each participant's calendar.

Scheduling an engagement is a human-time operation that may take days or weeks: it cannot be completed until the last participant has examined his calendar and agreed to the tentatively scheduled meeting – and that person may be on holiday. Because most transaction techniques work well only when locks are held for brief time-spans, scheduling is broken into two parts. First the event is tentatively scheduled, which is done at machine speed; confirmations are then obtained from each participant.

The event object itself serves as the commit record for the scheduling transaction. The Event object first locks the calendars of all the participants, finds an appropriate time slot, and then enters itself on all of the calendars in that time slot. A two-phase commit is used to ensure that the event is entered on all of the calendars, or is aborted; the locks can then be released. Each calendar replicates the data concerning the time span covered by the event; this is to enable

searches for free times to proceed without the need to invoke every Event object. Since the timing information does not change, its consistency is not an issue. The calendar also keeps information about the status of the event. Initially, the event is tentative; when the calendar's owner agrees to the engagement, the status becomes confirmed. Again, a two-phase commit is used to ensure that the Event object and the confirming calendar agree. When the last participant agrees to the engagement, the Event object attempts to notify all of the participants that the appointment is now definitely scheduled. However, this information is relayed to the calendars on a best effort basis; transaction techniques are not used. This is done so that the last user is not prevented from confirming an engagement just because another participant's calendar is unavailable (perhaps because a machine is down). The confirmed status in a calendar is thus a hint that needs to be checked against the truth held in the Event object; it is possible that the event is in fact scheduled, but that the calendar object has not been informed. The process of scheduling an event is shown in figure 5.

5. Some Assessment

Eden can be assessed in two frames of reference. First, considered as an architecture for supporting distributed applications, one can take specific features of Eden and see how they contribute to that goal. A recent SOSP paper [6] attempts this task; in this forum I will merely mention some of the more significant findings. Secondly, one can assess the current implementation of Eden as an artifact in its own right: how good a job did we do in building it, and in what ways did our substrate system – UNIX – help us or hinder us.

Considered as an architecture, Eden provides good support for applications. This is of course the finding we hoped for, but it is certainly not just experimental bias. The fact that two students could be assigned the building of a distributed mail system as a six week project for the graduate operating systems course – and complete the task on a kernel system that was still being actively debugged – was surprising even to us. However, a large part of the credit must go not to the system design itself, but to the programming language support that was provided, almost as an afterthought. EPL's provision of syntactic support for invocation receipt and dispatch, and the combination of synchronous invocation and lightweight processes, seem to be tools that are accessible to the ordinary programmer. Those of us deeply involved in the mystique of the innards of Eden were at first alarmed when programmers who had constructed substantial applications asked questions that displayed what was to us an amazing ignorance of the way the system worked. But of course, that is exactly as it should be: the programming language itself should present a coherent model of computation, and programmers should not need to delve below that level.

One concept that has proved to be very important is the *Abstract Edentype*. While a concrete Edentype is a particular piece of code that defines the interface and behaviour of a real object, an Abstract Edentype is an abstraction of this: the specification of an interface and a behaviour. This specification may be satisfied by many concrete Edentypes or by none. A given concrete Edentype may implement several abstractions. The most obvious application of this idea is in device-independent transput. The abstraction of a readable *Stream* is implemented by several concrete Edentypes, in particular by sequential files and by the terminal handler. The implementation of the *transfer* and *close* operations is obviously very different in each case, but an invoking object need not be concerned with this, provided that data is available when *transfer* is called. Once recognised, Abstract Edentypes crop up in all sorts of applications; for example, the extended version of Edmas [2] uses an internal abstraction *MailSink*, as well as sharing an abstract Edentype with the file system. Three different concrete types in the transaction-manager tree [17] implement a lock manager interface; in this case, they also share the same implementation module.

Status in Event	Status in Harper's Calendar	Status in Black's Calendar
Tentatively Scheduled by Harper:		
aborted	pre-tentative	
		pre-tentative
tentative	tentative	
		tentative
Confirmed by Harper:		
	pre-confirmed	
confirmed (Harper)		
	confirmed	
Confirmed by Black:		
		pre-confirmed
confirmed (Black)		
		confirmed
When all participants have confirmed ...		
scheduled		
	scheduled	
		scheduled

Figure 5: Stages in Scheduling an Event

Eden allows the use of Abstract Edentypes, but does not offer any explicit support for them. Because capabilities are not typed, a client that claims to be reading from a *stream* object will have no problem reading from a sequential file, provided that the file supports the right interface. Explicit support for abstract types is one of the design goals of a new distributed object-oriented language currently being implemented [8] [7].

One of the less successful features of Eden is *Checkpoint*. In its favour is simplicity of concept and universality: any desired updating of the passive representation can be achieved by using multiple checkpoints. However, in practice this is of little use, because the cost of checkpointing is too high to make it a useful primitive. It may take as long as a second to perform a checkpoint operation – if the amount of data is small (less than a few kilobytes), this time is more or less independent of the size of the data. The reason is that most of the time is occupied by UNIX overhead in updating the disk atomically. This 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 seventy five per cent of the CPU time used by an object in checkpointing a kilobyte is consumed by *link*, *unlink*, *open*, *creat*, and *access*; *write* uses seven per cent of the CPU time. Similarly, over seventy per cent of the time spent by the kernel process at the checksite is consumed by *link*, *unlink*, and *open*.

One way of avoiding the inefficiencies of the UNIX file system would have been to use the

raw disk interface instead. The problem with this is that we would no longer be able to use *exec* to load the code of an object into a new address space. Instead we would have to write our own loader, and execute out of data space. This in turn would have prevented us from sharing code between objects of the same Edentype that happen to be on the same machine. Since object code is large (usually over 150 kbytes, much of it in the form of common libraries) this would have significantly increased swapping and paging.

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. In fact, the paging hardware is equipped with dirty bits that could do an efficient job of recording exactly which pages have been changed since the last checkpoint – but of course the UNIX abstraction of address space does not allow one to access that level of the implementation.

Another place where UNIX hides power is discussed at length elsewhere [6]: the Berkeley 4.2 inter-process communication primitives, which omit to report if an IPC message has been dropped, even in the local case. But it is easy to criticise UNIX, and to forget its benefits. Apart from the availability of source code and the relative ease with which it could be modified, the chief reason for our choice of UNIX as a prototyping environment was that it provided a path whereby users could be migrated gradually onto 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 [3]. Similarly, we have an interface that allows one to use Emacs to edit Eden files as easily as UNIX files.

Another advantage of building Eden on top of UNIX is that the two systems can coexist. Eden is currently running on a sixteen Sun workstations, including the one on which I am composing and formatting this paper. When Eden activity is low, Eden does not intrude on the UNIX user, yet the constant availability of Eden makes it a more suitable laboratory than if the workstation cluster had to be rebooted with the Eden system for each experimental use.

In the final analysis, I think that we made an appropriate choice in picking UNIX as a prototyping environment. The main cost is performance, and Eden has been criticised on these grounds. In fact, performance of the invocation has improved substantially since our first messages were exchanged, and now approaches the limit of what one can expect from a system that requires four cross-address space calls for each invocation and reply. It is adequate for a wide range of experiments. Checkpointing and activation are limited by the speed and structure of the file system, and are more of a bottle-neck in some applications. I believe that further significant performance gains can come only from a major reimplementation, in which the use of UNIX processes is severely restricted. As an illustration, the current prototype of Emerald implements all the objects on a given node inside one UNIX address space. Local invocations can therefore be made without incurring the cost of UNIX context switches. Only when accessing the network is it necessary to call the UNIX kernel. Nevertheless, the Emerald workstations can still run UNIX, which is an aid to debugging and provides a suitable environment for the Emerald compiler.

6. Summary

Eden is an implementation of an advanced object-oriented distributed programming environment. It is supported by its own programming language, which provided an early implementation of what has come to be known as Remote Procedure Call † [5], including full stub generation for both the invoker and invokee. Over three hundred thousand lines of EPL code have been written, comprising about a hundred different Edentypes. Significant experimental research projects in transactions, concurrency control, replication and loadsharing, as well as many more minor studies, have been carried out on top of Eden; none of this would have been possible without the existence of the Eden system.

Eden has been criticised on the basis of poor performance. It would be naive to pretend that performance is not sometimes a problem. However, for many applications and experiments, Eden's performance is entirely adequate. Poor performance is part of the price that one pays when using one operating system as a substrate for another: nevertheless, the benefits of using UNIX were, I believe, worth the costs in our case. Of course, this is not to say that it would not be possible to get better performance even on top of UNIX if we had made some of our implementation decisions differently. Like most experimenters, we made some mistakes, some of which became apparent later. In continuing to use UNIX as a prototyping environment for other projects, we are attempting to learn from those mistakes.

Bibliography

[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. and Holman, C. Edmas: An Object-Oriented, Locally Distributed Mail System. Tech. Rep. 84-08-03, University of Washington, Computer Science Dept, August 1984.

[3] 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.

[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. Supporting Distributed Applications: Experience with Eden. *Proc. 10th ACM Symp. on Operating System Prin.*, December 1985, pp181-193.

[7] Black, A. P., Hutchinson, N., Jul, E. and Levy, H. M. Object Structure in the Emerald System. *Proc. First Conf. on Object-Oriented Programming Systems, Languages and Applications*, October 1986.

[8] Black, A. P., Hutchinson, N., Jul, E., Levy, H. M. and Carter, L. Distribution and Abstract Types in Emerald. *To Appear, IEEE Trans. on Software Eng. SE-12, Nr 12* (December 1986).

† In the context of Eden the name is particularly unfortunate: it is fundamental to the system that the invoker does not need to know whether the target is remote, and that the called entity is an object rather than a procedure.

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

[10] Jessop, W. H., Jacobson, D. M., Noe, J. D., Baer, J. and Pu, C. The Eden transaction based file system. *Proc. 2nd Symp. Reliability in Distributed Software and Database Systems*, July 1982, pp163-169.

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

[12] 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.

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

[14] Liskov, B. Overview of the Argus Language and System. Programming Methodology Group Memo 40, M.I.T., Laboratory for Computer Science, February 1984.

[15] Proudfoot, A. B. Replects: data replication in the Eden System. M.S. Thesis, University of Washington, Computer Science Dept, October 1985.

[16] Pu, C., Noe, J. D. and Proudfoot, A. Regeneration of Replicated Objects: A Technique and Its Eden Implementation. *Proc. Second International Conference on Data Engineering*, Los Angeles, CA, February 1986, pp175-187.

[17] Pu, C. Replication and Nested Transactions in the Eden Distributed System. Ph.D. Thesis, University of Washington, Computer Science Dept, July 1986.

[18] 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.

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