The Architecture of the Eden System

Edward D. Lazowska, Henry M. Levy, Guy T. Almes,
Michael J. Fischer, Robert J. Fowler and Stephen C. Vestal

Department of Computer Science
University of Washington
Seattle, Wa. 98195

Abstract

The University of Washington's Eden project is a five-year research effort to design, build and use an "integrated distributed" computing environment. The underlying philosophy of Eden involves a fresh approach to the tension between these two adjectives. In briefest form, Eden attempts to support both good personal computing and good multi-user integration by combining a node machine / local network hardware base with a software environment that encourages a high degree of sharing and cooperation among its users.

The hardware architecture of Eden involves an Ethernet local area network interconnecting a number of node machines with bit-map displays, based upon the Intel iAPX 432 processor. The software architecture is object-based, allowing each user access to the information and resources of the entire system through a simple interface.

This paper states the philosophy and goals of Eden, describes the programming methodology that we have chosen to support, and discusses the hardware and kernel architecture of the system.

1. Introduction

The University of Washington's Eden project is a five-year research effort to design, build and use an "integrated distributed" computing environment. This phrase captures much of the underlying philosophy of Eden.

Eden began from the observation that contemporary multi-user computing systems represent two extremes of a spectrum:

- Good centralized systems (e.g., UNIX and TOPS-20) provide nicely integrated multi-user environments, allowing several users to cooperate in solving a problem by sharing information and resources. Due largely to hardware limitations, though, such systems provide relatively poor support for personal computing (they suffer from erratic response, low display bandwidth, limited total capacity, etc.).

- Good distributed systems (e.g., the Xerox Mesa environment running on an Alto/D-machine / Ethernet hardware base) provide nice personal computing environments. Due largely to the objectives of their software designers, though, such systems provide relatively poor facilities for multi-user and multi-computer integration.

The foundation of the Eden project lies in a fresh approach to this tension between integration, which has

Eden is supported in part by the National Science Foundation under Grant No. MCS-8004111. Computing equipment and technical support for Eden are provided in part under cooperative research agreements with Digital Equipment Corporation and with Intel Corporation.

repeatedly demonstrated its value in communities of cooperative users, and distribution, which is the best approach to achieving the good support for personal computing that the Xerox experience has proven to be so valuable. In briefest form, the Eden project combines a node machine / local network hardware base with a software environment that encourages a high degree of sharing and cooperation among its users.

It is not accurate, though, to describe Eden as "a distributed time-sharing system." One of our principal goals is the specification and support of a programming methodology that is especially well suited both to our hardware base (we are designing and building node machines with bit-map displays, based upon the Intel iAPX 432 processor, and interconnecting them with an Ethernet local area network) and to our intended user community (computer scientists studying computer system design and building advanced distributed applications programs). We have chosen an object-based programming methodology for Eden, selecting and extending concepts introduced by Multics [Daley & Dennis 1968], Hydra [Wulf et al. 1974, 1975, 1981; Cohen & Jefferson 1975], CLU [Liskov et al. 1977; Liskov & Snyder 1979], Smalltalk [Ingalls 1978; Goldberg et al. 1981], Medusa [Ousterhout et al. 1980], StarOS [Jones et al. 1979] and others. While the description of Eden objects is an important goal of this paper, we will not attempt to justify strongly our basic choice of methodology. We believe that many apparently reasonable programming methodologies exist for distributed systems, and that experimentation with these methodologies is the best way to assess their strengths. Thus our objective in the Eden project is to thoroughly explore one apparently reasonable methodology. Among the alternatives to our own approach are the Mesa environment [Lampson & Redell 1980], generalizations of UNIX [Rashid 1980], TRIX [Ward 1980], the Apollo "domain" concept [Apollo 1981], and Extended CLU [Liskov 1979, 1980].

Our initial thoughts on these matters were developed during 1979 and were made part of a research proposal submitted to the National Science Foundation that October [Eden 1980]. In September 1980, the Eden project received NSF support as the first award in the new Coordinated Experimental Research Program. In this paper, written eight months after the official start of the project, we would like first to elaborate on the goals of Eden and on the programming methodology that we have chosen to support, and then to describe the hardware and kernel architecture of the Eden system.

2. Goals and Approaches

The Eden project as a whole has two fundamental goals: to design and build an integrated distributed computing system consistent with the philosophy stated in the previous section, and to provide support through this system for a community of computer scientists studying computer system design and building advanced distributed applications. Equally important to understanding Eden is an appreciation of the project's explicit non-goals:

- We are not concerned with supporting a network of heterogeneous nodes, and with the concomitant problems of communicating abstract values among such nodes. Eden will support one basic node type (variations in configuration will of course be allowed), and "foreign" machines will be interfaced to the system through such nodes. Eden users can invoke services on foreign machines through an "object-like" interface, but the relationship will not be symmetric. Machine independence is an explicit goal in the design of the Eden kernel. This task, although challenging, is considerably less difficult than supporting heterogeneous nodes.

- We are not concerned with extreme resistance to maliciousness. This is not to say that we are ignoring protection issues; indeed, a flexible protection structure is one of the virtues of a capability-addressed, object-based system. But Eden is intended to be used by a community of computer scientists, not by a bank.

- Similarly, we are not concerned with extreme reliability. On the one hand, the Eden kernel is being designed to be tolerant of failures in its components. On the other hand, the kernel does not automatically guarantee the extreme reliability of systems and applications built upon it.

All of the above problems are challenging and important. We simply have found it necessary to limit our objectives carefully.

We have stated that we have chosen to support an "object-based" programming methodology in Eden. This phrase has a certain fashionable ambiguity about it, and before proceeding with our description of the architecture of the Eden system it is important that we be more precise.

Many recent advances in Computer Science have been based on recognition of the importance of abstraction in the design of complex systems, and on the concrete realization of this view in programming systems. The idea is to decompose the design of a complex system into components and, for each component, to separate those details that are essential for its use (the specification) from those details that are only of concern to the implementor of the component (the algorithms and data structures). The word "object" has been used to describe a particular class of mechanisms that enforce this methodology. In an object, the data structures for an instance of a type are bound to the code sequences that implement the operations on

that type. The only way the user of an object can manipulate its data structures is by invoking code sequences defined for that type. Protection against external manipulation of an object's data structures is provided, and the details of representation and manipulation are concealed from the user.

Eden objects are consistent with this model. An Eden object may be viewed as an instance of an abstract data type. Each Eden object has a unique name, a representation (a data part), a type (a collection of procedures defining the operations on the object, shared among objects of the same type), and some number of invocations (threads of control). Eden objects refer to one another by means of capabilities, which contain both unique names and access rights. The representation of one object can be examined or manipulated by another object only by invoking operations defined by the first object's type. Eden objects are active entities, in the sense that each object supplies processes to execute operations in response to invocation messages originated by other objects.

In Eden, though, we go beyond these concepts by asserting that the details of location, concurrency, and error recovery should be encapsulated within an object, as are the more traditional aspects of implementation. We believe that the specification of an object's performance or reliability should be distinguished from how such performance or reliability is achieved. Details such as the location of an object within the network, the degree of concurrency within an object, and the mechanisms used for error or crash recovery are, like data structures and algorithms, of interest only to the implementor of an object (strictly speaking, the implementor of the type of which the object is an instance), who is responsible for meeting the external specifications regarding functionality, performance, and reliability.

The user's view of objects in Eden, then, is extremely simple. Possession of a capability for an object implies the ability to manipulate that object's representation by invoking some subset of the operations defined for objects of that type. The invocation proceeds in a manner not unlike a traditional procedure call: parameters are passed and the caller's thread of control is suspended pending completion of the invocation. (There is, of course, no shared memory.) Eden implements a location-independent address space of objects: it is the responsibility of the Eden kernel, when called upon to perform an invocation, to determine the node on which the target object resides and to forward the invocation message to that object. (It is for this reason that we use the phrase "node machine" rather than "personal computer" in describing Eden. An Eden node machine is expected to provide good personal computing support for the user whose office it heats, but is also part of a larger system and cannot be regarded as fully autonomous.) The user of Eden objects also perceives a single-level memory with no concept of backing storage: all objects are ready to receive invocations at all times.

The semantics within an Eden object, those of concern to the programmer who defines a particular type, are considerably more complex. The Eden type programmer must be concerned with manipulating the representation of an object from within, with inducing and controlling concurrency, with altering the location of an object within the system, with using backing storage to increase the reliability of an object, etc. These will be discussed in Section 4 of this paper.

The distinction between the user of Eden objects and the Eden type programmer is in one sense a false one: every Eden programmer is both a user of Eden objects and an Eden type programmer, for programming in Eden consists of defining types that invoke operations on objects of other types. In another sense, though, this distinction is the essence of Eden's programming methodology: Eden strives to facilitate the construction of experimental computer systems and of advanced distributed applications by simplifying the semantics of sharing.

3. The Hardware Architecture of the Eden System

In most respects Eden's hardware architecture strongly resembles that of existing networks of personal computers. Each user in Eden has a node machine consisting of one or more reasonably powerful processors, one or two million bytes of memory, a keyboard, a pointing device such as a mouse or trackball, a bit-map display, and mass storage. These node machines are interconnected by a local area network. Eden node machines are homogeneous, although minor variations in configuration are of course allowed. (Examples include the amount of primary memory, the amount of secondary memory, the number of processors, and the type of display on any particular node.) Special-purpose servers such as conventional time-sharing computers, high-resolution hard-copy output devices, gateways, and file servers, are interfaced to the system through node machines. This highest level hardware architecture is shown in Figure 1.

Our criteria for selecting node machines and communications were similar to those enunciated in Carnegie-Mellon University's Spice solicitation [Newell et al. 1980]. Although Eden was conceived as a software research effort, we have chosen to construct our own node machines. We have based our design upon the Intel iAPX 432 processor.

The system-level architecture of the 432 is shown in Figure 2, in the configuration of the default Eden node machine. The 432 has a modular structure, consisting of several interconnected computer systems: a central system responsible for program execution, interfaced by special-purpose processors to a number of

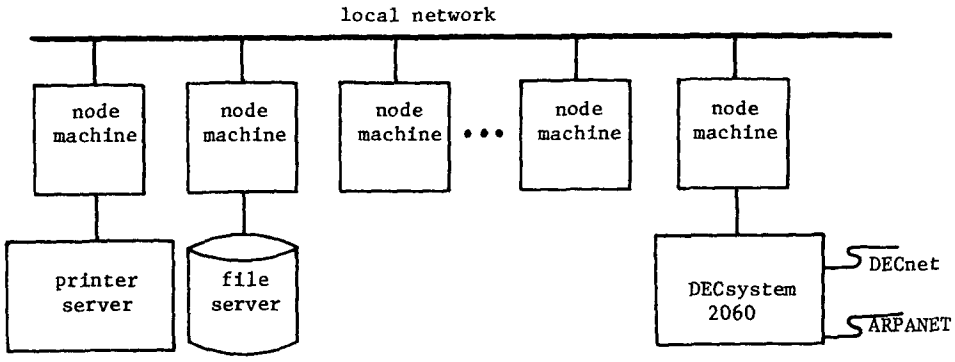

Figure 1. Eden system-level hardware architecture

satellite systems responsible for peripheral device control.

The central system consists of a packet-based interconnect bus which can accomodate multiple CPUs, called General Data Processors (GDPs), a significant amount of memory, and multiple Interface Processors (IPs). Each IP is responsible for communication between the central system and a specific satellite system. The default Eden configuration includes two GDPs, 1M bytes of memory, and two IPs. A node with this configuration can be "field upgraded" to four GDPs and 2.5M bytes of memory.

The two satellite systems each have a Multibus backplane with 128K bytes of global memory, an Intel 8086/8087 co-processor pair with 128K bytes of local memory, a DMA controller, and various device interfaces (one of which is the interface to the IP). Our choice of two satellite systems and our decisions regarding assignment of peripherals to these systems (this assignment is, of course, flexible) were based largely on Multibus bandwidth considerations.

Because of the widespread use of the Multibus standard (IEEE 796), our node machine construction effort consists largely of integrating available Multibus-compatible system components. This ability to select components from a large pool of vendors has provided us with considerable flexibility in configuring Eden node machines. On the other hand, we have encountered considerable difficulty in obtaining advanced devices, such as a bit-map display of suitable resolution and bandwidth.

From a purely technical point of view, the choice of the iAPX 432 has both advantages and drawbacks. Some of these are listed below:

- The internal architecture of the processor supports segmentation and capability addressing. This allows system designers considerable freedom to design address spaces that are suited to the abstractions they are trying to support, and to construct and dismantle these address spaces dynamically. This contrasts sharply with flat address spaces, such as that of the PDP-11 used in the Hydra project.

- The processor architecture also supports port-based inter-process communication and the short-term scheduling of processes. When combined with the ability to easily construct new address spaces, this encourages system designers to make liberal use of processes in structuring software.

- One of the advantages of the modular system-level architecture of the 432 is that additional GDPs and additional IP / satellite system pairs can be added easily should additional processing power be required.

- A drawback of this hierarchical structure, however, is that extremely close coupling between the processor and its peripherals (in particular, the display) is impossible. We hope to compensate for this by aggressive use of memory and processing power in the Multibus world.

- At this point in time the performance of the GDP is something of a question mark, especially in those portions of the instruction set dealing with invocation and address space creation. We expect architectural changes and additional on-board caching in future implementations to ameliorate these problems.

- User microprogramming, valuable for performance enhancement, is not possible.

For a summary of the iAPX 432 processor architecture, see [Intel 1981].

Having selected Intel-based node machines, the Ethernet jointly specified by Digital, Intel and Xerox [Metcalfe & Boggs 1976, Ethernet 1980] was the logical choice for our local area network; we had already satisfied ourselves of the suitability of Experimental Ethernet for our requirements [Almes & Lazowska

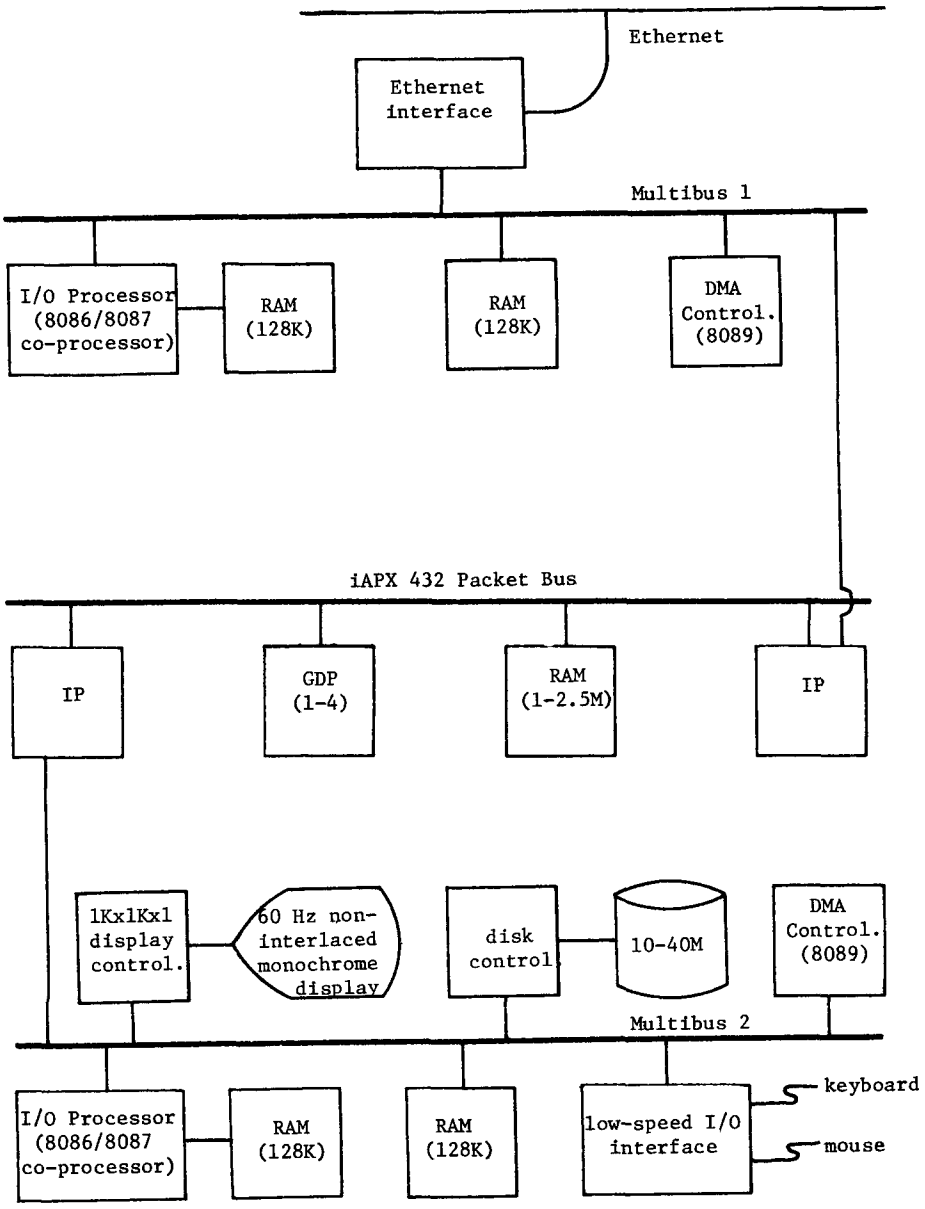

Figure 2. Eden node machine system-level architecture

1979]. The only other peripheral of special interest is the bit-map display. We have adapted the 1Kx1K monochrome display designed for the SUN Workstation at Stanford [Baskett et al. 1980]. We intend eventually to add a color display.

By late 1981 we expect to have five fully-configured prototype node machines in operation, one of which will be configured with a 300 megabyte disk to act as a file server. The five nodes will be interconnected by an Ethernet. During 1982 we expect to build an additional fifteen node machines.

4. An Overview of the Eden Kernel

The integrated environment seen by the Eden user is created by a collection of software systems operating on the distributed set of node machines. As in Hydra, our goal is to allow as much system software as possible to be constructed outside the kernel, that is, as user-level software. This is especially important because we hope that experimental systems work will continue long after the basic Eden system is usable.

The Eden kernel simply provides the set of primitives needed to support the object programming base of the system; for example, object and type manager creation and object addressing and invocation. Although there is no hierarchical structure to the systems outside the kernel (except that defined by the objects themselves through the graph structures connecting them) we can envision several logical levels of support for the programming environment. Some of these are shown in Figure 3.

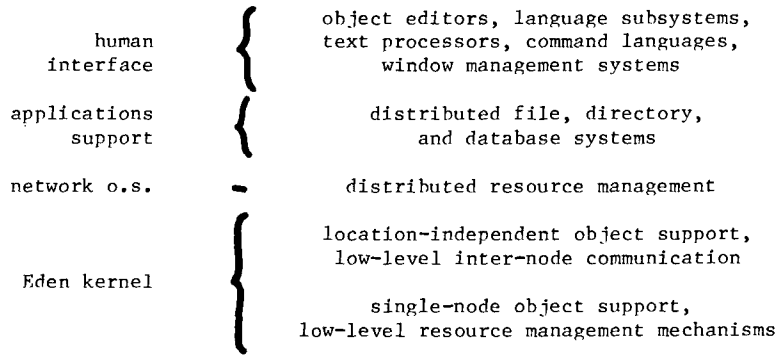

Figure 3. Eden software structure

This structure corresponds closely to structures found on typical time-sharing systems; it differs only in how the software is constructed, and in how distribution is transparently provided to the user.

When we refer to the kernel in this paper, we mean the software interface supplying location-independent object support. The kernel is below the level of the typical operating system interface, and is designed to support higher level operating system software. As an example, we believe that the Eden kernel would provide an extremely natural base for implementing the Extended CLU programming system.

Within the kernel will be several layers, some constructed themselves as Eden objects. For example, there will be code supporting a single-node object space and basic resource management mechanisms, surrounded by objects providing distribution facilities (e.g., the ability to invoke objects in a location-independent manner). All user programs, as well as traditional system software such as filing, directory, record management, and database systems, will be built using only the kernel-supplied object primitives.

In the following sections, we examine more closely the Eden notion of object that is used to construct the software systems comprising the Eden programming environment. We then describe an implementation model that allows the subsystem designer to exploit the distributed and multiprocessor characteristics of the hardware structure described in the previous section.

4.1 Eden Objects

Eden objects have two personalities that are reflected by the two principal goals of their design: simplicity and flexibility. When viewed from the outside, an object has a simple, consistent interface which supports a well-defined data or procedural abstraction. The only allowable form of interaction with an object is the invocation of an operation defined by the object's designer. Only a user possessing a capability with appropriate rights can request such a service from an object. The object verifies the user's rights, performs the service, and returns any status and output parameters to the user.

When viewed from the inside, however, an object may have more sophistication and complexity. The designer of the object (that is, of the supporting type) will wish to achieve desired goals of reliability, performance, and fault tolerance. The basic object structure must allow the designer the flexibility to exploit locality and concurrency in order to meet design goals. Of course, these details of the implementation are hidden from the outside user, in the same way that details of internal data structures and algorithms are hidden. Indeed, many type programmers in Eden will not be concerned with these details, because language subsystems will provide standard object templates.

Conceptually, an Eden object is composed of four parts, as shown in Figure 4.

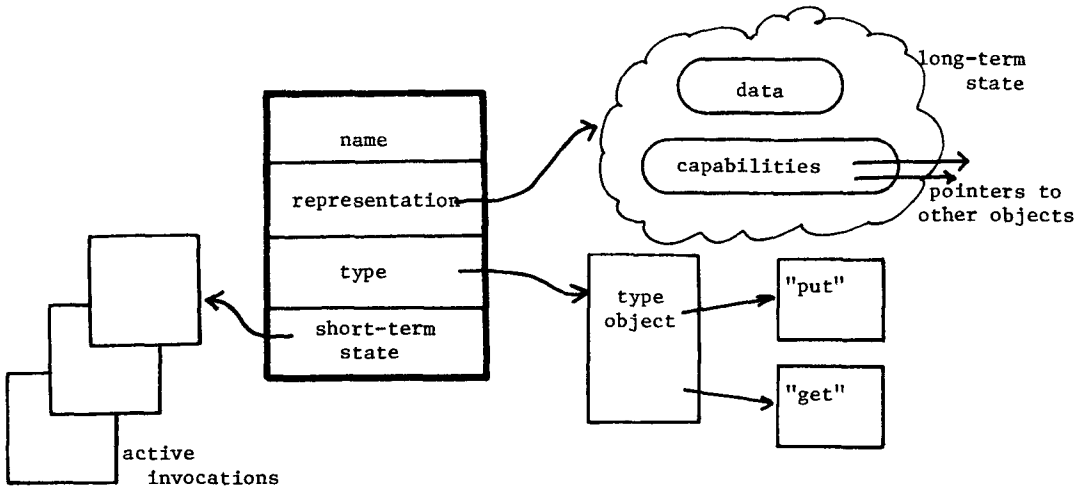

Figure 4. An Eden Object

- The name is a system-wide, unique-for-all-time binary identifier for the object; the name is location-independent, although it may indicate where the object was created.

- The representation consists of the data and capability segments that form the object's long-term state; these segments contain the data structures that implement any data abstraction. The long-term state is the only part of the object that is maintained and updated on long-term storage (although all objects do not necessarily occupy long-term storage).

- An object's type describes the set of routines that maintain the abstraction of which this object is a single instance. The type field contains a capability for another object in the system, a type manager, whose representation consists of instruction segments that define the operations allowable on object instances. On a single node, the type code can be shared by several instances of the type.

- Finally, the short-term state includes any temporal data, synchronization information, and processor state necessary to maintain one or more executing invocations. The short-term state is unique to each object instance, and is never written to long-term storage.

Eden objects form the basis of all programming in Eden, and are the fundamental units of distribution. Objects are addressed in a location-independent manner, and are the smallest entities that can be moved by the kernel from one node to another. All "traditional" programs, as well as physical and logical resources, are represented as objects. Eden objects have an active existence, in the sense that they are supported by active processes; there are no pure-data objects. Any data sharing must occur through the invocation mechanism, which we will now describe.

4.2 Invocation

The invocation operation, supplied by the kernel, is the major user-kernel interface. To invoke an operation on an object, the user supplies a capability for the object, the name of the operation to be invoked, and optionally a list of data and/or capability parameters, for example:

```
Invoke ( filecapa , "put" , "this is a new line" ) Returns ( status )
```

The invocation request may also contain a user-supplied timeout, specifying that the invoker wishes to be notified if the invocation is not completed within some time limit.

From the user's point of view invocation is a simple, synchronous operation much like a procedure call. The kernel blocks the caller's execution, builds the invocation message from the invocation request, locates the specified object, and sends the message to the object. At a later time, the object executes the request and responds with status and return parameters, which are packaged into a message and transmitted by the object's local kernel. When the response message is received, the invoker continues execution. (Asynchronous invocation also will be possible, either through a separate kernel primitive or through the ability to create subprocesses within objects.)

From within, the target object must execute and respond to the invocation request. Although Eden supplies a one-level memory system from the point of view of the invoker, objects actually exist in two possible states: active and passive. An active object has some processes executing within the virtual memory of a node. We first describe a model for active objects.

An active object can be viewed as a tree structure of processes. At the root of the object's process tree is a coordinator process. The coordinator consists of kernel code responsible for maintenance of the object, reception of invocation requests and responses, verification of rights, and dispatching of processes to invocations. An incoming invocation is thus routed to a port owned by the object's coordinator. The coordinator validates the request, and examines the state of the object to determine how the invocation will be dispatched. In the normal case, a new process will be created and assigned the invocation. This new process may also create other subordinate processes to aid in its execution. On a node with multiprocessing capability, these processes could execute concurrently.

Although multiple processes are allowed within an object, some controls are needed on the allocation of process resources. In creating a new type, the programmer divides the invocations into an exhaustive and mutually exclusive set of invocation classes, and specifies the number of concurrent processes that are allowed to be servicing each class. This limits the consumption of resources within an object and, in this sense, is an internal flow-control mechanism. Note that by limiting a class to one process, mutual exclusion is obtained among operations of that class. This is not the primary purpose for classes, however, and for fine-grained synchronization control, programmers can use kernel-supplied semaphore and message port primitives.

In contrast to active objects, passive objects exist on long-term storage and have no active processes (we will see in a later section how an object becomes passive). A passive object becomes active when an invocation request is received. When a passive object is "reincarnated" into an active one, the kernel creates a new coordinator process for the object. The coordinator will block the invocation while it attempts to execute the object's reincarnation condition handler. The reincarnation condition handler does any work needed to reinitialize the object, build temporary data structures, and so on. When the handler exits, the coordinator dispatches the invocation.

As part of its initialization tasks, the reincarnation condition handler may wish to spawn one or more detached processes to execute concurrently with invocation processing. Such processes, called behaviors in Eden, operate independently of invocations, except that they may exchange signals or data through any of the intra-object communication mechanisms. Behaviors can be used to perform object caretaking, for example, tree balancing or internal garbage collection. Note that a simple, single-thread traditional program might be implemented as an object with a single behavior and no invocable operations.

4.3 Object Location

Once again the tension between integration and distribution is seen in the notion of location. For the user of objects, Eden provides a location-independent address space: an invocation can be issued without knowledge of the target object's location. From within an object, however, location may be critical to achieving performance or reliability goals. Objects may require either co-location or distribution.

The notion of location in Eden is captured in the abstraction called a node. A node is an object that supplies virtual memory to store the segments of active objects and virtual processors to execute invocations. Although an abstract node corresponds roughly to a node machine, one physical machine may support several node objects, or several machines (i.e., a multiprocessor) may support one node object. At any point in time each active Eden object is supported by exactly one node. This node is responsible for supplying hardware resources and for receiving and processing invocations for the object.

Objects are capable of gaining location information from the kernel and making location changes. An active Eden object can request that responsibility for its resources be transferred to another node through the kernel-supplied move operation. In addition, some objects may have the ability to make location decisions for other objects in the system; for example, there may be a policy object responsible for the location of objects in a particular subsystem. To aid in locality and performance, some objects can be frozen. When an object is frozen its representation is made immutable, although it can still receive invocations. Such an object can be replicated and cached at several sites in order to save the overhead of remote invocations. Many traditional operating system utilities, such as compilers, will have this property.

4.4 Reliability

In Eden, we do not attempt to provide a system that is totally reliable, although we will make some guarantees about the correctness or atomicity of certain operations. What Eden does allow, however, is a simple method for the type programmer to protect objects against physical failures that would otherwise result in a complete loss of object state.

We have already commented on the existence of active and passive object states, and how a passive object becomes reincarnated when an invocation is received. Reincarnation is the basic method for object restoration following a failure. This requires that the type programmer take a two-level view from within the object, whereas the invoker sees a single-level world.

Active objects exist in the "virtual memory" supplied by the abstraction that we call a node. The virtual memory is supplied by physical memory along with possibly some swapping storage. Newly created objects exist only in this virtual memory, which is volatile with respect to failures, crashes, and the like. Eden makes no attempt to restore any state that existed in memory at the time of a crash or failure.

However, an object can request that the kernel record its long-term state (representation) on a reliable storage medium through invocation of the kernel checkpoint primitive. The type programmer must ensure that the object's representation is in a consistent state at the time the checkpoint is requested; that is, that the object can be reincarnated safely from that point. Following a node failure, if an invocation is received, the object will be reincarnated from the state that existed at the time the most recent checkpoint was executed. In addition to checkpointing, an object may specify, through the checksite primitive, which node is responsible for maintaining its long-term storage, and what level of reliability is required. Different reliability levels may cause different actions when a checkpoint is issued. Note that the checksite node that is responsible for maintaining an object's long-term state need not be the node responsible for supporting its active execution.

Finally, an object can crash itself. The crash simulates a virtual memory failure, destroying all exisiting active state. Following a crash, if an object has checkpointed itself, the object becomes passive and awaits the next invocation, when reincarnation will occur. An object may use crash to recover from its own internal failures, or as a form of exit operation to release system virtual memory resources.

4.5 Synopsis

The Eden environment is provided by a number of software layers, all of which are supported by the kernel. The kernel supplies a small set of relatively primitive operations, including:

- creation of new types and objects

- location-independent object invocation

- preservation of object long-term state over failures, and

- intra-object communication and synchronization.

Other components of the system are constructed on top of the kernel. These components rely only on the object mechanisms supplied by the kernel. At this point, our goal is to maintain the minimal size of the kernel. Later, additional functions can be moved into the kernel if measurements indicate that significant performance gains will result.

Although the Eden node machine is based upon the Intel iAPX 432 processor, every effort is being made to preserve machine independence in the design of the Eden kernel. The kernel is being implemented in Ada, which we expect to be widely available.

5. Summary

This paper has described the philosophy, goals, and architecture of the Eden system.

The Eden project is attempting to experimentally validate the hypothesis that the benefits of integration and of distribution can be successfully combined by using an object-based software environment on top of a node machine / local network hardware base. This approach serves to distinguish Eden from most distributed systems, which employ more traditional programming methodologies, and from most object-based systems, which rely on shared memory. The primary technical innovations of Eden arise in the definition of an object-based programming methodology that is well suited to a physically distributed environment.

The hardware architecture of Eden strongly resembles that of existing networks of personal computers: an Ethernet local area network interconnecting a number of node machines with bit-map displays, based upon the Intel iAPX 432 processor.

The software architecture of Eden is object-based. An Eden object may be viewed as an instance of an abstract data type, possessing a unique name, a representation, a type, and some number of invocations. The Eden user sees a location-independent address space of objects. Further, the user sees a single-level

memory with no concept of backing store: all objects are ready to receive invocations at all times. The semantics of object invocation are similar to those of a traditional procedure call. The Eden type programmer, on the other hand, implements abstract operations using facilities such as concurrency, locality, and backing storage.

We are in the first year of a five year research effort. Although the design we have described presently exists primarily on paper, by late 1981 we hope to have parts of the kernel running on a network of five prototype node machines. Work on higher-level software is proceeding in parallel, but was not described in this paper. Interesting aspects of this work include:

- An object editor. This research is attempting to provide a user environment in which all objects (such as directories, source programs, queues, etc.) have a syntactically structured visual representation, and in which all human interactions with objects are treated as editing operations applied to these visual representations. This "editing paradigm" for human interaction offers the possibility of a simple, powerful and consistent interface based around the physically intuitive notions of space and movement.

- An abstract type hierarchy. A system of abstract types for the description of Eden objects is being designed on top of the concrete notion of type provided by the Eden kernel. One type may be declared as a subtype of another, so that the subtype inherits the operations of its supertype. This type hierarchy, like the subclass hierarchies in Simula [Birtwistle et al. 1973] and Smalltalk [Ingalls 1978; Goldberg et al. 1981], provides a convenient mechanism for factoring information and for defining defaults. Examples of attributes that might usefully be inherited include display code for use with the object editor, and operations concerned with object location.

- A user-level system for naming, storing and retrieving Eden objects, to which we refer as the Eden File System (EFS). EFS will be transaction-based [Gray 1979; Israel et al. 1978; Reed 1978], storing immutable versions that may be replicated at multiple sites for reliability or performance enhancement. The "files" may be single objects or structures of objects. The design is oriented towards providing necessary services plus a base upon which further research and experimentation may be done. For example, concurrency control will be encapsulated to facilitate experimentation with alternate approaches.

Acknowledgements

The Eden project includes roughly six faculty members, fifteen graduate students and five staff members, organized into four working groups. Although this paper was written by the members of the kernel working group, most of the project members have contributed in some way to its ideas. In particular, John Bennett is responsible for coordinating the development of Eden node machines.

Robert J. Fowler is partially supported by an IBM Graduate Fellowship. Henry M. Levy is partially supported by Digital Equipment Corporation.

References

[Almes & Lazowska 1979]
Guy T. Almes and Edward D. Lazowska; "The Behavior of Ethernet-Like Computer Communications Networks"; Proc. 7th Symposium on Operating Systems Principles, Asilomar, December 1979.

[Apollo 1981]
"Apollo Domain Architecture"; Apollo Computer, Inc., February 1981.

[Baskett et al. 1980]
Forest Baskett, Andreas Bechtolsheim, Bill Nowicki and John Seamons; "The SUN Workstation: A Terminal System for the Stanford University Network"; Computer Science Department, Stanford University, March 1980.

[Birtwistle et al. 1973]
Graham Birtwistle, Ole-Johan Dahl, Bjorn Myrhaug and Kristen Nygaard; Simula Begin; Petrocelli/Charter, 1973.

[Cohen & Jefferson 1975]
Ellis Cohen and David Jefferson; "Protection in the Hydra Operating System"; Proc. 5th Symposium on Operating Systems Principles, Austin, November 1975.

[Daley & Dennis 1968]
Robert C. Daley and Jack B. Dennis; "Virtual Memory, Processes, and Sharing in MULTICS"; CACM 11,5, May 1968.

[Eden 1980]
"Eden Project Proposal: Research in Integrated Distributed Computing"; Technical Report 80-10-01, Department of Computer Science, University of Washington, October 1980.

[Ethernet 1980]
"The Ethernet: A Local Area Network. Data Link Layer and Physical Layer Specifications"; Digital Equipment Corporation, Intel Corporation and Xerox Corporation, September 1980.

[Goldberg et al. 1981]
Adele Goldberg, David Robson and Daniel Ingalls; Smalltalk-80: The Language and its Implementation; forthcoming.

[Gray 1979]
J.N. Gray; "Notes on Data Base Operating Systems"; in R. Bayer, et al., eds., Operating Systems: An Advanced Course; Springer-Verlag, 1979.

[Ingalls 1978]
Daniel H.H. Ingalls; "The Smalltalk-76 Programming System: Design and Implementation"; Proc. 5th ACM Symposium on Principles of Programming Languages, Tucson, January 1978.

[Intel 1981]
"Introduction to the iAPX 432 Architecture"; Manual Order Number 171821-001, Intel Corporation, Santa Clara, CA., 1981.

[Israel et al. 1978]
J.E. Israel, J.G. Mitchell and H.E. Sturgis; "Separating Data from Function in a Distributed File System"; Technical Report CSL-78-5, Xerox PARC, 1978.

[Jones et al. 1979]
Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwans and Steven R. Vegdahl; "StarOS, a Multiprocessor Operating System for the Support of Task Forces"; Proc. 7th Symposium on Operating Systems Principles, Asilomar, December 1979.

[Lampson & Redell 1980]
Butler W. Lampson and David D. Redell; "Experience with Processes and Monitors in Mesa"; CACM 23,2, February 1980.

[Liskov 1979]
Barbara Liskov; "Primitives for Distributed Computing"; Proc. 7th Symposium on Operating Systems Principles, Asilomar, December 1979.

[Liskov 1980]
Barbara Liskov; "Linguistic Support for Distributed Programs: A Status Report"; Proc. Workshop on Fundamental Issues in Distributed Computing, Pala Mesa, December 1980.

[Liskov et al. 1977]
Barbara Liskov, Alan Snyder, Russell Atkinson and Craig Schaffert; "Abstraction Mechanisms in CLU"; CACM 20,8, August 1977.

[Liskov & Snyder 1979]
Barbara H. Liskov and Alan Snyder; "Exception Handling in CLU"; IEEE Trans. on Software Engineering SE-5,6, November 1979.

[Metcalfe & Boggs 1976]
Robert Metcalfe and David Boggs; "Ethernet: Distributed Packet Switching for Local Computer Networks"; CACM 19,7, July 1976.

[Newell et al. 1980]
A. Newell, S.E. Fahlman, R.F. Sproull and H.D. Wactlar; "CMU Proposal for Scientific Personal Computing"; Proc. COMPCON 1980.

[Ousterhout et al. 1980]
John K. Ousterhout, Donald A. Scelza and Pradeep S. Sindhu; "Medusa: An Experiment in Distributed Operating System Structure"; CACM 23,2, February 1980.

[Rashid 1980]
Richard F. Rashid; "An Inter-Process Communication Facility for UNIX"; Carnegie-Mellon University, June 1980.

[Reed 1978]
David P. Reed; "Naming and Synchronization in a Decentralized Computer System"; TR-205, Laboratory for Computer Science, MIT, September 1978.

[Ward 1980]
Stephen A. Ward; "TRIX: A Network-Oriented Operating System"; Proc. Compcon 1980.

[Wulf et al. 1974]
William A. Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, Charles Pierson and Frederick Pollack; "Hydra: The Kernel of a Multiprocessor Operating System"; CACM 17,6, June 1974.

[Wulf et al. 1975]
William A. Wulf, Roy Levin and Charles Pierson; "Overview of the Hydra Operating System Development"; Proc. 5th Symposium on Operating Systems Principles, Austin, November 1975.

[Wulf et al. 1981]
William A. Wulf, Roy Levin and Samuel P. Harbison; Hydra / C.mmp: An Experimental Computer System; McGraw-Hill, 1981.