Edmas: A Locally Distributed Mail System

Guy Almes, Andrew Black, Carl Bunje and Douglas Wiebe

Department of Computer Science
University of Washington
Seattle, Washington 98195

Abstract

The Eden Project is a five-year effort to design, build and test operating system structures for local area networks. A specific goal is to allow users to obtain the advantages of both physical distribution *and* logical integration. Since Eden is physically distributed, its users can take advantage of personal workstations. Since Eden is logically integrated, system resources can be named and accessed in a location independent way.

Edmas, the Eden mail system, was designed to be an early test of Eden's usefulness. This paper describes the design and implementation of Edmas, and presents several of the lessons we have learned about constructing location independent services and about the design of mail systems.

1. Introduction

The Eden Project is a five-year effort to design, build and test operating system structures for local area networks [1, 2, 9]. A specific goal is to allow users to obtain the advantages of both physical distribution *and* logical integration. Since Eden is physically distributed, its users can take advantage of personal workstations. Since Eden is logically integrated, system resources can be named and accessed in a location independent way.

The Eden Mail System, or Edmas, is an early attempt to test Eden as a platform for building locally distributed applications.[2] In undertaking our work on Edmas, we adopted three goals:

- We wanted to determine whether Eden could be used to build distributed applications in a short amount of time.
- We also wanted to find out whether Eden would encourage the clean structuring of these applications.
- Finally, we hoped that, in the course of building Edmas, we would be able to use Eden to learn some things about the structuring of mail systems.

This paper reports positively in all three areas.

The paper begins by providing a brief description of the tools Eden provides for application designers, and shows that the Eden kernel and the Eden Programming Language (EPL) combine to allow the application designer to work at a high level. We then present the Edmas design, stressing the clean approach Eden allows for the transport/storage portion of the system. A discussion of the implementation shows how the high-level design was mapped onto EPL routines. Finally, a separate section is devoted to reflecting on the insights gained about Eden, distributed system design, and mail system design.

2. An Overview of the Eden System

The basic unit of system construction in Eden is the *Eject* (for *E*den ob*ject*). An Eden application consists of a collection of Ejects; the mail system that forms the subject of this paper is built from Ejects which represent directories, mail boxes and mail messages.

In this paper we are concerned with the view of Eden seen by a programmer creating an application in Eden; this interface is an "Eden virtual machine" provided by the Eden Programming Language (EPL) [10]. EPL is a superset of Concurrent Euclid [7, 8]; the additional functionality is provided at various levels. At the top, there are a few grammatical constructions in EPL

[1]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 a cooperative research agreement with Digital Equipment Corporation.

[2]By "locally distributed", we refer to applications that are distributed across a local area network. We specifically do not address applications that span long-haul networks or "internets".

0270-5257/84/0000/0056$01.00©1984 IEEE

which do not appear in Concurrent Euclid; these are implemented by means of a preprocessor and two source code generators. At the next level is quite a large library of subroutines that are called by the preprocessor output; underneath that is the Eden kernel itself [3]. Ejects and the Eden kernel itself are separate processes on our host operating system; Eden "system calls" are implemented as a pair of interprocess messages exchanged between a library routine in the Eject and the kernel. Supporting the Eden kernel is the Unix[3] operating system, and under that a network of VAX-11/750 and VAX-11/730[4] processors.

From the point of view of the applications programmer, Eden has the following characteristics.

- Each Eject has a unique identifier. In order for one Eject to communicate with another, the Eject initiating the communication must have a *capability* for the respondent. This capability contains both the unique identifier of the respondent and a set of *access rights*. The integrity of capabilities is guaranteed by the system; neither rights nor identifiers can be forged. Possession of a capability for an Eject does not imply any knowledge of its location within the Eden system.
- Ejects may receive and reply to *invocations* from other Ejects. The EPL code of an Eject defines and exports a number of *invocation procedures* which may be called from other Ejects. For example, if *msg* is a capability for a *MailMsg* Eject, and if *MailMsg*s define the invocation procedure named *Deliver*, then another Eject may make the invocation *msg.Deliver(<parameter list>)*.
- Each Eject has a concrete *type*, that is, a code segment that defines the set of invocations to which the Eject will respond. Eden types are conceptually similar to the collection of methods that make up a Smalltalk Class. The type-code of an Eject is actually a complete EPL program.
- Each Eject also has a data part: this corresponds to the set of variables and data structures defined by the type-code. The data part contains two kinds of data structures. Some make up the "long term state", the set of values maintained between invocations that defines the state of the Eject. Others, such as the local variables and parameters of invocations, are genereally not significant between invocations and are called the "short term state".
- Each Eject typically consists of a number of processes which communicate *via* monitors [6]. When one process invokes another Eject, it blocks until the invocation completes, but the other processes within the Eject may continue to run. Ejects may be thought of as active at all times; in addition to processes that respond to invocations and processes that make invocations of other Ejects, an Eject may contain processes that perform internal housekeeping operations. This contrasts with Smalltalk, where sending a message transfers control to the receiver, and replying to a message causes the receiver to quiesce.
- An Eject may perform a *Checkpoint* operation. The effect of Checkpointing is to create a *Passive Representation*, a data structure designed to endure system crashes. The data in a passive representation should be sufficient to enable the Eject they represent to re-construct its long term state. The checkpoint primitive is the only mechanism provided by the Eden kernel whereby an Eject may access permanent storage, i.e. the disk.
- In practice, Ejects are not always active, either because they (or their computers) have crashed, or because they have explicitly deactivated themselves in order to economize on the use of system resources. However, if a passive checkpointed Eject is sent an invocation, it will be activated automatically by the Eden kernel. When an Eject is so activated it will normally attempt to read its Passive Representation to put its internal data structures into a consistent state.

3. Mail System Design

Electronic Mail Systems may be divided into two parts. One is the interface seen by the user who sends and reads mail. The other is the transport and storage system that actually moves the messages around the system.

Any complete mail system must, of course, include both parts. The Eden demonstration mail system (Edmas) is innovative in the way in which it provides the transport and storage layer. We were eager to concentrate our efforts in this area because the problems encountered there are those directly addressed by Eden. In contrast, the user interface of Edmas is a low-cost adaptation of an existing Unix utility. This is because the prototype implementation of Eden on which the mail system was built did not include any facilities at all for communicating with a user terminal. Input and output had to be performed by "escaping" to the Unix system call interface.

3.1. Mail Transport and Storage

In most electronic mail systems, mail messages are implemented as a sequence of characters. Each message is either a text file or a segment of a larger file. In fact, messages do have an internal structure; there is a *From* field, a *To* field, a *Subject* field and so on. An Eject of type *MailMsg* implements this abstraction.

An Edmas *MailBox* is modelled on the receptacle at the end of an American driveway; mail is delivered to it

[3]Unix is a trademark of Bell Laboratories.

[4]VAX is a trademark of Digital Equipment Corporation.

and stays there until the owner picks it up. It does not attempt to be a filing system for old mail messages; we view this as a separate utility (and part of the user interface).

Both MailBoxes and MailMsgs are Ejects, and are thus referred to by capabilities. In particular, the *To* and *From* fields of a MailMsg are most naturally thought of as capabilities for MailBoxes. Because capabilities cannot be typed at a terminal, the user interface must provide some other way of referring to them. The mechanism we chose was the (already existing) type *Directory*, which implements a mapping from strings to capabilities.

Because capabilities refer to Ejects in a location independent way, and any Eject can be invoked by any other, regardless of their physical locations, there is no explicit transport mechanism in Edmas. We will now describe the principal Edmas types in a little more detail.

The MailMsg Eject

The MailMsg Eject implements a single mail message. Its long term state comprises the capability for the MailBox of the creator of the message (conceptually, the *From* field), a list of capabilities for the addressees' MailBoxes (the *To* field), a subject description string (the *Subject* field) and a message text string (the *Text* field). There is no need to explicitly record the date and time at which the message was sent because the date of creation of the MailMsg eject itself serves this purpose.[5]

Set and *Append* invocations allow calling Ejects to compose the message by initializing the various conceptual fields. *Get* invocations allow retrieval of these fields. The end of the composition process is indicated by the *Deliver* invocation, which has two effects. First, the MailMsg *freezes* itself, meaning that any further requests to alter the fields will be refused. Secondly, the MailMsg attempts to deliver itself to all the MailBoxes.

The MailBox Eject

The MailBox Eject need not contain copies of unread messages, merely capabilities for them. Its long term state contains a queue of MailMsg capabilities in order of delivery. Each Eden user has a personal MailBox to which capabilities for incoming MailMsgs are delivered.

[5]Given a capability for any Eject, the Eden kernel will provide its creation time on request.

The *MailBox.Deliver* invocation places an incoming MailMsg capability in the queue. MailBoxes allow their owner to retrieve the oldest MailMsg capability using the operation *Pickup*. A separate operation *Remove* is used to remove a MailMsg capability from the MailBox.

MailBoxes also maintain a "printable" name for their owners, and two Eden time stamps, one marking the last time that a MailMsg was delivered to the MailBox and the other the last time that a MailMsg was picked up or removed from the MailBox. Invocations allow calling Ejects to set the value of the name field and to retrieve the name, time stamps and a count of the number of messages in the MailBox. The printable name is used to construct the *From* and *To* fields when messages are displayed to the user.

Only *DeliverRts* are needed to deliver a message to a MailBox; ordinarily, capabilities for MailBoxes with *DeliverRts* would be available in a public directory. In order to pick up mail and remove mail, *PickupRts* are needed; ordinarily, the owner of a MailBox would retain these rights. Setting the name of a MailBox requires administrator rights; this is to prevent the owner of the MailBox from impersonating someone else.

3.2. User Interface

From the above descriptions, the "natural" interface to the mail system ejects is a set of invocations. Given a command language interpreter that permitted the user to create and invoke Ejects, mail messages could be sent without further work.

There are two problems with such an interface. First, it is very inconvenient; an editor-like interface is much easier to use. Secondly, at the time that Edmas was written, there was no command language interpreter for Eden.

The solution we adopted was to build editor interfaces to send and receive mail, and a special purpose command interpreter (the *MailBox Manager*) to create MailBoxes and enter them in the mail box directory. The editor interfaces are built on top of the Unix Emacs editor [4], which provides some quite general mechanism for starting Unix processes and sending them input collected from the editor's buffers. Of course, Emacs is incapable of making invocations, so two very primitive command language interpreter Ejects were built. We call these Ejects *interfaces* not because they are the part of Edmas seen by the user (they are not) but because they interface Eden to Unix. They accept Unix standard input, perform the ap-

propriate Eden invocations, and then generate Unix standard output.

The *Send Interface* accepts a sequence of characters which describe the fields of a message; its action is to create the message and tell it to deliver itself. The *Receive Interface* works similarly. Because the "languages" recognized by these interfaces are very simple, and because the editor macros ensure that their inputs are syntactically correct, the interfaces are simple Ejects.

3.3. A Session with Edmas

In order to clarify the way the various pieces of the mail system fit together, this section outlines the way Edmas is actually used.

Suppose that Carl Bunje wishes to send mail to Doug Wiebe. Within Emacs, he calls the editor macro *edmas-send-mail* by touching the appropriate key. A window appears on his screen which contains blank *To* and *Subject* fields. At the same time, an instance of the *Send Interface* Eject is created which locates a capability for Bunje's own MailBox. Bunje composes his message, filling in the *To* field with Doug Wiebe's *lookup name*, i.e. the name under which Wiebe's MailBox is entered in the mail box directory.[6] When Bunje is satisfied with the content of the message, he touches the *send* key on his terminal; this causes the text of the message to be piped to the *Send Interface*, and the window containing the message to disappear.

The *Send Interface* takes each name in the *To* field and looks it up in the mail box directory. In our case there is only one name, *wiebe*. Providing all the names are found, *Send Interface* creates a new MailMsg Eject and sets the various fields appropriately. If any name is not found, *Send Interface* does not create a message; it outputs a warning which is returned to the user in a *Send Errors* window of the editor.

The situation is now as shown in Figure 3-1. Each solid-headed arrow in the figures represents a capability. A is the capability held by the send interface for the mail message it has created. B_1 is the capability for Bunje's own MailBox held by the *Send Interface*, and B_2 is the same capability once it has been installed in the mail message. C_1 is the *Send Interface*'s capability for the mail box directory, wherein it looks up *wiebe* to obtain C_2, a capability for Wiebe's MailBox. This is installed in the message as C_3.

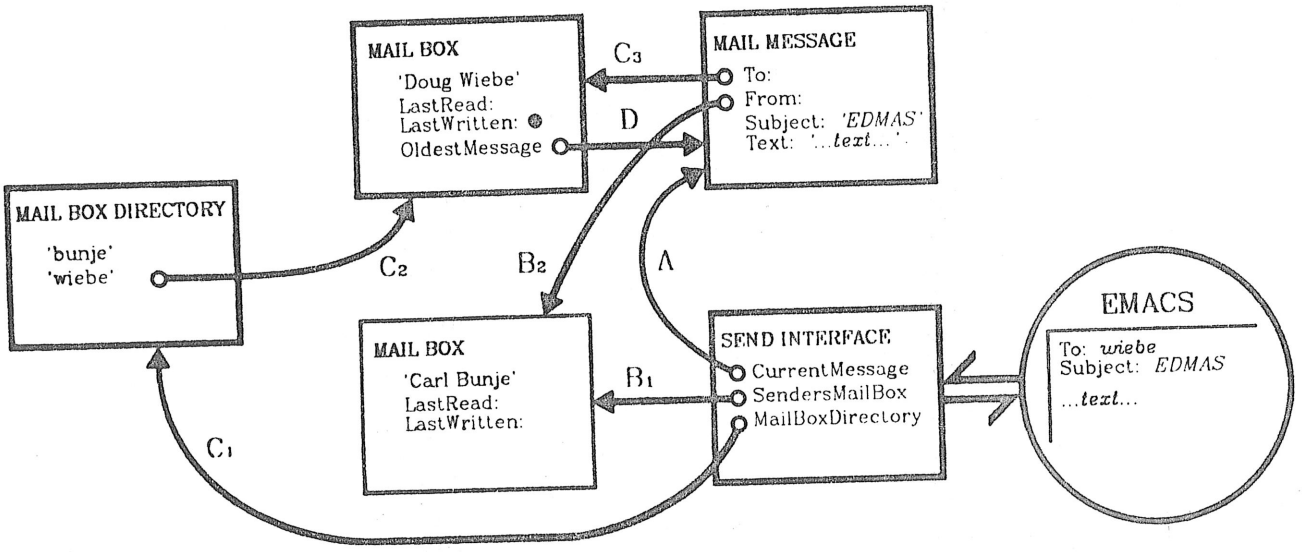

Figure 3-1: A MailMsg being Delivered

Once the message has been created, the *Send Interface* invokes *A.Deliver*. This causes the message to invoke *Deliver* on each of the MailBoxes in its *To* field; in this case, just C_3.*Deliver*. This invocation gives Wiebe's MailBox capability D in Figure 3-1: mail delivery is complete.

[6] Our convention is to use Unix login names for this purpose, but there is nothing that prevents the same MailBox appearing more than once in the mail box directory, and thus having more than one lookup name.

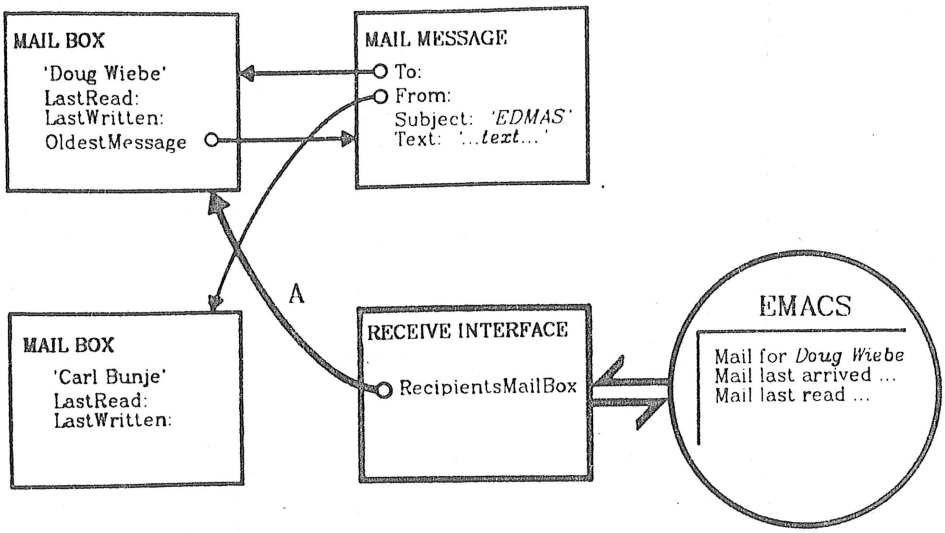

Figure 3-2: A MailBox Show Occurs

At some later time Doug Wiebe calls the editor macro *edmas-pickup-mail*. This creates an instance of the *Receive Interface*, which obtains a capability for Wiebe's own MailBox with *PickupRts* (capability *A* in Figure 3-2). Emacs asks it to performs a *Show* invocation on the MailBox, and displays the results in a window.

Wiebe sees that he has new mail, and touches the *pickup* key on his terminal. Emacs requests the *Receive Interface* to pickup and remove the oldest message from his MailBox. This is represented by capability A_2 in Figure 3-3, which becomes A_3 when it is returned to the *Receive Interface*. The *Receive Interface* then invokes A_3 to obtain the fields of the message, and pipes the resulting text to Emacs, which displays it in a window. Note that the name in the *From* field is the print name of Bunje's MailBox (obtained by invocation *via* B_1), not Bunje's lookup name.

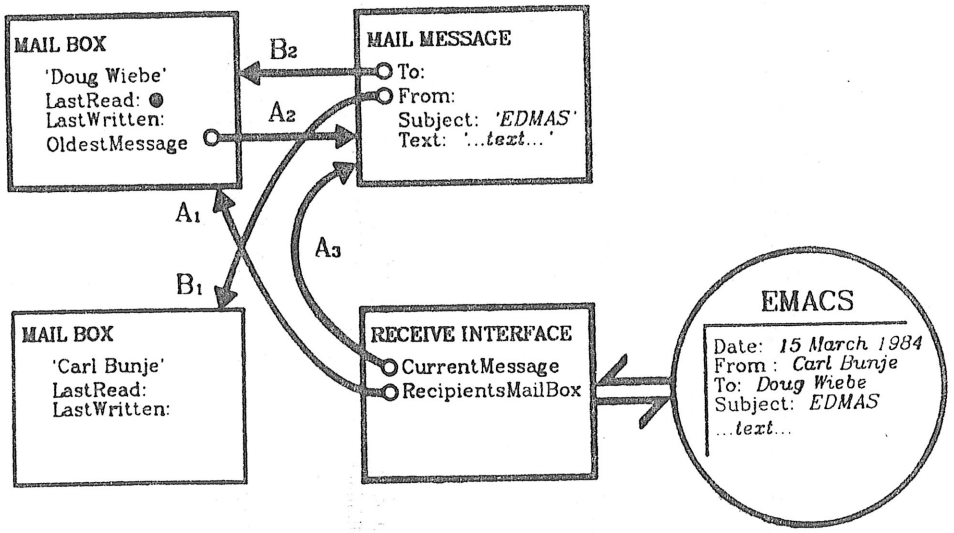

Figure 3-3: A MailBox Pickup Occurs

4. Mail System Implementation

In this section, we will describe how the Edmas design was translated into a working system. We first set the context for the implementation effort, then explore one of the five Eden types in detail to give the reader a feel for what programming in Eden is like. Finally, we discuss issues of productivity and bug histories.

4.1. Context

During the first week of May 1983, two of us[7] developed the design to the point where we felt that we could turn it over to the implementors.[8] In addition to discussions of the design ideas, we were able to use two more formal tools for conveying the design:

- a context-free description of the language accepted by the interface types, and
- an EPL module that defined the syntax of the invocations to be implemented by the MailBox and MailMsg types.

4.2. Implementing the MailBox Type

In this section we will carefully examine the implementation of the MailBox type, one of the five types produced for Edmas. Particular attention will be paid to the use of processes and monitors at the language level, and to the extensions to Concurrent Euclid provided by EPL. The reader may want to refer to Section 3.1 for specific aspects of the MailBox design.

Specifying an Invocation

The specification for the MailBox type included a description of the invocations available to users of the type. One of these is *Pickup*, whose specification is shown in Figure 4-1. The syntactic construct of the invocation procedure is an EPL feature not present in Concurrent Euclid. In every invocation procedure, the first (in) parameter is the set of access rights to the MailBox provided by the invoker; these can be checked by the MailBox to verify that the invoker is allowed to perform the invocation. Similarly, the final (out) parameter of each invocation procedure is a status to be returned to the invoker; one example of an error status that applies to all invocations is the *InsufficientRights* status. The more interesting parameters are WasPresent and Missive, (out) parameters through which the MailBox communicates to the invoker whether an unread MailMsg was present and, if so, a capability for one.

---

invocation procedure PickUp(
  CallersRights : EdenRights,
  var WasPresent : Boolean,
  var Missive : Capability **for** MailMsg,
  var Status : EdenStatus) =
external
{ Requires: *PickUpRts*.

Effect: *If a MailMsg is in the box, then the oldest one is returned to the caller along with a true WasPresent.*

If no MailMsg is present, then a false WasPresent value and a null Missive are returned.

In either case, the LastRead time stamp is updated.

Special Error Statuses: *none*. }

Figure 4-1: Specification of the Pickup Operation

Representation

Together with specifying the set of operations to be supported, the designer describes the abstract long term state of the MailBox. Two parts of the state of a MailBox that relate to the Pickup invocation are the queue of (capabilities for) unread MailMsgs and the time stamp of the most recent Pickup operation.

Recall that an Eject's long term state takes on two forms: the data part of the executing Eject, and a passive representation for use in checkpointing to permanent storage. For both of these forms, the time stamp of the most recent Pickup is implemented as the kernel-defined type *Kernel.Timestamp*.

The queue of capabilities for unread MailMsgs, however, is implemented differently in the two forms. In the *active representation* of a MailBox, the queue is implemented as a doubly linked list of records; each record contains a capability for a single MailMsg. In the *passive representation* of a MailBox, the queue of MailMsgs is implemented as a contiguous sequence of records. This use of different concrete structures has two particular advantages:

- The active representation of the queue makes insertions and deletions efficient, while the passive representation conserves on storage space.
- The passive representation is very robust with respect to likely changes in the implementation of the MailBox type. Consequently, new versions of the MailBox type code can be installed, even if the new version differs from the old version in the concrete structures used for the active representation. This avoids the *replugging* difficulties described by Goullon *et al.* [5].

[7] GA and AB.

[8] CB and DW.

Use of Processes and Monitors

Each active form in Eden executes within a single operating-system-level process with a conventional address space. The Concurrent Euclid language, however, provides multiple language-level processes for use by the programmer of Eden types. The MailBox type, for example, has several processes, some of which respond to invocations directed to the MailBox. This allows the MailBox to continue to respond to incoming invocations even if one invocation takes a long time or blocks on a logical condition.

```
var TimeStamps : monitor
   imports({ mods}  var Kernel)
   exports({ procs}  UpdatePickup, GetPickup,
                     UpdateDelivery, GetDelivery, Checkpoint,
                     Restore, Initialize)

   var TimeOfLastPickup : Kernel.TimeStamp
   var TimeOfLastDelivery : Kernel.TimeStamp

   procedure UpdatePickup = ...
   procedure GetPickup(var TS : Kernel.TimeStamp) = ...
   procedure UpdateDelivery = ...
   procedure GetDelivery(var TS: Kernel.TimeStamp) = ...
   procedure Checkpoint = ...
   procedure Restore = ...
   procedure Initialize = ...

end { TimeStamps} monitor
```

Figure 4-2: Sketch of the MailBox Timestamp Monitor

In order to ensure that concurrent activity of these multiple processes does not corrupt the representation, each part of a MailBox's representation is protected by a monitor. More generally, each invariant defined on the data part is protected by a monitor. As a simple example, there is a monitor that surrounds the declarations for the Timestamps used in a MailBox. This monitor is sketched in Figure 4-2. In addition to controlling concurrency, monitors function as modules. Thus all knowledge of the concrete data structures used to implement Timestamps is localized within this monitor.

Implementing the Invocations

Once the Eject's representation is defined with these monitors, the invocations can be implemented procedurally. In Figure 4-3, the Pickup invocation procedure is shown. Since the MailMsg queue and time stamps are implemented as separate monitors, the code consists principally of checking the invoker's access rights, then calling the appropriate monitor procedures.

```
invocation procedure PickUp(
   CallersRights : EdenRights,
   var WasPresent : Boolean,
   var Missive : Capability for MailMsg,
   var Status : EdenStatus) =
imports({ mods}  var Kernel, var StatusDefs,
        { mons}  var TimeStamps, var Messages)
begin
   if (PickUpRts not in CallersRights) then
      StatusDefs.SetStatusCode(Status,
            StatusDefs.EPLInsufficientRights)
      WasPresent := False
      Kernel.CapaMakeNull(Missive)
   else
      TimeStamps.UpdatePickUp
      Messages.GetOldestMessage(Missive, WasPresent)
   end if
end PickUp
```

Figure 4-3: Implementation of the Pickup Invocation

4.3. Experiences in Implementing the Mail System

One of the most pleasing aspects of our experience with the mail system was the consistent productivity experienced during its construction. During a four-week period, two students used a new system and a new language to build a nontrivial application of moderate size.

This productivity was achieved despite a very hostile debugging environment caused by instabilities in the new Eden kernel and undebugged Directory system. In fact, in the two weeks between the first attempts to run the mail system and the demonstration of the system to outside reviewers, more time was spent using the mail system to debug the kernel and the Directory system than was used to debug the mail system itself. Apart from the skill of the implementors, we feel there are several reasons for this experience:

- Concurrent Euclid's strong typing caused many bugs to be exposed at compile time.
- The EPL extensions to Concurrent Euclid provided linguistic support at precisely those areas, such as operating system interfaces, where most languages leave the programmer in an error-prone situation.
- The interfaces between Eject types was precisely specified at the syntactic level *via* EPL. While the semantics of these interfaces were described in English, they were quite clean.
- There were very few places where the mail system programmers had to circumvent misfeatures of the Eden kernel or of EPL. The worst case of such circumvention we did experience relates to an invocation of the MailBox type which passes Timestamps as parameters. Due to restrictions on the types of parameters which can be passed in EPL invocation procedures, these Timestamps parameters had to be explicitly converted to/

from byte sequences. This was, however, the worst case of kludgery required in the system.

5. Insights Gained

The Eden mail system is one of the first applications to be built on the Eden system. We feel that it illustrates a number of things about Eden, which will be referred to under the following headings:

- Location Independence
- The Power of Capabilities
- Extensibility
- Programmability

5.1. Location Independence

Location independence was always one of the design goals of Eden: a user need not know the location of an Eject in order to use its services. In the context of the Eden mail system, this means that the space of mail addresses is system-wide, that a user can read mail from any computer in the Eden system, and that no distinction is made at mail delivery time between local and remote mail.

Conventional electronic mail systems achieve some of this functionality, but in less convenient ways. Mail can be re-directed to a fixed site, but only by creating a mail alias on every other machine in the network. Mail can be read from a remote site only by performing a remote login. Mail delivery typically requires the use of many different programs which understand complicated addressing schemes, know about the network topology, and engage in conversations with remote nodes.

In Eden, location-independence is achieved without any effort on the part of the mail-system designers. Nowhere in the code of the mail system is the existence of the network even mentioned. As a consequence, the size and complexity of the Eden mail system is much less than that of a conventional network mail system, a point we return to in Section 5.4.

5.2. Power of Capabilities

The use of capabilities is central to the Eden mail system. The MailMsg itself is referred to by capability. MailMsgs are addressed to capabilities, and contain the capability of the sender. One of the consequences of this arrangement is that there must be a single registry in which one can look up the capabilities for all the addressees mailboxes. Within our own local area network, this is an advantage; it is certainly preferable to having a mail re-direction file on every machine mentioning every user. In a larger network, the use of a single registry would seem to be unworkable. On reflection, however, one realises that although the registry is conceptually a single service, it can be implemented as a collection of communicating Ejects distributed across the network, and using replication to achieve a high degree of availability.

Within the mail system, the use of capabilities as addresses has several advantages over the strings used for this purposes by conventional mail systems:

- There is no need to parse strings like *To: Andrew P. Black <black@uw-wally>* in order to decide what to do with a MailMsg.
- The translation from mnemonic names to capabilities is performed by a directory lookup. The directory is a perfectly ordinary Eden type specialized for this task. It was actually designed and built for use in the Eden file system, but its function is so general and useful that we expect it to be re-used in many applications.
- In order to reply to a MailMsg, one simply extracts the *From* field of the incoming MailMsg and makes it the *To* field of the reply. There is no need to edit a string to add or remove environment dependencies.

Of course, these advantages apply only within Eden, and if at some future time we wish to extend Edmas so that it can be used to communicate with non-Eden sites, say over the Arpanet or *via* uucp, some modifications would be necessary.

Because the MailMsg itself is uniquely identified by a capability, the reply can contain a field called *In-reply-to* that has as its value a Capability for the original MailMsg. An Eden mail interface would typically enable the reader to ask to see the MailMsg named by the *In-reply-to* field; the implementation of this mechanism would be independent of any directory of old mail that the user might or might not keep.

Additionally, Eden capabilities provide protected access rights, and Edmas benefits from this facility. As a consequence, there are no places where Edmas needs to exercise special privileges not available to other Eden types. This contrasts with most mail systems, where the protection needs of the mail system conflict with the facilities provided by the underlying operating system.

5.3. Extensibility

We believe that the Eden mail system achieves a high degree of modifiability and extensibility. Since it has existed for only a few weeks, there has so far been no opportunity to test this belief. Nevertheless, we feel

that there are convincing arguments on our side; as an example, let us consider how mail distribution lists and confidential mail could each be added to the Eden mail system.

Distribution Lists

Let us hypothesize the addition of another type of Eject to the mail system: let us call it a distribution list type. One essential characteristic of a distribution list is that mail can be addressed to it. Edmas uses directory lookup to perform the translation between the addresses that the user enters at the keyboard and the capabilities that form the *To* field of the MailMsg. Provided that the required distribution list is entered in the MailBox directory under a suitable name, no changes will be necessary to the MailMsg type or to the user interface.

Another essential characteristic of a distribution list is that mail can be delivered to it. In Edmas, this means that it understands the invocation *Deliver*, and respond appropriately. The appropriate response is, of course, to *Deliver* the MailMsg to all the MailBoxes on the list. In addition to *Deliver*, the distribution list would respond to invocations for constructing and displaying the persons on the list.

In essence, then, the role of a distribution list is twofold. From the point of view of a MailMsg, it has to support the *MailBox. Deliver* invocation as would a MailBox. From the point of view of a MailBox, it has to invoke this *MailBox. Deliver* invocation as would a MailMsg. Both of these things are easy to do because the only attributes of an Eject that can be observed from outside are whether a particular invocation is implemented, and the results of employing it if it is.

One ground rule that has helped us to achieve this sort of modularity is that the effect of an invocation should not depend on the identity of the invoker. Although invocation messages do contain a capability for the invoker, we have adopted the convention that the invokee should not look at it. A parallel may be drawn with programming languages: the effect of a particular procedure call should not depend on who makes it. Even though the return address is on the execution stack and could easily be accessed, procedural programming languages do not provide a primitive to do so. The semantics of procedure call would be greatly complicated by such a provision.[9]

[9]We do not take steps to conceal the capability of the Invoker, however; the parallel with programming languages convinces us that it may be very useful in debugging.

As a concrete example, the *MailBox. Deliver* invocation requires as one of its arguments the capability for the MailMsg which is to be inserted into the MailBox. In the existing mail system the only Eject which makes this invocation is the MailMsg itself; nevertheless, we chose the general invocation in preference to a more restricted *DeliverMyself* operation. Because of this, it is easy for distribution lists to deliver MailMsgs to MailBoxes.

Confidential Mail

Another possible extension, the provision of confidential mail, makes use of the access rights fields in Eden capabilities. As the mail system currently stands, it is possible for a user to give her secretary a capability for her MailBox. This will enable the secretary to read all of the user's mail. An attractive extension would be for senders to be able to designate certain MailMsgs as "confidential"; the secretary would be able to read only the non-confidential MailMsgs.

Implementing this extension would involve:

1. Augmenting the send interface so that MailMsgs can be marked as confidential;
2. Augmenting the MailMsg type to remember whether or not it is confidential, up until the time it delivers itself;
3. Adding an argument to MailBox.Deliver so that a MailMsg can tell a MailBox whether or not it is confidential;
4. Changing MailBox.Pickup so that it checks the access rights of its invoker, and returns the capability for a confidential MailMsg only if the invoker has *OwnerRts* in addition to *PickupRts*.

This example illustrates how capabilities and access rights help us to resolve the tension between modularity and protection. In a modular system the effect of a particular operation depends only on its arguments, and not on who makes the request; modularity is essential if one hopes to be able to re-use components. In a protected system only particular users and/or programs are allowed to perform sensitive operations. There is clearly a conflict between these two desiderata; Eden (and other capability-based systems) resolve it by putting all protection into the part of the system which distributes capabilities, and making all other operations work for any caller with sufficient rights.

5.4. Programmability

The final topic in this section is the programmability of Eden. By this we mean the ease (or otherwise) with which a complicated service can be built within the

Eden system. Our experience here has been very positive. Edmas was constructed over a period of four weeks by two graduate students, who were also taking a full load of courses. At the end of that time the system was well-enough developed and stable enough to be demonstrated to visitors from outside the department. Neither of the students had previously been involved with Eden, so this time includes the overhead of familiarization with the system. Furthermore, Edmas was one of the first applications to be constructed on top of Eden; as a result, many latent bugs in the Eden system kernel were discovered during its development. The resolution of these problems inevitably slowed down development of the mail system itself.

It should also be mentioned that of the six types which form part of the prototype mail system, only two have very much to do with mail! Of the others, one is the directory type, which was constructed as part of the Eden file system. The other three are special-purpose command language interpreters: one for invoking directories and initializing MailBoxes, one for creating MailMsgs, and one for invoking MailBoxes and reading mail. Our plans for the further development of Eden include the design and construction of a general purpose command language interpreter; if this had existed at the time that the mail system was written, the first of these (the MailBox Manager) would probably never have been written. The send and receive interfaces that we have implemented, integrated as they are with a text editor, are of course substantially better than would be provided by a general purpose command language. Nevertheless, the need to construct a command interface before the crudest prototype MailMsg could even be tested hardly accelerated the development of the system.

In spite of all these handicaps, the code of the mail system types is well structured and readable: it was sufficiently clear for us to distribute it to external reviewers as an example of an Eden application program. We feel that these early experiences tend to vindicate our belief that the model of computation that underlies Eden is simple enough to permit distributed applications to be built easily. We also believe that it possesses enough generality for the construction of fully functional designs. However, the current mail system is clearly a "toy" application, and full substantiation of this belief will involve further experiments.

6. Conclusion

The Eden Mail System has accomplished the goals we set for it:

- First, it has shown that it is possible to use Eden to construct distributed applications rapidly and systematically.
- Second, these applications can be easy to build and can exhibit clean structure.
- Finally, the structure of Edmas cleanly reflected the abstractions that are natural to mail: MailBoxes and MailMsgs, rather than files and user-ids that are the only tools available to designers on more conventional systems. This structuring advantage has allowed us to develop insights in the area of mail system design that might have been obscured to designers on other systems.

Acknowledgements

The Eden project currently includes about six faculty members, fifteen students, and five staff members. They have all contributed to the system that made this research possible. We would specifically like to mention Cara Holman, who now maintains and is extending the system. Also, we would like to recognize Jan Sanislo for his leadership in the construction of the Eden kernel and Norm Hutchinson and Barry McCord for their contributions to the EPL programming environment.

References

[1] Guy T. Almes.
Integration and Distribution in Eden.
In *Proc. International Workshop on Computer Systems Organization.* IEEE Computer Society, New Orleans, March, 1983.

[2] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe.
The Eden System: A Technical Review.
Technical Report, Dept. of Computer Science, Univ. of Washington, October, 1983.

[3] Guy Almes, David Jacobson, Leif Nielsen, and Jim Rees.
Kernel-Eject Interface Specification.
1983.
Eden Project Internal Document.

[4] James Gosling.
Unix Emacs.
Technical Report, Carnegie-Mellon University, May, 1982.

[5] Hannes Goullon, Rainer Isle, and Klaus-Peter Loehr.
Dynamic Restructuring in an Experimental Operating System.
In *Proc. Third International Conference on Software Engineering*, pages 295-304. IEEE Computer Society, NBS, and ACM, Atlanta, May, 1978.

[6] C. A. R. Hoare.
Monitors: An Operating System Structuring Concept.
Comm. of the Assoc. for Computing Machinery 17(10):549-557, October, 1974.

[7] R. C. Holt.
A Short Introduction to Concurrent Euclid.
SIGPLAN Notices 17(5):60-79, May, 1982.

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

[9] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal.
The Architecture of the Eden System.
In *Proc. Eighth Symposium on Operating Systems Principles*, pages 148-159. Assoc. for Computing Machinery, Pacific Grove, December, 1981.

[10] Barry C. McCord and Andrew P. Black.
EPL Programmer's Guide.
1983.
Eden Project Internal Document.