MIT/LCS/TR-376

REPLICATION AND RECONFIGURATION IN A DISTRIBUTED MAIL REPOSITORY

Mark S. Day

April 1987

Replication and Reconfiguration in a Distributed Mail Repository

by

Mark S. Day

February 1987

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Replication and Reconfiguration in a Distributed Mail Repository

by

Mark Stuart Day

Submitted to the
Department of Electrical Engineering and Computer Science
on December 22, 1986 in partial fulfillment of the requirements
for the Degree of Master of Science
in Electrical Engineering and Computer Science.

Abstract

Conventional approaches to programming produce *centralized programs* that run on a single computer. However, an unconventional approach can take advantage of low-cost communication and small, inexpensive computers. A *distributed program* provides service through programs executing at several nodes of a distributed system. Distributed programs can offer two important advantages over centralized programs: high availability and scalability. In a highly-available system, it is very likely that a randomly-chosen transaction will complete successfully. A scalable system's capacity can be increased or decreased to match changes in the demands placed on the system.

When a node is unavailable because of maintenance or a crash, transactions may fail unless copies of the node's information are stored at other nodes. Thus, high availability requires *replication* of data. Both the maintenance of a highly-available system and scalability require the ability to modify and extend a system while it is running, called *dynamic reconfiguration* or simply *reconfiguration*.

This thesis considers the problem of building scalable and highly-available distributed programs without using special processors with redundant hardware and software. It describes a design and implementation of an example distributed program, an electronic mail repository. The thesis focuses on how to design and implement replication and reconfiguration for the distributed mail repository, considering these questions in the context of the programming language Argus, which was designed to support distributed programming.

The thesis makes three distinct contributions. First, it presents the replication techniques chosen for the distributed repository and a discussion of their implementation in Argus. Second, it describes a new method for designing and implementing reconfigurable distributed systems. The new method allows replacement of software components while preserving their state, but requires no changes to the underlying system or language. This contrasts with previous work on guardian replacement in Argus. Third, the thesis evaluates the utility of Argus for applications involving replication and reconfiguration.

Thesis Supervisor: Barbara H. Liskov
Title: Nippon Electric Company Professor of Software Science and Engineering

Keywords: data replication, software reconfiguration, availability, reliability, scalable systems, distributed programs, electronic mail repositories, programming languages

Acknowledgments

I thank my advisor, Barbara Liskov, for her excellent comments and precise editing. The style and content of this thesis have been improved enormously by her suggestions. I can only hope that my thinking and analytical abilities have been improved as much.

Thanks to Boaz Ben-Zvi, Toby Bloom, Craig Chambers, Janice Chung, Dave Clark, Ken Goldman, Dave Gifford, Paul Johnson, Elliot Kolodner, Rivka Ladin, Mark Lambert, Gary Leavens, Jim O'Toole, Sharon Perl, Jim Restivo, Jerry Saltzer, Bob Scheifler, Rob Seliger, Mark Tuttle, Mike Vermeulen, Ed Walker, and Bill Weihl for many helpful discussions and valiant readings of countless drafts of this thesis. Bob and Paul earn special thanks for having created and maintained the Argus implementation I used for this thesis, and for answering a steady stream of questions. Mark Lambert gets a medal for his role in defining and implementing PCMail and for persisting in helping me to debug the communication between our implementations of DMSP.

Prof. Gruia-Catalin Roman deserves a large amount of the credit (or blame) for me being in graduate school, so it seems appropriate that he share the credit (or blame) for this thesis.

Thanks to friends who exerted a strong influence in keeping me sane: Hằng Trần (who is just plain wonderful), Tangi Anderson, Craig Chambers (life at MIT has been much duller since he went to Stanford), Virginia Cox, Mark Derthick, Ed Frankenberry, Arnold Juster, Jon McCombie, Lynette McMahon, John Morrison, John Norwood, Ellen Pint, Gabe Spalding (who heroically tried to talk me out of going to graduate school), and Edward Zamboni.

All of the members of my family have been supportive of my work at MIT: thanks to Father, Mother, Michael and Jane, Malcom, and Jonathan. I couldn't wish for a family that I would respect, admire, or love more.

Most of all, thanks to a truly supportive, helpful, and generally fantastic person, unquestionably my "better half": Hằng Trần. Anh thương em nhiều lắm, y'know?

Cho em Hằng:
Lạ gì thanh-khí lẽ hằng,
một dây một buộc ai giằng cho ra.

Table of Contents

Table of Figures

Chapter One

Introduction

Paradise is exactly like
where you are right now,
only much, much better.
—Laurie Anderson

The truth is rarely pure,
and never simple.
—Oscar Wilde

For some time, the cost of computers and communications has been decreasing, while their speed and capacity have been increasing. Whereas conventional approaches to programming produce *centralized programs* that run on a single computer, an unconventional approach can take advantage of low-cost communication and small, inexpensive computers. A ***distributed system*** is a collection of computers connected to a communications network, able to send information over geographic distances (a few meters to thousands of kilometers). In a distributed system, the processing elements (called *nodes*) are far apart compared to the processing elements of multicomputers or multiprocessors. Thus, communication between nodes of a distributed system is expensive compared to the cost of local computation. A ***distributed program*** provides service through programs executing at several nodes of a distributed system.

Distributed programs can be useful solutions for certain application areas. This thesis considers applications that do not have hard real-time deadlines, but require continuous availability of consistent data. For example, a reservation system for a world-wide airline is an example of a candidate application for a distributed program. Users can tolerate delays of several seconds, but the system is *always* in service. Loss of service, whether due to maintenance or equipment failure, causes a loss of revenues and may completely paralyze operations dependent on the system [17].

Ideally, distributed programs can offer two important advantages over centralized programs: high availability and scalability. Availability is typically defined in terms of the probability $P_{success}$ that a randomly-chosen transaction will complete successfully. High availability correspondingly means that this probability is very close to one. My arbitrarily-chosen definition of *high availability* is

$$P_{success} \geq 0.997$$

This probability corresponds to less than five minutes unavailable out of each 24 hours, assuming a uniform distribution of failures. High availability is possible in a distributed system because the system contains multiple processors. If one node of a distributed system fails, some good nodes are still available; the tasks of the failed node may be divided up between the remaining nodes, so that the system continues to function, though perhaps at reduced capacity.

Scalability means that the system's capacity can be increased or decreased to match changes in the demands placed on the system. Scalability is possible in a distributed system because the system's capacity may be increased by adding nodes, or decreased by removing nodes. These changes of scale can take place over a greater range and with greater ease than is possible for a single machine.

This thesis addresses the problem of building scalable and highly-available distributed programs without using special processors with redundant hardware and software. Since many organizations have invested a great deal in various sorts of conventional computer hardware and networks, it is not realistic to demand a large investment in new hardware and networks to achieve high availability and scalability. Instead, it should be possible to take a collection of conventional machines and a conventional communications network, and use them to construct a highly-available distributed program.

To achieve high availability, a distributed program must tolerate node failures, or *crashes*, and tolerate maintenance. The program must also tolerate communication failures, or *partitions*. The problems of partitions are discussed in more detail in Chapter 4. When a node is unavailable because of maintenance or a crash, the system may be unavailable unless copies of the node's information are stored at other nodes. Thus, high availability requires *replication* of data. In addition, both maintenance of a highly-available system and scalability require the ability to modify and extend a system while it is running, called *dynamic reconfiguration* [25] or simply *reconfiguration*. So high availability and scalability require replication and reconfiguration (see figure 1-1).

Since there is no general agreement on a method for constructing distributed programs, I have chosen to examine the issues involved in the design and implementation of a particular application. The experience and insights from this application may contribute to a future method

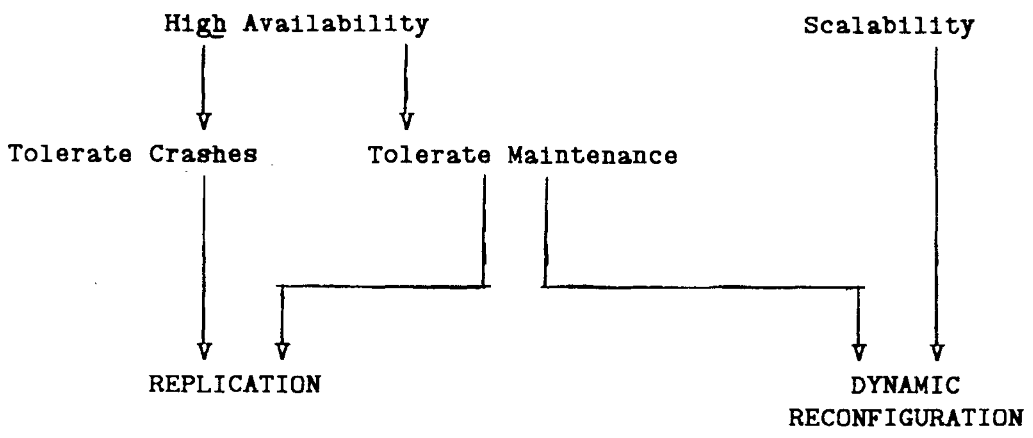

Figure 1-1: High Availability and Scalability Require Replication and Reconfiguration

for constructing distributed programs.

The example application is a repository for electronic mail. A mail[1] repository is essentially an electronic mail system without a friendly user interface. Personal computers (PCs) provide the user interface and editing capabilities required to view and compose messages. These PCs communicate with the repository over a high-speed network to send, receive, store, and retrieve messages.

The example application is related to ongoing work in distributed systems at MIT, particularly in the Argus group and at Project Athena. Argus is a language for building distributed programs [28]. Project Athena is a large distributed system being built to provide computing power for undergraduate education at MIT, as an educational experiment [2].

A centralized repository has been built and is in use. This thesis is concerned with a distributed version of this repository. In designing and implementing the system, there were several goals:

1. to present the same interface to PCs as the centralized repository;
2. to provide high availability in spite of reconfiguration and hardware failures;
3. to test the programming language Argus in an application designed to support a large

[1]Throughout this thesis, the term "mail" refers to *electronic mail*, or *electronic messages*.

user community, with tight constraints on availability;

4. to evaluate Argus for its utility in constructing highly-available and scalable distributed programs; and

5. to serve as a prototype for a distributed repository that could provide mail service to MIT's Project Athena.

The remainder of the thesis describes the background for the construction of a distributed mail repository, the problems of design and implementation, conclusions to be drawn, and directions for future work.

The next chapter summarizes the features of PCMail, the existing centralized mail repository. Chapter 3 introduces the relevant features of the programming language Argus, discusses the tradeoffs that were made in constructing a distributed repository, and describes the operation of the repository in simplified terms (ignoring replication and reconfiguration).

Chapter 4 presents the choice and implementation of replication techniques for a highly available distributed program. Only a few replication techniques can meet the constraints of the application. In addition, almost all previous work on replication has considered replication of files or simple objects. The mail repository requires replication of relatively complex objects (for example, tables that point to other tables, which in turn refer to the object of interest). Straightforward replication methods would lead to an expensive replicated call for each object reference; I describe a technique for compressing several such calls into a single remote call. Performance constraints require that the calls making up a replicated call be performed concurrently; I present techniques for avoiding deadlock among concurrent transactions.

Chapter 5 presents a new method for designing and implementing reconfigurable systems. The new techniques allow replacement of software components while preserving their state, but require no changes to the Argus system or language. This contrasts with previous work on guardian replacement in Argus.

Chapter 6 concludes the thesis by discussing lessons learned from the implementation, evaluating the ways in which Argus helped and hindered the construction of the system, and pointing out directions for future research.

Chapter Two

PCMail – the Centralized Mail Repository

Not of the letter, but of the spirit:
for the letter killeth, but the spirit giveth life.
—II Corinthians 3:6

Oui, cela était autrefois ainsi,
mais nous avons changé tout cela.
(Yes, it used to be that way,
but we have changed all that)
—Molière

This chapter briefly describes the rationale for a mail repository and the structure of a centralized implementation. First, I distinguish between the repository model and transport model of mail service, describe the division of the overall mail system into the repository and agents, and differentiate the repository from external mail systems. Then, I describe the structure of the repository and its communications protocols, and outline the rationale for a distributed implementation of a repository.

The presentation in this chapter is based on an existing centralized mail repository, called PCMail [7]. PCMail is not a contribution of this thesis; rather, PCMail provides a background and motivation for this thesis.

2.1 Repository vs. Transport Models

In a *repository* model of mail service, a central server holds a user's messages until the user explicitly deletes those messages. Messages read by the user are copies that do not affect the status of the repository. In a *transport* model of mail service, a central server only holds a user's messages until the user has had an opportunity to copy those messages into the user's personal computer (PC). After being read, the messages are no longer stored at the server — they have been "transported" to the user's PC. Grapevine [5] is an example of a mail system that was built as a mail transport service; the Grapevine servers delete a user's messages after they have been sent to a user's PC and acknowledged.

In a transport model, a retained message is stored only on the PC; in a repository model, it is also stored on the server. Thus, a repository service typically requires more storage at the server than a transport service. However, a repository service is typically more reliable and flexible than a transport service. A repository is more reliable because a catastrophe on a PC, such as a disk crash or the loss of a floppy disk, does not result in the loss of mail on the repository; and a repository is more flexible because a user can read mail from several different PCs without spreading the actual messages over those machines, thereby fragmenting the user's mail state.

PCMail is based on a repository model of mail service.

2.2 The Repository, Agents, and DMSP

PCMail is divided into two pieces: the *repository* and the *agents* (see Figure 2-1).

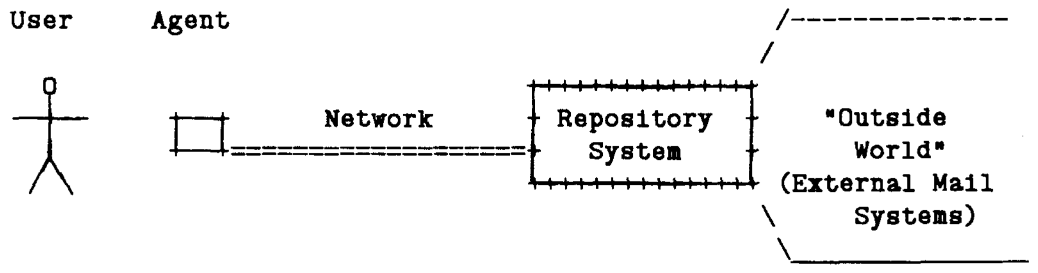

Figure 2-1:A User, an Agent, and the Repository

The repository holds all of the messages that have been delivered to a user but not yet discarded. The repository also holds any information required to deliver messages correctly. The information kept at the repository for a user is called that user's *global mail state*.

Agents[2] are personal computers that provide a user-friendly interface to the repository, allowing users to read and send mail. An agent may store information on its disk(s) to improve performance or to allow operation of the agent while not connected to the repository. Each agent's stored information serves as a cache, and is called its *local mail state*.

[2]The PCMail documentation uses the term *client* instead of *agent*. I prefer *agent*, since otherwise it is easy to get confused between a client of the system (a user) and a client of a server (an agent). Also, the term *agent* more clearly expresses the use of a personal computer as an agent of a user, communicating with the repository.

The protocol for communication between an agent and the repository is called the Distributed Mail System Protocol (DMSP) [7]. DMSP was defined as the interface between an agent and a centralized repository. If the distributed repository is to have the same interface to agents as the centralized repository, it must use DMSP without changes. Section 2.6 describes DMSP in more detail, and appendix A contains a complete specification of the protocol.

2.3 Repository and External Mail Systems

The repository is not only a system for sending mail among repository users, but also a system for sending and receiving mail between repository users and users of other mail systems. These other mail systems are collectively referred to as *external mail systems*. External mail systems communicate with the repository by means of a standard protocol, the Simple Mail Transfer Protocol (SMTP) [36]. SMTP is not used within the repository at all; it is used only as an interface to other mail systems. SMTP does not place any more constraints on the repository's operation than DMSP does, so I will not describe it further.

2.4 Rationale for PCMail

The design of PCMail is driven by three observations:

1. Personal computers are portable. This means that they are not always attached to a network, but still must be usable when not attached. For example, some personal computers could be used to read and compose mail while the user is travelling.

2. Personal computers are poor in resources. They rarely have more than one megabyte of memory and a few hundred kilobytes to a few megabytes of mass storage. In addition, they may have access only to relatively low-speed communications (such as a 300-baud modem link as opposed to a 10-megabit Ethernet connection).

3. Personal computers are relatively inexpensive. It is plausible that a person could own and use more than one (for example, have one at home and one in the office). Also, students may use different personal computers at different times.

Each of these observations leads to a feature of the design of PCMail. The local mail state allows operation when not connected to the repository (for example, when travelling) and helps to improve the response time of the system if the link to the repository is slow.

Since a personal computer may not be able to store the user's entire global mail state, there must be a shorthand way of representing the global mail state. In PCMail, each message has a *descriptor*, which is a short (about 100 bytes) summary of information about the message. Usually an agent will be able to store all of the descriptors for a mailbox, even if it cannot store all of the messages in that mailbox. Based on the information in the descriptors, the user can choose which messages to fetch into the local mail state if they will not all fit.

Synchronization is performed between an agent and the repository so that a user can see a consistent view in spite of using several PCs.

2.5 Naming

Each user has a unique *user_id*. A user can have any number of mailboxes. Each mailbox has exactly one *box_name*, which is unique among the mailboxes of its user, and zero or more *addresses*, each of which is unique within the entire repository. An address points to a single mailbox — addresses are not used for mailing lists. Mail delivery is based on addresses; mail retrieval is based on box_names. A user's agents each have exactly one *agent_id*, which is unique among the user's agents. The messages in a mailbox each have exactly one *msg_id*, which is unique in that mailbox.

Mail sent to an address is delivered to the mailbox with that address. A mailbox with more than one address receives mail sent to any of its addresses, and a mailbox with no address cannot receive mail (such a mailbox may be useful as an archive for old mail, to organize messages already received).

Figure 2-2 shows two users, "Bert" and "Ernie," each with two mailboxes. Bert has a mailbox named "main" and one named "junk". Ernie also has one named "main", and one named "more-mail". If someone sends mail to "Bert", then it goes to Bert's "main" mailbox, because that's where the address "Bert" points to. Similarly, mail to "Ernie" goes to Ernie's "main" mailbox. Ernie's "main" mailbox also receives mail sent to the addresses "net-help" and "bert-stuff". The addresses of mailboxes are not necessarily related to their names or their owners. "Bert" goes to Bert and "Ernie" to Ernie only because of a sensible administrator who set up addresses that way. Note, too, that Ernie's "more-mail" mailbox has no addresses, and cannot receive incoming mail. Ernie might use it to save previously-received mail.

A user_id and box_name together uniquely identify a single mailbox in the repository. Similarly,

User_id:	Bert		Ernie	
Box_name:	main	junk	main	more-mail
Address(es):	Bert	bert-junk	Ernie net-help bert-stuff	

Figure 2-2: Two Users, Each With Two Mailboxes

a user_id, box_name, and msg_id together uniquely identify a single message in the repository. When a user logs in to retrieve mail, that user's user_id is implicitly used as the user_id for all subsequent operations.

The repository stores information about changes for each [agent_id, box_name] pair of a user. For a given user_id, a [agent_id, box_name] pair identify the messages that have changed since the agent last examined the mailbox. A message can change only by

- arriving for the first time,
- having one or more flags change state, or
- being expunged (permanently removed).

2.6 DMSP

The Distributed Mail System Protocol (DMSP) [7] allows an agent to

- login and logout,
- fetch the descriptors that have changed since this agent last connected to the repository,
- fetch a specified range of descriptors by their msg_ids,
- fetch a particular message's text by its msg_id,
- set one of several flags on a message identified by msg_id

- expunge the messages that are flagged as "deleted",
- send a message,
- create, delete, and list mailboxes,
- create, delete, and list addresses for a particular mailbox,
- create, delete, and list agents usable by this particular user,

Appendix A provides a full definition of DMSP. Although not explicitly defined by the protocol, the following assumptions about reasonable mail semantics are built into the protocol and the existing centralized implementation:

- Messages that have arrived in a user's mailbox persist until deleted and expunged.
- Messages that have been expunged do not reappear.
- Users, agents, and mailboxes do not appear or disappear except when created or destroyed, respectively.
- The msg_id-to-message binding is immutable: a msg_id is never reused, and always refers to the same message.
- Within a mailbox, the msg_id's form a total ordering, in order of message arrival. There are no gaps in the sequence of assigned msg_id's.
- The effects of all write operations, such as setting flags on messages, are apparent immediately (on next read).
- Concurrent operations on a single user's mailbox are not allowed.

I make these assumptions explicit since they are a part of the interface that a distributed repository must present to agents, and they complicate the task of implementing a distributed repository. The syntax and semantics of DMSP define the interface that a distributed repository must present to agents. The system's behavior must not change radically, even if such changes would make the distributed repository more straightforward or more efficient.

2.7 Rationale for a Distributed Repository

The PCMail architecture has been realized in a system built at the MIT Laboratory for Computer Science. The repository is implemented in C and runs under 4.2 Berkeley Unix on a single VAX-11/750. Users with IBM PC/AT computers can connect to the repository machine in order to access mail. While this system is effective for the current user community, there are limits to its usefulness for a large community. It is not possible for more than a few dozen users to be connected at any one time, and mail service is unavailable whenever the single repository machine crashes or is taken down for maintenance.

When the demands of the community overwhelm a single machine, the community will require either a larger machine or multiple machines. At some point, a single large machine is simply a bad choice: it still represents a single point of failure (to an ever-larger number of users), and may be more expensive than multiple smaller machines [11].

Consider a situation in which the user community has grown to the point where two or more machines must be used. One way to use two or more machines is to run a centralized repository on each machine. With centralized repositories, the large community must be partitioned into several groups, each group using a different repository. However, this is a poor organization for a multiple-computer application. While failure of a single machine does not deny service to the whole community, any individual user is still served by only a single machine and sees no improvement in availability over the old repository. Further, the static allocation of users to particular machines means that some repository machines may be overloaded while others are lightly loaded. With a collection of centralized repositories, there is no way to reassign users dynamically to other machines. Finally, the partitioning of users is typically reflected in the addressing structure (with users addressed as, "user@machine8" rather than simply "user"), making the system look even less like a single large repository.[3] The capacity of computers forces the name space to be partitioned at a finer grain than might be otherwise desirable.

The problems of centralized repositories are avoided if the mail repository is a distributed program, instead of a collection of single-machine programs. Copies of respository information can be kept on more than one node so that the information can be read and written in spite of hardware failures and maintenance. Machines can be treated as a pool of processors, each having roughly equal access to resources. When a machine fails, a user's agent can connect to another machine; this may even be accomplished quickly enough that a user doesn't notice. Further, users need not be assigned to already-overloaded repositories in order to access information stored there. Finally, the repository provides a single large name space that supports a large user community. The division of the large name space into smaller, realizable name spaces is an

[3]Many organizations with multiple computers supporting a large user community manage to avoid this naming problem by building a small distributed database to do it. Each machine has a local database for finding the machine assocated with a particular user, and all machines have identical copies of the database. This is essentially an *ad hoc* replication technique, serving the same function as replication of registries in the distributed repository (see chapter 4).

implementation issue, not something that affects the interface of the repository.

The next chapter presents the design and implementation of a distributed repository.

Chapter Three

The Distributed Repository

Nothing must pass this line
Unless it is well-defined
You just have to be resigned
You're crashing by design.
—Pete Townshend

So near, and yet so far.
—Alfred, Lord Tennyson

This chapter presents a simplified view of the design and implementation of the distributed repository. The distributed repository is described as a distributed implementation of the centralized repository outlined in chapter 2. In this chapter I simply assume that components rarely fail; chapters 4 and 5 describe the replication and reconfiguration techniques that support this assumption.

I first describe the programming language Argus and the features it provides for building distributed programs. Then, I briefly sketch an architecture for a large distributed system like MIT's Project Athena and consider its implications for a distributed repository. Having presented the language to be used and the environment for the repository, I present the goals for the system and the tradeoffs chosen to achieve a good overall system. I present the overall architecture and data structures of the distributed repository, and give some examples of how the components of the system interact.

3.1 The Programming Language Argus

Argus is an integrated programming language and system designed and implemented by Barbara Liskov's group at the MIT Laboratory for Computer Science. Argus is intended to support distributed programs that need reliable, consistent storage but that do not have severe real-time constraints. The distributed repository is an example of such a distributed program, so the features of Argus should help in the construction of the distributed repository. Argus is fully described in [28, 29]; this section presents the features of Argus important to the distributed repository.

The key constructs of Argus are *guardians* and *actions*. Atomicity of actions is ensured by shared *atomic objects*. The following sections describe these in more detail.

3.1.1 Guardians

A guardian encapsulates a resource such as a hardware device or a database. Guardians are the unit of failure; they either function correctly or fail completely (in the latter case they *crash*). Guardians provide ***handlers***, which are objects like procedures that can access the resource encapsulated by the guardian. Handlers are invoked by remote procedure call, and arguments and results of handler calls are transmitted by value [18]. Like other objects, guardians have state; a guardian's state consists of a *volatile state* (cheap but lost in a crash) and a ***stable state*** (expensive but survives crashes). The guardian's stable state survives crashes with an arbitrarily high degree of certainty [26]. After a crash, a guardian recreates its volatile state from its stable state by executing *recovery code* defined by the guardian's implementor. A guardian executes on only a single *node*, but a node may be able to support more than one guardian executing at a time.

A guardian type provides one or more *creators*. A creator is a special kind of procedure that creates a new instance of a guardian and initializes its stable state. A guardian is an active object: unlike other objects, a guardian has multiple internal processes. Some of these processes are created to service calls to the guardian's handlers (a new process for each incoming call), while others may be created by the guardian to perform background tasks independent of the handler workload.

All Argus objects are local to a single guardian, except for guardians, nodes, creators, and handlers. Transmission of a local object to another guardian is performed by sending an abstract value that represents the object. Thus, the recipient of a transmitted object can mutate it without affecting the copy held by the the sender. Guardians and handlers are transmitted by reference: all "copies" of the guardian or handler refer to the same object. A change to the guardian (for example, mutating its state or destroying the guardian) can affect all of the clients of the guardian or handler.

Guardians have separate address spaces: it is impossible for a guardian to have a reference to an object local to another guardian. Guardians are relatively large and expensive objects, so they are used to encapsulate collections of objects rather than single objects. A reference to an object at another guardian must consist of a reference to the remote guardian and some tag to distinguish among the objects at that guardian. Thus, data structures that are intended to span guardians may be more complicated than data structures contained entirely within a single guardian.

3.1.2 Actions

Actions help to mask the effects of concurrency and failures. Actions can help in this way because their execution is *atomic*. Atomicity is composed of two properties, indivisibility and recoverability.

Actions are ***indivisible***, meaning that the execution of one action never appears to overlap or contain the execution of another. Since Argus actually permits concurrent execution of actions, such executions are ***serializable***: that is, the concurrent execution of a set of actions is equivalent to some serial execution of the actions [14].

Actions are also ***recoverable***, meaning that the overall effect of the action is all-or-nothing: either all of an action's effects take place, or none do. Actions either *commit* or *abort*. When an action commits, its changes are made permanent, and become accessible to other actions; when an action aborts, all its changes are undone.

Actions in the Argus system can be ***nested***: an action may have ***subactions***, which may perform some computation and commit or abort without committing or aborting the parent action. An abort by a parent action undoes all the changes performed by any subactions, even those that committed. A guardian's stable state is only modified when a ***topaction*** (a top-level action — the root of a tree of nested actions) commits [34].

3.1.3 Atomic Objects

In Argus, atomicity is provided by the objects shared by actions. Immutable objects are always atomic (no action can see changes made by another action since the object never changes). To provide atomicity for mutable objects, each such shared object must provide its own concurrency control (by locking) and recovery (by some scheme of keeping versions). In addition, each such object must allow for resilience through logging data to stable storage. Providing appropriate locking and recovery for objects is expensive, so Argus distinguishes between *atomic objects* and *non-atomic objects*. Actions are guaranteed to be atomic only when all the objects shared by those actions are atomic objects. Argus provides several *built-in atomic types*. These provide atomicity through a read/write locking scheme and versions.

In read/write locking, several actions can concurrently read an object's value, but an action intending to modify the object must ensure that it has exclusive access to the object. An action modifying an atomic object actually modifies a copy of the object's state, called the *current version*. If the modifying action commits, the current version becomes the object's new state; if the modifying action aborts, the current version is discarded and the old state (the *base version*) remains the state of the object. Actions hold locks until commit or abort (strict two-phase locking) to avoid *cascading aborts*, where the abort of an action forces the abort of other actions. A subaction's locks are not released if the subaction commits: instead, the locks are inherited by the parent action [34].

The operations of some abstract objects allow more concurrency than can be achieved with simple read/write locking. For example, a mailbox might allow one action to deliver a new message while an old message is being read by another action, rather than requiring the delivering action to obtain a write lock on the mailbox. Since read/write locking disallows this kind of concurrency, the built-in atomic types can limit the performance of the system. To circumvent this problem, Argus allows the construction of *user-defined atomic types* that allow greater concurrency than the built-in atomic types. [44]

3.2 The Environment of the Repository

The distributed repository is planned for a distributed system similar to MIT's Project Athena. The network architecture is composed of a *spine network* and numerous *local clusters*. Each local cluster consists of client workstations and servers, connected by some local-area network. Each local cluster is then connected to the spine. The spine carries only inter-cluster traffic: there are no clients or servers connected directly to the spine. The entire system is fairly compact, since MIT does not have a large campus. The cost of a remote call using the local-area network(s) is high compared to the cost of a local call on the same machine, but the difference between an intra-cluster call and an inter-cluster call is comparatively small. Thus, there is no important sense in which one remote machine is "closer" than another.

Many of the workstations will be in public areas, available to all who wish to use them. In particular, students will often use different workstations at different times. Students with personal computers will either hook them directly to available ports in local clusters or call up available workstations by phone lines.

3.3 Building a Distributed Mail Repository

Argus is the main tool for building a distributed repository, and Project Athena forms an environment for the operation of such a repository. I now consider the design goals for a distributed mail repository and discuss choices to be made in designing and implementing the repository.

3.3.1 Design Goals

The set of services to be provided by the mail repository is specified by the protocols that it must support: DMSP [7] for communicating with agents, and SMTP [36] for exchanging mail with external mail systems. There are a number of higher-level goals to be considered in the design and implementation of these services. This section describes the design goals for the distributed-repository.

First, a *mail transaction* is defined as one of the following:

- a single synchronization of mail state,
- a single mail message fetched, or
- a single mail message sent.

The repository must be ***highly available***, meaning that a given mail transaction has a very high probability of being able to run to completion in spite of failures within the repository ($P_{success}$ is close to 1).

The repository must be *scalable*, meaning that it can gracefully grow or shrink by adding or removing hardware and software resources. The repository should serve communities of up to about 20,000 users — large enough to support the entire MIT community[4].

[4]MIT has roughly 9000 students, 1000 faculty, 2000 professional and research staff, and 5000 support and supervisory staff.

The repository must be *efficient*: its functionality and speed must be comparable to the centralized repository implementation (PCMail) to attract users. In addition, it must not require excessive processor power or storage, or it will not be attractive to support the system for large communities of users.

The repository should present a *consistent* view of mail state to its users. Users should not be aware of whether the repository is implemented as a centralized or distributed system, except for the higher availability of a distributed system. Therefore, the repository should not allow anomalies to arise from crashes or network partitions. This contrasts with a system like Grapevine [5, 40]. In Grapevine, updates are made at a single location and propagated through the system. This means that it is possible for the same name to be added (with two different bindings) at two different sites. The addition of both names succeeds, but subsequently one will overwrite the other (based on a total ordering of creation times in the system). This may come as a surprise to the "loser," especially if that person is unaware of Grapevine's implementation as a distributed system.

The repository should continue to operate *reliably* in spite of node crashes and communication failures. In particular, the repository should neither lose a message once the message has been accepted for delivery, nor lose a message from a user's mailbox after it has been delivered. An acknowledgement of a (DMSP or SMTP) request must mean that the appropriate changes have been incorporated into the repository's stable state. This reliability must be provided in the face of anticipated hardware or software changes (maintenance), and unanticipated hardware failures (crashes). The repository is not intended to tolerate unanticipated software failures, design failures, or operation failures (for example, software errors or errors by the managers and operators of the system).

3.3.2 Tradeoffs

Having enumerated the design goals for the distributed repository, I now consider what compromises are necessary.

To achieve ***high availability***, the components of the system must be replicated, so that the failure of a single node does not make a needed component unavailable. In addition, the components must be constructed so that they can be replaced without making the system unavailable. Replication can improve the reliability of the system, since the permanent loss of a replica does not necessarily mean that the data is permanently lost. Unfortunately, replication reduces the scalability and efficiency of the system because users require more resources than they do in a non-replicated system. In particular, replication of large data structures like users' mailboxes may be uneconomical.

To achieve *scalability*, the system must be constructed from modular components that can be created, destroyed, and rearranged to meet changing needs and hardware configurations. To avoid running out of space or time at a node, no physical data structure can be allowed to grow as fast as the user community; it must be possible to partition all such data structures into smaller entities that can be distributed across nodes. Since systems can shrink as well as grow, it is clear that the data structures and algorithms required to support a large community must not have excessive overhead when used to support a small user community. Given components designed for scalability, replication may be somewhat easier since there are identifiable modules that may be replicated. Efficiency may be degraded if there are too many modules or they are scattered too far apart, causing common computations to require many expensive remote calls.

Efficiency requires minimizing the amounts of storage, computation, and communication required. The tradeoff between space and time is well known: a function can usually be made faster at the expense of storage, and vice versa. Since communication between nodes is expensive compared to computation (even on a local-area network), efficiency is improved by *caching*. A cache stores the results of expensive operations in order to save the cost of recomputing those results. Common operations should be optimized at the expense of less common operations.

Full *consistency*, so that the distributed system behaves exactly like a centralized system, may require that the system be unavailable rather than give out inconsistent information. A distributed system that presents functionality exactly equivalent to a centralized system may not be more available than the centralized system, and may even be *less* available, since it is exposed to more possibilities for hardware failure. The challenge in designing the mail system is to trade off consistency for performance: to weaken the user view as little as possible in order to gain as much availability and efficiency as necessary [30].

For *reliability*, Argus (see section 3.1) provides stable storage and atomic actions that allow a distributed program to function reliably in spite of node crashes and communication failures. Reliability helps achieve consistency, availability, scalability, and manageability by providing a simpler model on which all of those can build. However, the mechanisms provided by Argus to achieve reliability involve a certain amount of overhead that reduces the efficiency of the system.

The remainder of this chapter and chapters 4 and 5 describe the system and its actual tradeoffs in full. First, I summarize the major tradeoffs involved. In the distributed repository, multiple guardians are used for availability and scalability. Use of multiple guardians means that expensive remote calls are necessary. To maximize efficiency, the repository is designed to require only one remote call for each user request wherever possible. If more than one remote call is required for a request, the multiple calls are performed concurrently if possible to increase efficiency. The built-in actions and atomic types of Argus are used in the implementation for a more reliable and simpler system. The system's workload is made of short actions to increase concurrency, and some user-defined atomic types are also used to gain further concurrency. The distributed repository has a weaker view of consistency than the centralized repository, especially in the presence of partitions: this weaker view allows the distributed repository the flexibility to behave either exactly like the centralized repository or in a somewhat inconsistent way for higher availability. This inconsistency appears only during a partition of the underlying network, and only for users who move from one side of the partition to the other. To such users, the system will seem to be unreliable since previous work has been "lost".

The next section discusses the functions of the repository and how these are divided into different guardian types. It also explains the interaction of the components, demonstrating the choices and tradeoffs made.

3.4 The Design of the Repository

The repository stores messages, provides a DMSP interface for agents, provides an SMTP interface for external mail systems, and names objects (messages, mailboxes, agents, addresses). The repository design uses Argus guardians to modularize the repository's functions, one type of guardian for each of these services. This modularization improves scalability at the cost of efficiency. Scalability is improved because guardians can be created or destroyed independently of each other. Unfortunately, some efficiency is lost since guardians must communicate by remote procedure call protocols, even if they are executing on the same node. Whenever possible, information is cached to avoid unnecessary remote calls.

Each of the repository functions is realized as a separate type of guardian:

- a *mailsite* stores messages;
- a *frontend* presents a DMSP interface to agents;

- an *alien* presents an SMTP interface to other mail systems; and
- a *registry* provides naming.

See figure 3-1.

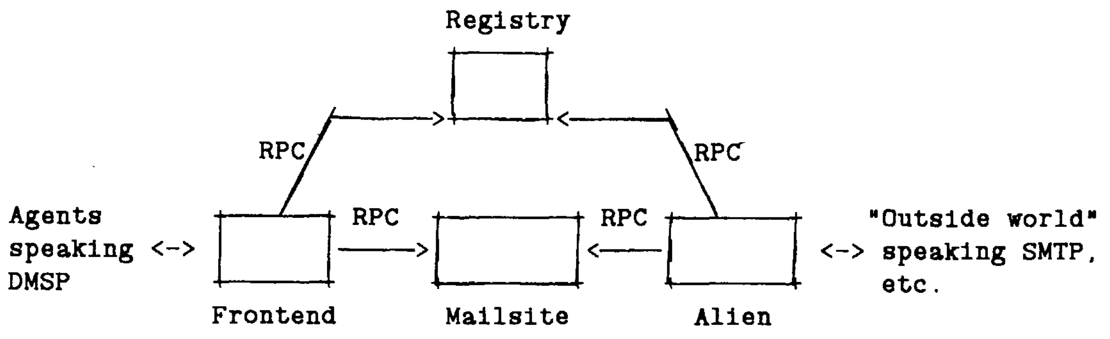

Figure 3-1: The Guardians of a Distributed Repository

To accomodate heavy usage efficiently, a distributed repository for a large community may include more than one of each of these guardians. However, this use of multiple guardians is not a replication of resources for availability. In this chapter, I assume that guardians do not crash; Chapter 4 relaxes that assumption. In this chapter, I use multiple guardians only to divide up work for greater efficiency. The remainder of this section considers each of these guardian types in more detail.

3.4.1 Mailsite

A mailsite actually stores messages in mailboxes for users. A mailsite contains a number of mailboxes, each of which is identified by a unique box_id. Each of these mailboxes may contain messages, identified by msg_id. The msg_id of a newly-arrived message is assigned by the mailbox, to ensure that it is the next msg_id in sequence.

Each mailbox keeps track of messages that have changed since each agent last examined the mailbox. This information is used to synchronize the mail state at an agent with the mail state at the repository. Each mailbox contains a collection of *update vectors*, one for each agent_id. An update vector is an ordered list of msg_ids; these msg_ids correspond to the messages in the given mailbox that have changed state since the last time the agent reset the update vector. Recall that a message can change only by arriving for the first time, having one or more flags change state, or being expunged.

When a message arrives for the first time, the msg_id is added to the update vector of every agent belonging to the user. When a flag is changed or the message is expunged, the msg_id is added to the update vector of every agent except the agent making the change.

The state of a mailsite is stable. There may be one or more mailsites in the system, and each mailsite may hold an arbitrary collection of mailboxes. There is no requirement that all of a user's mailboxes be at the same mailsite.

3.4.2 Frontend

A frontend listens on a well-known port for DMSP connections from agents, forking a process to service each new connection[5]. For each DMSP command sent by an agent, the frontend interprets the command and calls other guardians as needed to perform the DMSP action requested. The frontend is an interface between the non-atomic world of network protocols and the atomic world of Argus remote procedure calls. Each DMSP operation is processed as a separate topaction, and that topaction must commit or abort before any status information is sent back to the agent. A frontend caches information about a user's global mail state and registry information to speed up DMSP operations. For example, a frontend may cache information about which registry contains which names, or it may cache new messages from a user's mailbox on the assumption that the agent will ask for them. However, no harm is done if this cached information is not needed or if the frontend crashes and loses the information. Thus, all of the cached information is volatile. Unlike the other guardian types, a frontend has no stable state, and may be created or destroyed easily since there is no stable state to initialize or preserve.

A frontend provides no creators or handlers other than a simple creator that creates a new frontend.

3.4.3 Alien

An alien mailer (hereafter, "alien") listens on a well-known port for SMTP connections from external mail systems, forking a process to service each new connection. The alien interprets the

[5]For those unfamiliar with well-known ports and connections, the principle is much the same as used in Citizen's Band (CB) radio. In CB, parties talk on a specific initial channel long enough to agree on which other channel to use for their conversation, then switch to that channel. Correspondingly, the frontend has a process which listens on a fixed, agreed-on port long enough to establish an agreement with the caller as to the new port to use for the connection. Then the frontend forks a process to service the requests coming from the new connection.

addresses provided with a message and constructs a message object from the information sent on the SMTP connection. Since the repository is a final destination for mail, not an internet router, the alien does not accept mail for users not listed in the registry. Like the frontend, the alien is an interface between the non-atomic world of network protocols and the atomic world of Argus remote procedure calls. Unlike the frontend, incoming commands are not processed immediately; instead, after the addressees of a message have been looked up in the registry, the message is placed in a stable queue to be delivered by a delivery process that runs periodically. Successful delivery of a message removes it from the stable queue. In addition to this incoming SMTP service, an alien also provides an outgoing SMTP service when a user of the repository sends a message to a user outside the repository. Again, this service consists of simply placing the message in a stable queue from which messages are delivered periodically. To avoid the complications of routing internet mail, the current implementation of an alien provides outgoing SMTP service by simply delivering outgoing messages to the local Unix sendmail daemon, which then forwards the messages to their final destinations.

3.4.4 Registry

A registry provides a mechanism for naming users, mailboxes, and agents. It is used by frontends and aliens to find the appropriate mailsites for incoming mail; frontends also use the registry to find mailboxes from which mail should be fetched. For a large community, the database required for naming will be quite large. Keeping all of the naming information in a single registry guardian may lead to performance problems. Thus, it must be possible to divide registry data among two or more guardians. A registry guardian will contain a portion of the database (its *local database*) and a *registry_guardian_table* mapping other portions of the database to other guardians. The registry_guardian_tables are identical across all registries: all registries agree on which registries contain which names. When asked for information about a name in its local database, a registry guardian will return the requested information; when asked for information about a name outside its local database, it will look in its registry_guardian_table and return the appropriate guardian, so that the client can call there instead. Clients of the registry — frontends and aliens — can cache the registry_guardian_table as a hint, and update it when a request to a registry fails because it doesn't hold the information for that name.

The database is divided into local databases using a scheme that requires only a small amount of information for the registry guardian table. Since the user_ids of the system have a natural ordering (they are character strings), the database is divided into ranges of user_ids. For each range, only the endpoints of the range need to be stored in the registry guardian table. To change the division of data between registries, a transaction must be executed that updates the tables at all of the registries. Alternately, the necessary information could be propagated lazily between nodes. Such redivisions of the data should occur very infrequently.

Using the registry_guardian_tables, the names in the system are divided among registries without affecting the naming structure. This division is superior to the scheme for names used in Grapevine (RNames). Grapevine RNames have two parts: the second part explicitly names a Grapevine registry and the first part is the name within that registry. For example, a Grapevine RName like "MDay.lcs" indicates that the "lcs" registry contains the name "MDay". This is a simpler scheme as far as the system is concerned, but is a poor choice for the users of the system. Grapevine RNames force users to be concerned with the internal structure of the system and force a user's name to change when moved from one registry to another.

The repository's scheme of dividing registry information does not allow individual reassignment of a name to a particular registry; for example, it is impossible to move a particular user's names into a particular registry guardian that is "closer" to that user's location. However, recall from section 3.2 that the notion of "closer" has little relevance for a system like Project Athena, and that most students will have no particular fixed location for receiving mail. Thus, the repository's method for dividing registry information should be effective.

A registry contains a table mapping user_ids to various information for a user and a table mapping addresses to mailboxes. The information for a user includes a list of agent_ids and a list of box_names, and each such box_name also maps to a *mailbox_interface*. Clients treat a mailbox_interface as if it were a container for the messages in the mailbox, but messages are actually stored at mailsite guardians. The implementation of a mailbox_interface object turns operations on the "mailbox" into remote operations on the mailsite that stores the actual mail state. This hiding of the mailsite is important for replication, to be discussed in chapter 4.

3.5 Scenarios

This section outlines how the distributed repository services requests from agents. The scenarios are intended to give a flavor of how the repository works, and provide examples to clarify later discussions of replication and reconfiguration.

There are three basic scenarios that occur most often in the repository. These are:

- *sending mail* — an agent presents the repository with a message to be delivered. Some or all of the addressees may be outside the repository.
- *synchronizing mail state* — an agent synchronizes its state with the global mail state, fetching changed descriptors and new messages, altering states of flags and update vectors to reflect the synchronization. This is how a user reads new mail from the repository.
- *delivering external mail* — an external mail system contacts the repository with a message to be delivered to one or more repository addresses.

The following sections discuss the repository's behavior for each of these scenarios.

3.5.1 Sending Mail

1. An agent connected to a frontend (already logged in) sends a DMSP SEND-MESSAGE command, including the text of the message in network standard format. The standard format is described in RFC-822 [8].
2. The frontend receives the command and message.
3. The frontend checks the "From:" field of the message against the identity of the authenticated user. If the "From:" field is incorrect, the frontend sends a FAILURE message to the agent.
4. The frontend parses the recipients from the "To:" field of the message.
5. As a single topaction, the frontend does the following for each recipient of the message:
    - The frontend checks whether the address is a local repository address or an external address. An external address is syntactically distinct from a repository address. The current repository treats all addresses containing '@' (as in "mday@xx.lcs.mit.edu") as external addresses, and treats all other addresses as local addresses.
    - If the recipient is a repository address, the frontend looks up the destination mailbox at the appropriate registry and puts the message in the mailbox. Delivery of a message to a mailbox changes the update vectors of that mailbox, as described in section 3.4.1. If any failure occurs in this process (for example, the address doesn't exist, or the mailbox is inaccessible) the frontend aborts the topaction and sends a failure message to the agent.
    - Otherwise (when the recipient is an external address), the frontend sends a copy of the message to an alien, where it is queued up to be sent out.

   Aborting the topaction on failure of delivery is a safe course of action because the agent should still have a copy of the message to be sent until it receives an OK from the frontend.

6. If the topaction committed, the frontend sends an OK message to the agent.

3.5.2 Synchronizing Mail State

1. An agent connected to a frontend (already logged in) sends a DMSP LIST-MAILBOXES command.

2. The frontend starts a topaction and asks a registry for a list of mailboxes. Then it gets information from each of these mailboxes (actually querying the appropriate mailsite) about

   - the next available message identifier in that mailbox,

   - the number of messages in the mailbox, and

   - the number of unseen messages in the mailbox.

   After committing the topaction, the frontend assembles the received information to send back a DMSP MAILBOX-LIST reply. If the topaction aborted, a FAILURE reply is sent.

3. The agent corrects its local mailbox list to match the new information.

4. The agent fetches a collection of changed descriptors:

   a. The agent sends a DMSP FETCH-N-CHANGED-DESCS command, specifying a mailbox and an upper bound *ub* on the number of changed descriptors to send.

   b. The frontend starts a topaction and fetches the first *ub* changed descriptors from the mailbox in increasing msg_id order (or all the changed descriptors, if there are fewer than *ub*). After committing the topaction, the frontend returns a DMSP DESCRIPTOR-LIST reply containing this information. If the topaction aborted, a FAILURE reply is sent.

   c. The agent updates its local descriptor list to match the new information, and sends a DMSP RESET-N-DESCS command to inform the repository that it successfully received the last set of changed descriptors.

   d. The frontend starts a topaction and resets *n* descriptors of the mailbox, so that the same agent issuing a DMSP FETCH-N-CHANGED-DESCS would not get the same descriptors. It then commits the topaction and sends an OK reply. If the topaction aborted, a FAILURE reply is sent instead.

5. Steps 4.a-4.d may be repeated for all of the mailboxes owned by the user. The process of fetching descriptors may be interleaved with the process of fetching messages, described by steps 6.a-6.d.

6. To fetch the text of a message (new or old):

    a. The agent sends a DMSP FETCH-MESSAGE command, specifying the mailbox and the message identifier within that mailbox.

    b. The frontend starts a topaction and fetches the message from the mailbox. After committing the topaction, the frontend sends a DMSP MESSAGE reply containing the text of the message. If the message does not exist or has been expunged, or the topaction aborted, the frontend responds with a FAILURE reply.

    c. If the message's flags indicate the message is "unseen", the agent will typically issue a SET-MESSAGE-FLAG command to indicate that it has been "seen".

    d. The frontend will carry out such a command by starting a topaction and setting the specified flag on the message in the mailbox. If the topaction commits, the frontend responds with an OK; if the topaction aborts, the frontend responds with a FAILURE message.

3.5.3 Delivering External Mail

1. An external mail system opens an SMTP [36] connection to an alien.

2. The alien behaves like any other SMTP server. The external mail system SMTP provides a list of destination addresses, and the alien checks each at a registry. The alien refuses mail for any address not listed at some registry of the repository.

3. When the message has been received by the repository, the alien starts a topaction and puts the message in a stable queue. Then it commits the topaction and sends a positive acknowledgement to the external mail system.

4. Periodically, the alien starts a single topaction and tries to deliver all of the messages in the stable queue. For each, it looks up the address at a registry to find the mailbox, then stores the message there. Storing the message in the destination mailbox changes the update vectors at the destination mailbox as described in section 3.4.1. If a problem occurs in delivery, the undelivered message is put into another queue. The alien will try to deliver a message a reasonable number of times and will then return it to the address shown in the message's "From:" field. If the message can't be delivered to this address, the alien will send it to a "dead-letter office" inside the repository that is read by system maintainers.

3.6 Action Structure

All of the topactions in the system originate in a frontend or an alien, which also serve as interfaces to the world outside the repository.

There is no explicit "commit" or "abort" in the interface of the repository, so actions are not visible to a user. DMSP was originally defined without any notion of grouping a sequence of commands into an atomic action. While an Argus implementation could provide such features, it would be a poor approach to the problem since users could then hold locks for an arbitrary length of time. The design of the current system ensures that locks are held only long enough to provide an answer to a single user request. Since the system's work load is composed of these short actions, there is no need to invent special rules for breaking locks when they have been held too long [15, 41].

The use of transactions considerably simplifies the implementation of the repository. For example, a failure in the delivery of a message to a repository address aborts the topaction performing the delivery; a message cannot be delivered to the mailboxes of some recipients but not others. The user can correct the problem causing the failure and resubmit the message to be delivered.

The typical request/response structure, apparent from the scenarios, is that a request is decoded outside the action system, and then a topaction is started. All remote calls necessary for the request take place within this topaction. After the topaction commits, an OK or the requested data is sent. If something goes wrong or the topaction aborts, a FAILURE message is sent. A positive acknowledgment sent to an agent or an external mail system means that the request has actually been successfully and permanently completed, so the repository cannot send an OK within the topaction in case the topaction subsequently aborts.

If a topaction commits and the frontend crashes before it can send an OK to the agent, the agent will probably retry the request. This can be undesirable, since it may mean that a message is delivered more than once. It is possible to eliminate the problem of retrying, but it would require extensive modification of DMSP. The problem of retrying occurs only very rarely, and in electronic mail it is preferable to deliver too many messages than to lose a message, so the repository simply delivers multiple messages in such a case.

3.7 Summary

The distributed repository is composed of a collection of guardians of four different types: frontends, registries, mailsites, and aliens. Frontends and aliens serve as interfaces between the repository and the outside world, while registries and mailsites store most of the state information of the repository. This chapter presented the reasons for the division of the repository into these guardian types and illustrated their interactions for typical agent requests. The next chapters consider how these guardians can be replicated for higher availability and how the distributed repository can be reconfigured by altering the number or kind of guardians it contains.

Chapter Four

Replication

We shall not flag or fail.
—Winston Churchill

It's no big disgrace
There's no loss of grace
The trouble is, there's always
Someone there to take your place.
—The Bangles

The mail repository outlined in Chapter 3 operates in much the same way as a centralized repository, except that the modularization of functions into guardians allows more flexibility in distributing the system. With four separate types of guardians, there is a spectrum of distribution possibilities, ranging from all four guardians running on a single node at one extreme, to each guardian running on a separate node at the other extreme.

If all of the guardians are running on a single node, the functionality provided is equivalent to a centralized repository. As components are distributed, the overall availability of the system declines, since failure of any single node may render the system unavailable. However, if the repository's information is replicated so that copies are stored on more than one node, the information can remain accessible even if some nodes in the system fail.

4.1 Replication of Services and Replication of State

There are two different classes of replication techniques, both of which appear in the distributed repository. The simpler form of replication involves guardians that provide a stateless service. For example, a PC can connect to a frontend and use it to carry out commands in the system. If that frontend fails, the PC establishes a connection to another frontend and continues. There is no state in the frontend that must be maintained across connections. Similarly, an alien receives mail from external mail systems and sends to external mail systems. It does not provide any way for clients to check the status of their mail. Abstractly, it is a stateless pipe for delivering messages, even though its implementation involves stable queues of messages. If an alien fails, its client can try to deliver the message to another alien. Since they provide stateless services, frontends and aliens can be replicated as necessary in order to provide more external interfaces for the system.

The more difficult form of replication is replication of state information. Both registries and mailsites contain mutable state, and this state must have some consistent abstract interpretation across the multiple physical copies. Registries contain the bindings for various names in the system, such as the names of users, mailboxes, and agents. Since the registry information stored per user is fairly small, it is reasonable to replicate all registry information. Replication of registry information is then a public benefit — it benefits not only the user(s) whose information is replicated, but also others who are trying to locate the user(s).

Mailsites contain mailboxes, simple containers of messages. A user's mailboxes tend to be much larger than the information kept in a registry for that user. Rather than replicating entire mailsites (in the way that an entire registry can be replicated), only a few mailboxes will be replicated. Replication of a mailbox represents a mostly private benefit to the owner of the mailbox.

This chapter discusses the techniques used to replicate registry information and mailbox information. The first part of the chapter describes the requirements for replication of registry and mailbox information. Then follows a presentation of two published methods that best meet the requirements of the repository. Finally, one of the replication schemes is chosen to be used in the repository, and its implementation is described. Since an Argus remote call is very expensive compared to local computation, the discussion emphasizes the aspects of the implementation that reduce the amount of time spent on remote calls.

4.2 Requirements

An algorithm for replicating objects must

- maintain consistency,
- tolerate partitions,
- perform well, and

- be reasonable to implement in Argus.

4.2.1 Consistency

All replication algorithms are intended to preserve some sort of consistency of the replicated data in spite of failures and concurrency. Our standard of consistency will be *one-copy serializability*: multiple concurrent transactions executing on replicated data must be equivalent to some serial execution of those transactions on a single copy of that data [3].

Some system implementors have argued that the transaction mechanisms required for one-copy serializability are too expensive, and have sacrificed one-copy serializability to increase performance [5]. However, Argus provides a computational model that already includes an efficient implementation of transactions and stable storage, so it seems sensible to start with an implementation that provides one-copy serializability, weakening this standard if necessary for performance.

4.2.2 Partitions

In any distributed system, there are two kinds of failures: crashes and partitions. Recall that in Argus, a crash means that the guardian simply ceases to function. In a partition, a guardian continues to function, but communications have failed so that it is isolated from some other guardians. In the "Athena-like" network (see section 3.2), the failure of gateways or repeaters may occasionally isolate local clusters from the spine network, producing a partition. Based on experience with existing networks, it is reasonable to assume that such partitions will occur infrequently and will be repaired quickly.

If the communication links to a node have failed, other nodes cannot get a response from it; thus, that node appears to have crashed. It is impossible to distinguish reliably between a situation in which one or more nodes have crashed and a situation in which those nodes are operational but are isolated by communication failures. It is possible to increase the connectivity of communications so that partitions are very unlikely. However, it is good engineering practice to tolerate partitions if they should occur, rather than simply asserting that they will never occur.

As discussed in Chapter 2, a user of the repository has certain "common-sense" expectations for the repository's behavior. Messages that have arrived in a mailbox must persist until deleted and expunged, expunged messages must not reappear, and users, mailboxes, and agents must not appear or disappear except when created or deleted, respectively.

There are several ways in which partitions can violate common-sense expectations of what is to occur. The first group of problems arises from a partition separating the frontend from registries. If the frontend cannot contact any appropriate registry at all, the agent cannot find any other resources (personal mailboxes, other users' mailboxes, lists of agents, lists of addresses, and other information). To reduce the likelihood of this sort of failure, registries must be replicated. Then a problem arises if the frontend cannot contact "enough" registries (where "enough" is determined by the replication protocol being used): the user may be able to access information stored in the accessible registries, but that information may not be accurate. Because registry information is fundamental to the integrity of the repository, operations can only be carried out if they can access "enough" registries.

The second group of partition problems arise when the partition separates the frontend from mailsites. If the frontend cannot contact the user's mailsite, the user cannot read new mail or delete mail held by the system. However, the mail that is cached at the user's agent PC is still available for reading and deleting. Changes made to the agent's local mail state can be reflected in the global mail state as soon as a connection can be established to the user's mailsite. If the frontend cannot contact other users' mailsites (to deliver mail), then mail will be queued up for those mailsites. For those users who must have more reliable access to their mail, mailboxes can be replicated on several mailsites. Then, if the frontend cannot contact "enough" mailsites (again, "enough" depends on the replication protocol used), the user can read information stored in the accessible mailsites, but it might not be accurate. Thus, new messages might not appear, or old deleted messages might reappear. These problems are acceptable if accompanied by a warning about the possible inconsistencies.

Since the system's users are mobile (they can move from one side of a partition to another), they can see different views during a partition. The repository must avoid any situation in which message 32 exists on one side of the partition but not on the other. Any such situation that arises without a warning to users will be seen as evidence of unreliability or erratic behavior.

Tolerating a partition therefore means the following:

- The system's service may be degraded, but the repository will continue to provide as much service as possible to as many users as possible (the repository does not simply shut down during a partition).

- During a partition, users are presented with warnings before seeing data that may be inconsistent. Such warnings must not depend on detecting the partition (recall that it is impossible to reliably distinguish between partitions and crashes).

• Operation during a partition does not lead to inconsistencies after the partition ends.

This definition of tolerating a partition allows a reasonable compromise between availability and consistency. Replication for availability never violates the logically centralized model of the repository without providing a warning. The repository appears to continue operation normally if the partition does not affect the user of interest. The repository gives warnings if the mailbox information being presented might be inconsistent, and it "goes down" if it is not possible to use registry information consistently. In an agent-frontend protocol without provision for warnings, such as DMSP, service may be denied whenever a warning would be provided, or service may be allowed to continue. Denial of service matches the behavior of the centralized repository; allowing some inconsistency allows higher availability. The administrators of a distributed repository may choose which model of service to offer.

4.2.3 Performance

If replication is to be useful in the repository, it must not cost much in terms of performance. In the current Argus implementation, a remote call between guardians on the same node is only slightly faster than a remote call between guardians on different nodes. Recall from section 3.2 that there is no real notion of one remote machine being "closer" than another. All remote calls can be considered to have comparable costs, and optimizations for calling "nearby" guardians can be ignored.

4.2.4 Implementability

Some algorithms are straightforward to implement, while others are quite complex and obscure. Similarly, some algorithms have a simple interface to the underlying transaction system, while others must interact extensively with the underlying system in order to keep track of various system resources and states. Since Argus provides an underlying transaction system based on locks, with little accessibility to the implementation of that system, simple schemes that do not require interaction with the transaction system will be much more easily implemented than complex schemes that require such interactions.

4.3 Choosing a Replication Algorithm

In this section, I examine published algorithms for replication of data. I quickly discard any that do not tolerate partitions and any that would be difficult or impossible to implement in Argus. The well-known algorithms that I do not consider further include Primary Copy [1], Available Copies [4], the ISIS replication strategy [24], regeneration [37], early quorum consensus algorithms [23, 43], True-Copy Token [33], the algorithms for distributed INGRES [42], Missing Writes [10], and Virtual Partitions [12].

General Quorum Consensus [19, 20] is probably implementable in Argus with a great deal of work and ingenuity, and can tolerate partitions. However, GQC uses timestamps and logs instead of locks and versions, and a straightforward implementation in Argus would probably be unacceptably slow. Integrating concurrency control for performance (as discussed in [19]) would require a language other than Argus.

LOCUS version vectors [35] can re-establish consistency after a partition is repaired, but cannot provide a warning of possible inconsistency during a partition. Thus, a version vector scheme does not allow administrators of the distributed repository to faithfully emulate a centralized repository at the expense of availability. Instead, version vectors permit behavior that is impossible with a centralized repository. If the distributed repository did not need to emulate a centralized repository, version vectors might be of interest.

Clearly the most interesting replication algorithms are the ones that do the right job and can be implemented in Argus. The algorithms considered are Gifford's Weighted Voting algorithm [16], and the Daniels and Spector algorithm for replicated directories [9].

4.3.1 Weighted Voting

Gifford's weighted voting [16] provides a general method of building replicated file suites. A certain number of *votes* are divided among the physical copies of a replicated file, and operations on the file are required to acquire a specified quorum of these votes. Each physical copy of a file contains three things: the state of that data file, the version number of that state, and the number of votes for that copy of the file. There are valid reasons for weighting individual copies differently, by assigning them more or less votes; this flexibility is an important advantage of voting. However, I assume for simplicity that every physical copy of a file has a single vote.

The version numbers are used to determine which copy contains the result of the most recent write. A read operation must read at least r copies, where r is the size of the read quorum; the most recent state of the file is in the copy or copies with the highest version number. A write operation must first read the version numbers (from at least r copies), increment the highest version number to produce a new version number, and write the new state and new version number out to at least w copies, where w is the size of the write quorum.

With read quorum r and write quorum w, Gifford's algorithm requires

$$r+w > n$$

where n is the total number of copies. This condition ensures that every read quorum intersects every write quorum, so that each read sees at least one copy of the most recent write, and that writes are ordered consistently.

4.3.2 Replicated Tables

With Gifford's weighted voting, each copy of a replicated object has a single version number. The version number can become a concurrency bottleneck, since all reads and writes of the object affect or are affected by that version number, even if the operations refer to distinct elements of a structured object (for example, different entries of a directory). By using more version numbers (one on each entry), we can increase the concurrency available. In the examples of this section, diagrams like Figure 4-1 will be used.

```
                 Version

   -----------
   | 7   Foo |      3
   -----------
   | 95  Bar |      2
   -----------
```

Figure 4-1: A Table Mapping Integers to Strings

These example tables map from an integer key to a string, and each entry has a version number. Figure 4-1 shows a table with two entries, one mapping 7 to "Foo" and the other mapping 95 to "Bar". The first has a version number of 3 and the second has a version number of 2.

Because of failures at sites, not all replicas will have the same state. Existence of an entry can be ambiguous in a directory that allows both creation and deletion of entries. Thus, deleted entries must stay in the table to show that they have been deleted. To see this, consider Figure 4-2, which shows three replicas of a directory.

```
This configuration:

      --------     --------     --------
      |8  Foo| 6   |      |     |  ??  |
      --------     --------     --------
                               (Unavailable)

  Could be

      --------     --------     --------
      |8  Foo| 6   |      |     |      |
      --------     --------     --------

  or

      --------     --------     --------
      |8  Foo| 6   |      |     |8  Foo| 6
      --------     --------     --------
```

Figure 4-2: Three Copies of a Table, One Unavailable

In Figure 4-2, majority voting is in effect, with $r = 2$ and $w = 2$, and two replicas are available. One replica contains a mapping of 8 to "Foo", and one is empty. If a client wants to know the mapping of 8, what is the correct answer? It could be that the unknown copy is also empty, and the copy containing a mapping was not modified when the entry was deleted; but it could be that the unknown copy has the mapping for 8, and that the empty copy was not modified on the last update. Since one copy is inaccessible, it is not possible to determine if the entry exists or has been deleted.

This situation could be resolved if entries were retained for deleted mappings. Then, if the mapping for 8 had been deleted, it would be kept in the table and marked as "deleted", with a higher version number than the earlier data. In other words, deletion would be treated as simply a mutation of the data. However, in a directory with a high turnover of entries (such as in an academic environment with students graduating), the overhead of keeping these deleted entries may be unacceptable, particularly if each deleted entry takes up storage at several replicas. A deleted entry may be physically removed from the tables only if all replicas show the entry to be deleted.

Daniels and Spector [9] describe a variant of Gifford's algorithm for replicated directories that can reduce the overhead of deleted entries. The keys of the directory must form a total order, and there is a gap between each pair of keys representing all of the absent keys that could fall between the pair of keys. A gap is present and used for the algorithm even when no key can fall between the pair of keys. Figure 4-3 shows an empty table (mapping integer keys to strings), containing a single gap with version number 0.

```
                 Version

-----------
|         |       0
-----------
```

Figure 4-3: An Empty Table

In figure 4-4, an entry mapping 3 to "Foo" has been added to the table.

```
                 Version

-----------
|         |       0
-----------
| 3   Foo |       1
-----------
|         |       0
-----------
```

Figure 4-4: A Table with One Entry

Similarly, in figure 4-5, an entry mapping 5 to "Bar" has been added. The gap version numbers are crucial to the scheme, and are incremented when a gap "overwrites" an existing data item.[6] Figure 4-6 shows the effect of deleting 3 from the table shown in figure 4-5. This deletion creates

[6] If the second entry mapped 4 to "Bar", there would be no need for a gap between the entries. There is no integer between 3 and 4, so there is no need to represent such possible entries. However, the algorithms are kept simpler by simply having a gap between every pair of keys, regardless of whether or not a key could actually fall in that gap.

```
                    Version

 -----------
 |         |        0
 -----------
 | 3   Foo |        1
 -----------
 |         |        0
 -----------
 | 5   Bar |        1
 -----------
 |         |        0
 -----------
```

Figure 4-5: A Table with Two Entries

a gap with a version number of 2 (one higher than that for the entry being deleted). Although the table could be left this way, it is more efficient to merge the gaps as shown in the figure. The three adjacent gaps are consolidated into a single gap with a version number that is the maximum of the version numbers of the consolidated gaps.

```
                    Version

 -----------
 |         |        0
 -----------                                  Version
 |         |        2
 -----------                -----------
 |         |        0       |         |        2
 -----------                -----------
 | 5   Bar |        1       | 5   Bar |        1
 -----------                -----------
 |         |        0       |         |        0
 -----------                -----------
```

Figure 4-6: A Table with Gaps, After a Deletion

Because the empty mapping carries a version number, a client can resolve ambiguities about creation and deletion by comparing the version number of the gap and the mapping: whichever has the larger version number is correct.

The Daniels and Spector algorithm has a higher initial storage overhead than the Gifford algorithm. However, in the absence of a garbage-collection strategy for deleted entries, a table implemented with Gifford's algorithm will continue to grow as entries are deleted. Thus, a Gifford table uses an ever-larger proportion of its total storage for deleted-entry overhead. A Daniels and Spector table uses a constant proportion of its total storage for deleted-entry overhead. The replicated tables in the repository were implemented using the Daniels and Spector algorithm so as to avoid the difficulties of integrating a suitable garbage collection mechanism.

4.4 Implementing Replication

Recall that both registries and mailboxes can be replicated in the repository. Replication of mailboxes is both simpler and less important to the operation of the repository, so this section focuses on replication of registries.

The data stored in each registry replica is organized in the same way as data stored at a single non-replicated registry, except that two tables are different. A *registry_user_list* maps a user_id to an agent_list and a mailbox_list, information that can be used to find mailbox_interfaces for the user's mailboxes. A *registry_address_list* maps an address to a mailbox_interface, for delivering mail to that mailbox. The replicated tables are summarized in figure 4-7.

registry_user_list: user_id → agent_list x mailbox_list

registry_address_list: address → mailbox_interface

Figure 4-7: Tables to be Replicated

The data maintained by each registry guardian consists of a collection of related objects. Answering a request for information is not simply a matter of reading a single value: to answer a request, a registry may have to look in some tables to find the appropriate information. The significance of the data structures shown in Figure 4-7 is that they serve as "gateways" into the databases maintained by the registry guardians. Each read or write of shared information must pass through one of these tables. If each piece of data is accessible from only one entry of one top-level table, then the version number of the top-level table entry can also serve as the version number of all the information below it. Section 4.4.4 considers how to handle mutable data that is accessible by two or more paths; for the moment, assume that there is no sharing between different entries of the tables, so that only a single version number need be considered for any operation.

A parameterized type called a *semi_version_table* is used to support the implementation of the Daniels and Spector algorithm for these replicated tables. Its specification is shown in Figure 4-8. A semi_version_table is parameterized by two types:

    semi_version_table[Key, Data: type]

It maps elements of type Key to elements of type Data. The type Key must have operations to establish an ordering and also to test for equality of elements. The semi_version_table manages the gaps and version numbers of the table. An operation on a semi_version_table always returns a version number, either as part of a normal return or as part of an exception. Operations that mutate the semi_version_table must provide a new version number, which must be larger than the existing version number for the entry or gap; otherwise, the operation will signal an exception. A semi_version_table is a user-defined atomic type; its implementation is similar to the map implementation described by Weihl [44], and appears in Appendix C.

Figure 4-9 shows how both of the replicated tables are defined in terms of semi_version_table.

4.4.1 Replicated Operation Protocol

A single "registry" as used by a registry client is actually an object (called a *multi_registry*) containing references to several registry replicas. The implementation of multi_registry contains code to carry out the replicated operations on each replica and interact correctly with the replicas. This section outlines the protocol carried out by the implementation of multi_registry.

To read an item (for example, to look up an address for mail delivery), a frontend calls an appropriate multi_registry operation (for example, *lookup_address*). The operation does a set of concurrent calls to the *lookup_address* handlers of the replicas contained in the multi_registry object, one call per replica. When sufficient calls have returned values or signalled, the multi_registry compares the version numbers included in the results or exceptions and determines the correct result or exception to signal.

```
semi_version_table = cluster[KEYTYPE, DATATYPE: type]
                        is new, store, fetch, delete,
                           member, equal, transmit
  where KEYTYPE has equal: proctype(KEYTYPE, KEYTYPE) returns(bool),
                lt: proctype(KEYTYPE, KEYTYPE) returns(bool)

  new = proc() returns(semi_version_table)
    % returns a new, empty table.
    end new

  store = proc(t: semi_version_table, key: KEYTYPE,
           data: DATATYPE, vers:version)
        signals(not_stored(version))
    % if vers is less than or equal to the currently-stored version number for key,
    % signals not_stored and returns the currently-stored version number.
    % Otherwise, associates the given data with the given key, overwriting any
    % previous association for key and storing vers as the new version number.
    end store

  fetch = proc(t: semi_version_table, key: KEYTYPE)
        returns(DATATYPE, version) signals(not_in_table(version))
    % if key is in the table, returns the associated data and version number.
    % Otherwise, signals not_in_table and returns the version
    % number of the gap containing key.
    end fetch

  delete = proc(t: semi_version_table, key: KEYTYPE, vers: version)
         signals(not_deleted(version))
    % if vers is less than or equal to the currently-stored version number for
    % key, signals not_deleted and returns the currently-stored version
    % number.  Otherwise, removes the key from the table, writing vers into the
    % resulting gap.
    end delete

  member = proc(t: semi_version_table, key: KEYTYPE)
         returns(bool, version)
    % returns true iff key is in table
    end member

  equal = proc(t1, t2:semi_version_table) returns(bool)
        where DATATYPE has equal:proctype(DATATYPE, DATATYPE)
                             returns(bool)
    % returns true iff t1 is the same object as t2
    end equal

  end semi_version_table
```

Figure 4-8: Specification of Semi_Version_Table

registry_user_list = semi_version_table[user_id, ab_list]

registry_address_list = semi_version_table[address, mailbox_interface]

(ab_list = struct[agents:agent_list, boxes:mailbox_list])

Figure 4-9: Implementations of Replicated Tables

To change or delete an item (for example, to remove a user), a frontend again calls an appropriate multi_registry operation, for example *delete_user*. The multi_registry first concurrently calls each of the replicas to get a version number for the data of interest, in this case calling *user_version*. The multi_registry selects the largest version number returned or signalled and increments it by 1 to produce a new version number. It then calls all of the replicas concurrently with the new version number to be used in changing or deleting the information.

Note that version number information is completely hidden from the frontend. This means that a multi_registry can present the same interface as a non-replicated registry, and a frontend need not cache version numbers. Also note that the writes and deletes do not obtain a write lock on the item to be written before reading it. This point will be discussed further in Section 4.5.1.

4.4.2 Replicating Mailboxes

Replication of a mailbox involves a mechanism similar to the one used for registry replication. The mailbox's mailbox_interface object is stored at a registry and used by a frontend. The mailbox_interface for a normal mailbox contains a *box_id* and references to the mailsite storing that mailbox. A box_id effectively serves as a remote reference to the mailbox stored at the mailsite. Operations on a mailbox_interface are turned into handler calls to the mailsite, sending the box_id as one of the arguments to each call. A replicated mailbox contains references to several mailsites, instead of just one. The operations on the mailbox_interface are turned into concurrent handler calls to the mailsites, and the result of the operation is determined by weighted voting [16]. The mailsite's information is stored in a semi_version_table that maps box_ids to mailboxes and version numbers. For non-replicated mailboxes, the version numbers are meaningless and ignored by the caller.

4.4.3 Finding a Replicated Registry

The object representing a replicated mailbox can be stored in a replicated registry, but a replicated registry stored in a replicated registry could lead to bootstrapping problems — how to find a registry without first finding a registry. There are two techniques used in the repository. Replicated registries can be stored in the Argus catalog, and every registry replica must be stored in the catalog individually. If no replicated registry can be found in the catalog, the replicated registry type provides a *find* operation that looks up the single registries in the catalog and dynamically creates a new replicated registry from them.

4.4.4 Version-lists

Section 4.4 assumed that the distributed respository's data is divided between two tables, so that any item of data can be reached through one and only one of these tables. In such a divided system, the version number associated with the top-level table entry may be used as the version number for any subsidiary item of data.

In the actual data structures of the distributed repository, there are objects that are accessible from both the registry_user_list and the registry_address_list. A given mailbox's mailbox_interface, which contains a mutable list of the addresses pointing to the mailbox represented, is such an object. The sharing of these objects is both convenient and efficient for certain operations (primarily when deleting a mailbox, so that the addresses of the mailbox can be easily removed).

Whenever there is (atomic) mutable data that can be reached by two different paths, the version number at the top of one path is not enough information to know if the local copy is the most recent.

```
      User A          Addr X              User A          Addr X
    v:3     \        /   v:7        v:3     \           /   v:7
             \      /                        \       /
              MBox                              MBox
```

Figure 4-10: Two Copies of Data Reachable by Two Paths

In figure 4-10, mailbox_interface "MBox" is accessible by two paths. One path is through the registry_user_list, "User A" with version number 3. The other path is through the registry_address_list, "Addr X" with version number 7. Assume there are three replicas (Left, Right, and Unshown) and the data is write-two, read-two. An action writes the Left and Unshown copies of the mailbox, mutating "MBox" to substituting "NewMBox" and incrementing the version number of "User A" (see figure 4-11).

```
    User A           Addr X                 User A          Addr X
 v:4     \           /   v:7           v:3     \           /   v:7
          \         /                           \         /
          NewMBox                                   MBox
```

Figure 4-11: Two Copies of Data Reachable by Two Paths, After a Write

When a subsequent action reads the Left and Right copies using the "User A" path, it gets the correct result of NewMBox. However, if an action reads using the "Addr X" path, it finds that both entries have the same version number but different data.

This problem arises because we updated the version number of the path rather than the version number of the data. If there is no sharing of data between paths (as for example in Figure 4-12) then all changes to

```
    User A                                  User A
 v:4     \                             v:3     \
          \                                     \
          NewMBox                                   MBox
```

Figure 4-12: Two Copies of Data With No Sharing

all data can be treated as changes in the top-level path and its version number. To solve this problem, a version number is attached to the shared data as well as the top-level table. Instead of returning a single version number to be compared with the version numbers from other replicas, the lookup operation returns a sequence of version numbers, with the first version number from the top-level table. This sequence is called a *version-list*. The correct result of a call using local lookups and version-lists is the result with the largest version-list (as described below). If not all of the version-lists are equal, the index at which the version-lists diverge is the first data structure that must be updated with the correct values when performing a write.

```
    User A         Addr X            User A         Addr X
v:3     \         /   v:7        v:3     \         /   v:7
         \       /                        \       /
          MBox                             MBox
          v:1                              v:1
```

Figure 4-13: Two Copies of Data Using Version-Lists

Figure 4-13 shows the effect of this modification on the example of Figure 4-10. Here, a lookup using the "User A" path gives a version-list of [3, 1] and a lookup using the "Addr X" path gives a version-list of [7, 1].

```
    User A         Addr X            User A         Addr X
v:3     \         /   v:7        v:3     \         /   v:7
         \       /                        \       /
        NewMBox                            MBox
          v:2                              v:1
```

Figure 4-14: Two Copies of Data Using Version-Lists, After a Write

Figure 4-14 shows the effect of version-lists on a write. The new data written (NewMBox) has a new version number. Now, even a lookup using the "Addr X" path will give the correct result, since the new data (on the left) has a version-list of [7, 2] and the old data (on the right) has a version-list of [7, 1]. Comparing elements left-to-right, [7, 2] is larger than [7, 1]; therefore the left-hand replica has the correct data.

A version number within a version-list indicates a particular state of a corresponding table entry. The comparison of version-lists element-by-element is essentially the comparison of corresponding tables. As soon as one copy is more up-to-date than another, the subsequent information of the out-of-date copy is uninteresting. The end of a version-list comes either at the entry of interest or at a point where an intermediate table did not contain the entry needed. Thus, a version-list that is shorter than another but has all the same version numbers (so that it is a prefix of the longer list) suggests that there is an error somewhere. Identical version numbers should indicate identical states, so that it would not be possible for one to have an entry while the other does not.

The following are the formal rules for comparing version-lists:

1. Two version-lists are *incompatible* if one is shorter than the other and the shorter is a prefix of the longer.

2. Two version-lists are *equal* if they are the same length and contain the exact same numbers.

3. Two version-lists that are neither equal nor incompatible are compared from element-by-element, from the first element to the s^{th} element, where s is the length of the shorter list. As soon as one list has an element that is greater than the corresponding element of the other list, the former is *larger* than the latter.

The techniques used for dealing with information reachable from two replicated data structures are useful, but not vital, in the repository. However, they may be important for future distributed programs that have more complex sharing of replicated data than the distributed repository, so this section has described the techniques in more detail. One motivation for version-lists is that they effectively compress several replicated lookups into a single call. Such compression is worthwhile in the current Argus implementation, since the extra cost of processing version-lists is small compared to the cost of replicated calls.

4.5 Summary and Discussion

The distributed repository requires replication techniques that maintain consistency, perform well, tolerate partitions, and can be implemented in Argus. A large number of published algorithms fail to meet one of these criteria. Two algorithms — Gifford's Weighted Voting [16] and Daniels and Spector's Replicated Directories [9] — meet all of the criteria. The Daniels and Spector scheme is the basis for replication in the distributed repository, since it avoids the need for garbage collection to deal with deleted entries.

4.5.1 Deadlock

A ***deadlock*** is a situation in which a collection of actions ceases to make progress because of a ***circular wait*** for locks in the system [22]. A two-action example of deadlock is when action A has locked a resource *R1* that action B needs, while B has locked a resource *R2* needed by A. Neither action can proceed until the other releases a locked resource, but neither can release a locked resource until it can proceed.

Replication deadlock arises only in replicated systems, where concurrent actions may access different copies of a single resource in such a way as to cause deadlock. Replicating a system immediately introduces new opportunities for deadlock. For example, consider actions A and B trying to access a single resource R that has three copies R_1, R_2, and R_3. Any operation requires two copies of the three. Unfortunately, R_2 is not available — its node has crashed. Now, if action A locks R_1 and action B locks R_3, there is a deadlock.

Replication deadlock can be prevented by locking resources in a fixed order. To return to the two-action example given previously, a deadlock cannot arise if both actions A and B must lock the copies R_1, R_2, and R_3 in that order. Imposing a total ordering on resource locking ensures that no circular wait can arise. However, in the current Argus implementation, a single null remote call takes about 38 ms and forking concurrent actions takes roughly $0.9n$ ms, where n is the number of coarms forked [39]. Thus, executing a replicated call to 3 guardians concurrently requires about 41 ms, whereas executing those calls in sequence requires about 114 ms. If calls to replicas must be performed concurrently to achieve reasonable performance, as is the case in the distributed repository, there is no way to control the order of accesses to different replicas.

Replication deadlock is most likely to occur on the heavily-used "top-level" directories, shown in figure 4-7. There are relatively few concurrent accesses to the same user, same address, or same mailbox. The semantics of the top-level directories allows concurrent operations on different elements. By locking objects in a manner that allows such concurrency, most of the replication deadlocks that could occur are eliminated. Thus, a user-defined atomic type is not only useful to obtain greater concurrency, but in fact is necessary to reduce the number of replication deadlocks.

Because of replication deadlocks, writing operations do not obtain a write lock before reading the version number for an item (as described in Section 4.4.1). Two writes or deletes can conflict, possibly leading to a deadlock, only if they involve the same key (user or address). Any two writes to the same key can suffer replication deadlock, and those deadlocks cannot be prevented without causing an unacceptable performance penalty. Obtaining write locks cannot prevent replication deadlock; in fact, obtaining write locks in an unfortunate order may cause replication deadlock.

Most of the activity in the mail repository is composed of short actions requiring relatively few resources. These short actions have few conflicts since users read only their own mailboxes, and submitting mail is decoupled from receiving or reading mail. For any conflicts that do occur, such as mail delivery deadlocking with mail reading at a replicated mailbox, high-level timers on operations time out any operation that may be involved in a deadlock. This seems to be the most reasonable solution to the problem, since complete prevention of deadlocks is incompatible with performance requirements. True deadlock detection (finding circular waits and breaking them) is not built into the Argus system, and Argus applications have no direct access to the lock and action information that would be needed to build deadlock detection.

The avoidance and detection of deadlocks (by dynamically linking in a debugger) consumed a significant amount of development time for the distributed repository. A deadlock detection system in Argus would greatly improve the programmer's ability to debug systems.

4.5.2 Cluster-Based Replication and Guardian-Based Replication

Using the terminology from Herlihy's thesis [19], the implementation of a replicated object consists of a collection of *data managers* (DMs) that hold the state of the object, and a collection of *transaction managers* (TMs) that manage access to the DMs.[7] To provide resilience and independent crashes, each of the DMs must be in a separate guardian. The TM can either be in a guardian called by the client accessing the replicated object (so that the replicated object is a *guardian-based subsystem*) or the TM can be a cluster contained in the client's code (so that the replicated object is a *cluster-based subsystem*).

With a guardian-based subsystem, it is simple to replace the TMs. However, a guardian-based interface is much more expensive. The expense has two sources: one is that multiple guardians are required as TMs, and one is that extra remote calls are needed.

A single guardian is not sufficient to act as a TM for a replicated object, since the availability of the TM must be at least as great as the availability of the DMs. Additional guardians are much

[7]Herlihy now uses the term "front-end" instead of TM and "repository" instead of DM. These aren't good terms for this thesis, for obvious reasons. Herlihy's change of terminology is intended to avoid confusion with a conflicting technical use of the terms TM and DM in the general database literature. Readers familiar with those definitions of TM and DM should be aware that TM and DM have a different meaning here.

more expensive for the whole distributed system than some additional code in the client. Argus thus discourages the construction of guardian-based subsystems.

An operation on a replicated object always involves two sets of calls: one call from the client to the TM, and a collection of concurrent calls from the TM to the DMs. Since a local call is much cheaper than a remote call, performance is far better if the client-to-TM call is a local call rather than a remote call.

Based on these performance considerations, a cluster-based replication scheme is the only viable approach to replicated objects in an Argus implementation of the repository. As previously described, replicated registries and replicated mailboxes are implemented by putting the additional data and code needed into the multi_registry and mailbox_interface objects used by clients.

4.5.3 Another Replication Scheme

The key reason for using the Daniels and Spector scheme was that it avoided the need to do some form of garbage collection of deleted entries that would be needed with Gifford's algorithm. In retrospect, it seems that it might be straightforward to use Gifford's algorithm and a highly-available server [31] for garbage collection in the system. If partitions were a particular problem, the replication algorithm might be modified along the lines of the recently-published Views algorithm of El Abbadi and Toueg [13].

Chapter Five

Reconfiguration

Good old Watson! You are the one fixed point in a changing universe!
—Sir Arthur Conan Doyle

How can I be sure
In a world that's constantly changing?
—The Rascals

A *configuration* of an Argus distributed program describes

- the number and kinds of nodes in use,
- the relationship between guardians and nodes (which guardians are running on which nodes), and
- inter-guardian communication patterns.

Conventional programming methods produce applications with *static* configurations. The configuration is fixed when the executable program is constructed, and cannot be changed without reconstructing the program. In such applications, changing resources used by the program requires "taking the system down" and rebuilding it with the appropriate changes.

Long-lived systems must be altered to meet changing requirements, to tune performance, or to correct faults. When the system is intended to be highly available as well as long-lived, it is important to be able to change components of the system without stopping all of the system from running. Such a system has a *dynamic configuration*, and the techniques for changing its configuration are termed *dynamic reconfiguration*.

This chapter discusses the dynamic reconfiguration techniques used for the distributed repository. First, I review Bloom's work on dynamic replacement of guardians [6]. Then, I introduce a new method for building and changing reconfigurable systems, and contrast it with Bloom's work. Then, I consider the implementation of this method in Argus.

5.1 Bloom's Work

Bloom [6] studied the problem of dynamic replacement of modules in Argus. Her goal was to determine what kinds of replacement would be meaningful, and how to carry out all of those replacements. Bloom emphasized methods that would not restrict replacements unnecessarily.

The guardian was chosen as the smallest unit of replacement because first, a guardian has a well-defined state to be moved to the replacement guardian, and second, replacement of smaller modules might require replacement of the guardian anyway.

Bloom also supported replacement of multi-guardian subsystems, such as the replicated objects discussed in section 4.5.2. Bloom defined a formal model that specified when a replacement is legal. The model was intended to assist programmers in determining whether a replacement was correct and was not used by the replacement mechanism to check a replacement. The actual mechanism proposed ensured full type-checking of replacements, and ensured that no information was lost from the stable state of the guardian(s) being replaced. However, the replacement mechanism required a number of special features from other parts of the Argus system. An Argus guardian is either "up", providing services to any client, or "crashed," providing no services at all. Bloom's replacement mechanism required a new state for guardians, so that guardians could be "removed from service". When removed from service, a guardian appeared to have crashed to every entity except the replacement system. Bloom also assumed the existence of a service for rebinding handlers, so that messages intended for old handlers could be forwarded automatically to new handlers. These facilities do not exist in the current prototype implementation of Argus, and no work except Bloom's has required these facilities. Therefore, I looked for an approach to reconfiguration that would not require special facilities.

5.2 A New Method

Bloom's work was intended to be a general replacement method that could be used to replace arbitrary modules in running Argus applications. This chapter presents a new method that deals with replacement in systems that are designed for replacement, where the reponsibility for replacement rests entirely with application-level code. Use of this method requires the construction of *replaceable guardians* that provide a few specific facilities at the application level so that they may be replaced in a straightforward manner. In addition, any guardian communicating with replaceable guardians is built so that it is informed of replacements, or can recover from a failure caused by a replacement.

The new method requires no modifications to the underlying Argus system. It takes advantage of a feature of the current implementation to help perform replacement. Although the new method is presented in the context of the distributed repository, it is general and can be applied to other distributed programs written in Argus.

5.3 The Replacement Mechanism

This section considers changes to the implementation of a guardian while the guardian provides the same interface. Section 5.5 deals with changing the interface of a guardian, which necessarily means changing the implementation.

Creation and destruction of guardians are straightforward in Argus. Guardian types provide creator operations that create new instances of guardians, and guardians can destroy themselves by executing a **terminate** statement [29]. Since no guardian can actually destroy another, each guardian that can be destroyed must provide a handler (called ***finish*** by convention) that will execute a **terminate** statement after performing any necessary checking to ensure that it is safe to destroy the guardian.

The current Argus implementation does not provide primitive operations for moving or replacing Argus guardians, so a mechanism for replacement must be invented. In performing a replacement, no information may be lost from the stable state of the guardian. Simply substituting a newly-created guardian for an existing guardian will lose all of the state of the existing guardian.

A simple replacement mechanism involves adding another handler to the interface of every replaceable guardian. The handler (call it *get_state*) allows the caller to get the entire stable state of the guardian in some predefined, transmissible form (a *state object*). The new implementation of the guardian must have a creator (call it *create_with_state*) that takes the state object as an argument, and initializes the new guardian's stable state from the state object. Since guardians are linked separately, it is possible to have two guardian images of the same type with different implementations.[8]

[8]There are mechanisms defined for choosing between guardian images of the same type on the same node. However, they have not been implemented yet. Currently, replacement must be performed using at least two nodes.

Abstractly, replacement of a collection of guardians proceeds as follows:

1. Get exclusive access to the guardians involved.
2. Start a topaction.
3. Replace a single guardian:
    a. Call *get_state* of the old guardian, which returns a state object representing the entire stable state of the guardian.
    b. Call *create_with_state* of the new guardian implementation, which creates a new guardian, initializing its state from the state object.
    c. Replace handlers so that calls to the old guardian now go to the new guardian.
    d. Destroy the old guardian.
4. Repeat step 3 for each guardian to be replaced.
5. Commit the topaction.

Although this scenario is presented in terms of replacing a single guardian with another guardian, it is equally plausible to consider multi-guardian replacements: for example, replacing 3 guardians with two others. The key difference is that the creators of the two new guardians would each take 3 state objects as arguments, rather than one.

There are several parts of the scenario that require further explanation: how to provide exclusive access to a guardian, how to implement state objects, and how to replace handlers. The following sections describe these issues.

5.3.1 Exclusive Access

There must be a locking mechanism to ensure that after a replacing action calls *get_state*, the old guardian does not allow the state to change until the replacing action commits or aborts. The simplest method for doing this is to disallow all incoming handler calls, so that their callers receive ***unavailable*** exceptions. However, it is unpleasant to write guardians where each handler first tests some global lock to determine if it should proceed. Additionally, every call to a replaceable guardian must pay the extra cost of checking the global lock. One solution is to modify the language to support some special per-guardian lock, but changes to the language are to be avoided, especially changes that may not be useful in other situations.

Instead, there is a two-step process to gain exclusive access to a guardian. First, crash the guardian; this ensures that it is not accessible to anyone. Then, replace the normal guardian code with special replacement code. The normal guardian code includes the handlers *get_state* and *finish* in its interface, but the handlers have trivial implementations: for example, they can signal an exception when called (see figure 5-1). Since the replacement handlers have only a trivial implementation in normal operation, a client cannot inadvertently destroy the guardian by calling *finish*.

```
my__guard = guardian is ...
                    handles do__work, get__state, finish, ...

    do__work = handler(stuff:stufftype) signals(unavailable(string))
       work()
       more__work()
       even__more__work()
       end do__work

    get__state = handler() returns(image)
                           signals(cant__reconfigure(string))
       signal cant__reconfigure("No replacement code")
       end get__state

    finish = handler() signals(cant__reconfigure(string))
       signal cant__reconfigure("No replacement code")
       end finish

    . . .

    end my__guard
```

Figure 5-1: The normal guardian implementation

The replacement code fully implements *get_state* and *finish*, but signals *unavailable* for all of the other handlers (see figure 5-2). Since the replacer provides the implementation of the replacement code, *get_state* and *finish* can be arbitrarily complex so as to provide checking and authorization of calls made to the handlers. This helps to ensure that even during a replacement, the guardian's state is not inadvertently released or destroyed.

In the current implementation, the code is accessible via the underlying Unix operating system, whereas the stable state is managed by the Argus system. Probably any reasonable Argus implementation would have a similar separation between code and stable state. With this

```
my__guard = guardian is ...
                handles do__work, get__state, finish, ...

    do__work = handler(stuff:stufftype) signals(unavailable(string))
       signal unavailable("Guardian being replaced")
       end do__work

    get__state = handler() returns(image)
                          signals(cant__reconfigure(string))
       so:state__object := cons__up__state()
       return(image$create[state__object](so))
       end get__state

    finish = handler() signals(cant__reconfigure(string))
       terminate
       end finish

    ...

    end my__guard
```

Figure 5-2: The guardian implementation during replacement

separation, the guardian's code can be replaced so long as the implementation of the stable state is not changed.

5.3.2 State Objects

The state object returned by *get_state* and accepted by *create_with_state* will be of type **image**. **Image** is an escape from type-checking for transmissible types. Any transmissible object Obj of type T can be translated into an **image** object Img by an *image$create* call:

Img:**image** := **image$**create[T](Obj)

and the resulting **image** object Img can be translated back to an object of type T by an *image$force* call:

Obj:T := **image$**force[T](Img)

The type **image** is used so that changes in the representation of the state object do not affect the interface of the guardian. Otherwise, a change in the representation of the guardian's internal state would change the interface of the guardian. However, using **image** means that some compile-time type-checking is lost; if there is a mismatch between the object returned by *get_state* and the object expected by *create_with_state*, it will be detected only at run time (when *create_with_state* tries to force the state object to the type it is expecting).

There is a parallel between the transmission of state between guardians and the transmission of abstract values in handler calls. In Argus, abstract objects of abstract type are implemented by objects of a representation (*rep*) type within a guardian. Different guardians may have different implementations of the abstract type, so that transmitting the representation is not an appropriate way to send abstract values. Instead, an *external representation*, or *xrep*, is defined for the type. The xrep serves as a canonical form for transmitting abstract values, and each implementation of the abstract type provides procedures for translating between the xrep and that implementation's rep [18]. Similarly, the state object is a canonical form for the guardian's state. Each guardian being replaced must translate its state into this canonical form, and a guardian being created as a replacement must initialize itself from the canonical form. However, the new guardian's stable state can be represented quite differently from the old guardian's stable state. State objects are somewhat more flexible than xreps of transmissible types because state objects are transmitted as **images**. The canonical form of the state can change without affecting the interface of the guardian type. Both old and new guardians must agree on the actual type that was encoded into the state object of type **image**, but this agreement need not take place until replacement time, and a single guardian type can be replaced using a number of different types of state object. In contrast, a change to the xrep of a transmissible type produces a new type.

5.3.3 Replacing Handler Objects

After a replacement, calls to a handler of the old guardian should go to the corresponding handler of the new guardian. Bloom [6] assumed that existing handler objects would not change, and that clients of a replaced guardian need not be notified of the replacement. Instead, the Argus system would map calls on those objects to the new handlers. In contrast, the new method assumes that clients of a replaced guardian are prepared for replacement, and that the Argus system does not provide a service for rebinding handlers. These assumptions mean that the problem to be solved is different from handler rebinding: obsolete handlers must be found and *replaced* with the corresponding new handlers. The process of replacing handlers must be inexpensive, with low costs for communication, computation, and storage. In addition, the replacement process must be effective, able to replace all of the handlers that must be replaced. Finally, the replacement process must tolerate failures, so that it can work even when some guardians possessing an old handler have crashed or are inaccessible.

One simple method assumes that the action performing the replacement knows which client guardians may be using the old server guardian. Each of these client guardians provides a handler that takes references to the old and new server guardians as arguments. If the client is using or storing a reference to the old server, it instead stores a reference to the new server. However, this method breaks down if any of the client guardians have crashed or are inaccessible. A crashed or inaccessible guardian cannot respond to a handler call and cannot change its state by storing a new object.

Rather than requiring that all clients be available when a replacement is performed, the new reconfiguration method stores guardians in the Argus *catalog* under fixed names. The Argus catalog permits a transmissible object to be stored with a user-specified string name. Catalog operations are parameterized by type. The new method uses the following catalog operations:

    insert = **proc**[T](name:**string**, object:T) **signals**(exists)

    replace = **proc**[T](name:**string**, object:T) **signals**(not_found)

    lookup = **proc**[T](name:**string**) **returns**(T) **signals**(not_found)

Guardian **User** uses a guardian of type GT1. Initially, the guardian ***g*** of type GT1 is stored in the catalog by the call

    catalog$insert[GT1]("gname", *g*)

This is typically performed by the creator that is called to create ***g***.

To find ***g***, **User** performs

    catalog$lookup[GT1]("gname") *returns* ***g***

In order to replace ***g*** with a guardian ***h***, the replacer performs

    catalog$replace[GT1]("gname", ***h***)

At some point, **User** will attempt a call to *g*, which will fail since the guardian no longer exists. This failure is critical, since it is at this point that **User** picks up the new guardian. On failure, **User** tries the catalog lookup again:

    catalog$lookup[GT1]("gname") *returns* ***h***

5.4 An Example Replacement

Consider a simple mailsite storing a collection of mailboxes indexed by unique keys. Each mailbox contains text in a compressed form, and the collection of mailboxes is organized in a balanced tree. The system implementor decides to replace this mailsite with one that uses more space for faster access; the new mailsite uses a hash table and stores the messages as ASCII text, to save the cost of decoding the compressed form. The state object will be a linear list of key-mailbox pairs, with the messages in ASCII text; its type will be State_Obj_Type (See figure 5-3).

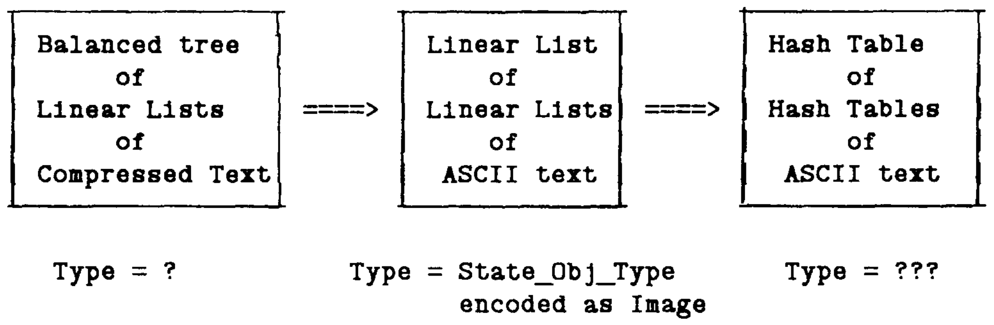
Balanced tree of Linear Lists of Compressed Text
====>
Linear List of Linear Lists of ASCII text
====>
Hash Table of Hash Tables of ASCII text
Type = ?
Type = State_Obj_Type encoded as Image
Type = ???

Figure 5-3: Old State, State Object, and New State

There must be a code image for the guardian being replaced that includes a *get_state* routine that will translate the old state into the state object. In addition, the *create_with_state* creator of the new guardian must correctly translate from the state object to the new guardian's state.

The replacement proceeds as follows:

1. Crash the old mailsite.
2. Replace the old mailsite's code with the new code fully implementing only *get_state* and *finish*.
3. Restart the mailsite.
4. Begin a topaction.
5. Call the old mailsite's *get_state* handler to get the state object.
    a. *Get_state* translates the mailsite's balanced tree of linear lists of compressed text into a linear list of linear lists of ASCII text, whose type is State_Obj_Type.

   b. It takes the new list and calls *image$create* on it.

   c. It returns the resulting **image** object to the caller.

6. Call the new mailsite's *create_with_state*, providing the state object just received.

   a. *Create_with_state* calls *image$force* on the state object, forcing it to State_Obj_Type.

   b. It translates this linear list of linear lists of ASCII text to a hash table of hash tables of ASCII text.

   c. It performs any other initialization necessary for the new guardian.

7. Call the catalog to replace the old mailsite with the new one.

8. Call the old mailsite's *finish* handler to destroy it.

9. Commit the topaction.

We will not lose the guardian's state or entry in the catalog if one of the steps goes wrong—for example, if a node crashes, or there is a programming error in *create_with_state* so that the state object is forced incorrectly. The topaction is simply aborted if anything unexpected occurs. The old guardian and its catalog entry will not be destroyed unless the topaction commits. If we need to restore the old guardian to service, we can replace its code with the old code.

5.5 Changing a Guardian's Interface

If the old guardian provides handlers for *get_state* and *finish*, and the new guardian provides a *create_with_state* creator, replacement can proceed even if the new guardian provides an interface that is different from the old guardian. The problems with changing a server's interface arise in the clients. If a guardian is to be replaced by a guardian with a different interface, the new guardian is of a different *type* from the old guardian. In a strongly-typed language like Argus, this means that the new guardian cannot be treated as if it were of the same type as the old guardian. For example, consider a mailsite with the handlers *read_mail*, *deliver_mail*, *create_user*, and *ping*. Most of these handlers are actually called by only one client or type of client:

- frontend *fe* calls *read_mail* and *ping*;
- alien *al* calls *deliver_mail* and *ping*; and
- registry *reg* calls *create_user* and *ping*.

In this example, there is no way for a user to delete mail. If we want to add a *delete_mail* handler for the frontend *fe* to call, we may create a new mailsite which includes this new handler. For the frontend to use this new handler, it must be replaced with a new implementation of frontend. However, the alien and the registry must also be replaced, even though they don't use the new handler. Also, any frontend that chose not to use the new handler would still have to be replaced; otherwise it could not use the "old" handlers of the new guardian interface type.

This illustrates the problems that arise when clients use guardian interface objects as their interfaces to guardian services, even though they actually need only a much narrower interface. Clients are thereby exposed to changes in the interface that should not affect them.

Instead of using guardian interface objects, clients can use different interface objects. Each special interface object will be an immutable collection of named handlers, each handler potentially of a different type. In Argus, such a collection would be constructed as a **struct** of handlers [29]. For example, the mailsite would have three different interfaces:

```
fe_interface = struct[read_mail: read_mail_type,
                      ping: ping_type]

alien_interface = struct[deliver_mail: deliver_mail_type,
                         ping: ping_type]

reg_interface = struct[create_user: create_user_type
                       ping: ping_type]
```

A mailsite would put each of these interfaces into the catalog when it is created, instead of simply putting the whole guardian interface into the catalog. Now, the addition of a *delete_mail* handler only affects the frontend:

```
new_fe_interface = struct[read_mail: read_mail_type,
                          delete_mail: delete_mail_type,
                          ping: ping_type]
```

The other interface objects can be replaced as above, since their type hasn't changed.

This technique requires the server to identify all the subsets of its interface that will be used by clients. This is simple to do in the distributed repository, since the clients of all guardians are other components of the repository or else non-Argus components.

It is not reasonable to require that all guardians of a particular type are replaced by guardians of another type at the same time. With replication, it is possible that clients will have to deal with two or more different guardian types that implement copies of the same data structure or service. Assuming that the client involved actually uses the different interfaces, the multiple types can be hidden inside a cluster whose representation is a tagged, discriminated union (a **oneof**) [29]. Such a cluster is much like the mailsite_interface and multi_registry clusters described in Chapter 4. Instead of hiding the replication or existence of remote guardians, this sort of cluster hides the existence of multiple types of guardian. The *find* operation finds an operational frontend of either interface and sets the tag of the **oneof**. Operations on objects of the cluster type are then translated into appropriate handler calls, based on the tag of the **oneof** (see figure 5-4).

As shown in Figure 5-4, the reconfiguration method uses a *failure* exception as an indicator that a guardian has been destroyed, assuming that any problems with an existing guardian will appear as *unavailable* exceptions. Certain programming errors in the encode/decode routines of abstract type implementations may also cause *failure* to be signalled. If the caller catches the *failure* signal, the string returned with the exception may be used to determine whether to attempt a reconfiguration. Otherwise, the unhandled *failure* will simply cause the calling guardian to crash.

5.6 Another Example

Now these notions of replacement with a different interface can be applied to a replacement of the mailsite. This time, the transformation of state is ignored, although such a transformation could be carried out at the same time. A *delete_mail* handler will be added to the mailsite. A frontend will be replaced so that it can use the new interface; however, since there is another mailsite in the system that cannot be replaced at the moment (for whatever reason), the new frontend must be able to use both types.

1. Crash the old mailsite.
2. Replace the old mailsite's code with the new code implementing only *get_state* and *finish*.
3. Restart the mailsite.
4. Begin a topaction.
5. Call the old mailsite's *get_state* handler to get the state object.

```
frontend__clust = cluster is find, read__mail

    rep = oneof[old:fe__interface, new:new__fe__interface]

    find = proc(site:string) returns(cvt) signals(not_found)
        begin
            old:fe_interface
            old := catalog$lookup[fe_interface](site)
              except when not_found: exit not_old end

            old.ping()
              except when failure(*): exit not_old
                     when unavailable(*): % ok, just crashed
                          end

            return(rep$make_old(old))
            end
          except when not_old:
                        new:new_fe_interface
                        new := catalog$lookup[new_fe_interface](site)
                          except when not_found: signal not_found end

                        new.ping()
                          except when failure(*): signal not_found
                                 when unavailable(*):
                                      % ok, just crashed
                                      end
                        return(rep$make_new(new))
                        end
        end find

    read__mail = proc(widget:cvt, number:msg__id) returns(message)
                        signals(unavailable(string), failure(string))
        tagcase widget
            tag old(fe:fe__interface):
                return(fe.read__mail(number))
                  resignal failure, unavailable
            tag new(fe:new__fe__interface):
                return(fe.read__mail(number))
                  resignal failure, unavailable
            end
        end read__mail
end frontend__clust
```

Figure 5-4:A Cluster to Hide Two Interfaces

6. Call the new mailsite's *create_with_state*, providing the state object just received.

    a. *Create_with_state* initializes the stable state from the state object.

    b. It then inserts a new_fe_interface into the catalog, and replaces the alien_interface and reg_interface objects in the catalog.

7. Call the old mailsite's *finish* handler to destroy it.

8. Commit the topaction.

9. Crash the old frontend.

10. Replace the old frontend's code with new code implementing only *finish*.

11. Restart the frontend.

12. Begin a topaction.

13. Call the new frontend's *create*. The new frontend created is able to deal with both the old interface (fe_interface) and the new interface (new_fe_interface).

14. Call the old frontend's *finish* handler to destroy it.

15. Commit the topaction.

16. Whenever the frontend first tries to call the replaced mailsite, it will receive a *failure* exception. It can then execute a *find* operation, which will find the new mailsite (new_fe_interface) in the catalog.

5.7 Discussion and Related Work

I have presented a new method for reconfiguration of Argus distributed programs. The method depends on constructing replaceable guardians whose state can be fetched from them, on constructing clients of such replaceable guardians so the clients can tolerate reconfigurations, and on the ability to crash a guardian and replace its code without altering its stable state. The new reconfiguration technique is simple and effective. Since the changes are made in Argus application code, the Argus system need not be modified to support this style of reconfiguration.

There are several elements in the reconfiguration method that can be considered abstractly. The first element is a *fixed reference point*, allowing an indirect reference to the objects that move. In the new technique, this fixed reference point is the Argus catalog. The catalog cannot be reconfigured by the techniques of this chapter, since a client of the catalog that cannot find the catalog has nowhere else to check.

The second element in the new reconfiguration method is the use of a canonical form for the representation of the state of a guardian. Rather than performing a transformation of old state to new state (like Bloom's transformation function [6]), I perform a transformation from the old state to an intermediate form and from the intermediate form to the new state. This gives a cleaner structure to the process of transformation and forms a useful parallel with the encoding and decoding of abstract data types.

The third element is the replacement of handlers, which in this reconfiguration method takes the form of catalog lookups by client guardians.

However, this implementation of reconfiguration is far from ideal. Constructing applications so as to be easily reconfigured is a reasonable approach, but it is difficult to express the necessary concepts in Argus. A better reconfiguration implementation would support the state transformations in the language (in much the same way that transmissible types are supported) and would allow rebinding of handlers within the system. Such rebinding would be supported by a name server performing functions similar to those performed by the catalog in the current implementation. The server would map a handler identifier to another handler identifier and a guardian. The run-time system would (at least conceptually) perform the mapping of handler id to handler on each call. [21]

CONIC [25] is another language intended for programming distributed applications. CONIC was constructed specifically to explore issues of dynamic reconfiguration, and uses the interconnection of typed ports for remote communication instead of calling handler objects. CONIC has separate languages for programming and configuration, so that programs have no control over their configuration and no way to determine their configuration. This means that it is easier to configure programs, but it is impossible for programs to configure themselves.

There are certain advantages to using typed ports for communication. Each outgoing port must be connected to some incoming port of another module, which means that modules are affected only by the portion of the interface that they use. No special interfaces or system-defined subsetting are required. However, the basic paradigm of passing handlers (in Argus) cannot be used, since ports cannot be passed. Instead, calls are forwarded from one module to another. For example, a frontend would probably not maintain a connection to every mailsite in the system. Instead, it would have a connection to the equivalent of the catalog, which could forward the call as necessary. Since a frontend cannot dynamically establish more outgoing ports, it would be difficult for it to accommodate an increasing workload; either the catalog or its connection(s) to the catalog could easily become bottlenecks for the system. CONIC has no notion of stable storage or actions, so that it is not even clear when to perform a replacement or how to undo it if part of it fails. In general, simple reconfiguration is easier in CONIC; however, when issues of replication, concurrency, crashes, and large systems are considered, it is not clear that CONIC will work well.

Chapter Six

Conclusions

Causa finita est.
(The case is concluded)
—St. Augustine

I am a Bear of Very Little Brain,
and long words Bother me.
—Winnie the Pooh

This thesis has presented a design for a distributed mail repository using replication and reconfiguration to achieve scalability and high availability. The distributed repository uses registry, mailsite, frontend, and alien guardians to provide its services to agents and external mail systems. The guardians are designed to run on nodes of a large distributed system composed of many interconnected local area networks.

The repository's naming information is partitioned between registry guardians by simply dividing the name space into ranges and assigning each range to a registry. Registry information is replicated by a voting technique using tables with version numbers for each entry [9]; the tables are implemented as user-defined atomic types to gain concurrency and avoid deadlocks. Each "replicated registry" used by a client is actually an object containing references to registries; the implementation of the object's type includes code to perform the replicated calls correctly.

Individual mailboxes are replicated instead of entire mailsites, so that the expense of replicated mailboxes can be managed. As with replicated registries, each replicated mailbox is an object containing references to mailsites, with the implementation of the object's type including code to correctly perform replicated operations at the mailsites.

The distributed repository can be reconfigured by using new techniques that operate at the Argus application level. By crashing guardians and replacing their code without changing their stable state, an administrator can gain exclusive access to guardians for replacement. Each replaceable guardian can return a state object representing a canonical form of its state, which can be used to initialize a replacement guardian.

This chapter evaluates the Argus programming language for its usefulness in building the distributed repository, compares the thesis to related work, and draws some final conclusions.

6.1 Evaluating Argus

A number of Argus concepts and constructs were helpful in the design and implementation of the distributed repository. Some parts of Argus caused problems, and in some cases the language did not allow an effective expression of what was needed for the distributed repository. First, I consider the elements of Argus which were helpful: guardians, actions, user-defined atomic types, remote procedure call, strong typing checked at compile time, automatic storage allocation, and the Argus debugger. Then I discuss the drawbacks involved in using Argus: deadlocks, the awkwardness of retrying actions, problems of process scheduling, and the difficulty in expressing replication.

6.1.1 Argus Advantages

Guardians served to divide the system into components which could be independently created, destroyed, or replaced. Guardians also served as units of replication, since a guardian could crash without affecting another.

The atomic actions of Argus simplified many aspects of the repository. In fact, it is difficult to imagine building the repository without them, since there would have been so much additional complexity in the system. The effect of actions is largely invisible since they present the model a system designer wants: simple, clean semantics for failures and concurrency. By greatly reducing the number of states to be considered due to failures or concurrency, actions made the entire repository implementation conceptually simpler. The complexity of the distributed repository came almost entirely from the use of replication and reconfiguration; a straightforward distributed repository was simple to write and run in Argus. The nesting of actions was important for concurrent actions performed on copies of a replicated object, so that the replicated operation could commit at all copies or abort at all copies without committing or aborting the higher-level operation being performed. Nesting of actions was also important since it supported the Argus model of "at most once" remote procedure call.

The semi_version_table was implemented as a user-defined atomic type. The ability to construct a user-defined atomic type was vital for avoiding replication deadlocks in the system. Experience with the construction of semi_version_table suggests that it is not especially difficult to build a user-defined atomic type if it is a modification of an existing user-defined atomic type. However, the testing and debugging of a user-defined atomic type is significantly more complex than the testing and debugging of a conventional, sequential type. Problems in the operation of an atomic type seemed to arise from unanticipated interleavings of events. A tool that could systematically generate different interleavings of events from two or three processes would allow much better testing of user-defined atomic data types.

Argus remote procedure calls were a very useful high-level model for designing and building the system. Instead of a concern with retrying packets, lost messages, sequencing, and so on, the repository designer needs to focus only on arguments provided, results returned, and the exceptions *unavailable* and *failure*. Abstractly, the repository uses replication to try to mask the effects of *unavailable*, and reconfiguration to mask the effects of one kind of *failure*. Only if many copies are unavailable does an operation signal *unavailable*; only if no replacement exists need an operation signal *failure*("guardian does not exist").

The strong typing of Argus, compile-time type-checking, automatic storage management by garbage collection, abstract data types, and all of the other features inherited from CLU [27] made Argus programs easier to construct.

Finally, the Argus debugger made an enormous difference in the construction of the distributed repository. The Argus debugger can be dynamically loaded into or unloaded from a running guardian. The debugger allows the examination and single-stepping of individual processes within a guardian, and uses type information stored outside the guardian to ensure that operation calls made from the debugger are type-safe [38].

6.1.2 Argus Disadvantages

Deadlocks were a frequent problem in dealing with the distributed repository. A large number of the deadlocks in the system were replication deadlocks, and vanished with the implementation of semi_version_table. Another set of deadlocks were simple problems involving such statements as

```
    x := f(x, y)
```

This statement means that x is first read, then written. If x is an atomic object, that means that an action first acquires a read lock on x, then a write lock. Two actions executing this statement in parallel may each succeed in acquiring the read lock, and then deadlock since neither may acquire a write lock. Such deadlocks can be found by inspection and either preceded by an explicit lock acquisition or enclosed in a mutual exclusion construct.

However, eliminating replication deadlocks and simple read-before-write deadlocks did not eliminate all of the deadlocks from the system. In spite of careful but informal attempts to ensure the absence of deadlocks, there were still occasional design errors that showed up as deadlocks. This seems to indicate a weakness in design methods rather than a weakness in Argus, but the language and its tools were not especially helpful in preventing or finding these problems. Only the existence of the Argus debugger allowed the detection and correction of these deadlocks, since debuggers could be loaded dynamically into guardians that seemed to be misbehaving to determine if the cause was a circular chain of locks. A faster method of indicating deadlock or a built-in deadlock resolution system would be useful for developing and debugging systems.

Occasionally a topaction aborts due to a crash, or an unusual chain of exceptions. In such cases, one desirable solution is to retry the topaction involved. However, it is quite awkward to write code that allows retrying of actions. Typically the solution requires either a duplicate topaction in an exception handler, or requires that the topaction be enclosed in an infinite loop. If the topaction completes normally, execution breaks out of the loop; otherwise, the flow around the loop retries the topaction. It is even more awkward to write code that retries once and then does something else. Retrying of a topaction is not a natural paradigm in Argus.

Some difficult-to-debug problems can arise with Argus processes. There is no preemption of processes and no associated priorities, so an infinite loop in one process may cause the entire guardian to halt, since the offending process never gives up the processor. There is no straightforward means of determining the status of a process created explicitly with **fork**, nor any way to determine how many processes are active servicing handler calls. A problem arose with an alien where a large number of incoming mail messages each caused a new process to be **fork**ed to handle the incoming message. These processes bogged down the guardian so much that the external mail system timed out on the delivery, and tried again later — with the same result. Since the guardian had no way of controlling its number of active processes, an overload caused it to fail catastrophically. If the Argus run-time system provided facilities for a guardian to limit its number of active processes, Argus programmers could construct guardians that would be unlikely to thrash or fail under high load.

Expressing replication is generally quite clumsy in Argus. Good software engineering practice [32] suggests that the replication technique should be encapsulated, so that the distributed repository could, for example, switch from the Daniels and Spector algorithm [9] to the Views algorithm [13] for replication. However, such an encapsulation is quite difficult. The replication algorithm is encapsulated in the interface object (Transaction Manager) of the replicated object, but typically the replication algorithm is spread throughout the various operations. There are many features common to the various procedures, but they all have slight differences dependent on whether they read or write replicated data, and what are their arguments, results, and exceptions. Intricate Argus programming can reduce the amount of code involved by writing a limited sort of replication encapsulation. However, the resulting obscurity of the replicator module is perhaps worse than having a lot of simpler code spread throughout the implementation. Conceptually, replication is a "meta-activity", where the programmer describes the replication technique and then provides various procedures or data structures to be automatically replicated. Unfortunately, the construction of an abstract "replicator" module seems impossible in Argus.

6.1.3 Discussion

Argus works well for the problems of constructing reliable distributed programs, which was the original intent of the language. The problems and complexity arise from the use of replication and reconfiguration: while the language was designed to allow such use, it was not designed specifically to support it. Since scalability and availability are two key reasons for a distributed implementation of a system, it is important for a useful distributed programming language to include support for replication and reconfiguration. Incorporating appropriate facilities for replication and reconfiguration into Argus would probably require fundamental changes to the language. It is important to determine what these changes are and how best to deal with replication and reconfiguration.

6.2 Related Work

It is worthwhile to compare the distributed mail repository with two pieces of related work. Grapevine is a distributed mail system that served as an inspiration for parts of the distributed repository, and CES was the first large application implemented in Argus.

6.2.1 Grapevine

Grapevine [5, 40] is a well-known distributed mail system developed at Xerox PARC. Grapevine provides a registry (naming) service and a mail (communication) service, each of which is a client of the other. A message can be delivered to one of a number of mailboxes, but is only delivered to one. Grapevine is a transport service — messages are deleted from the system after an agent has picked them up. The registry is replicated, and updates are propagated in background, so that a change can be made at any replica and that change will gradually make its way through the system. Rather than being consistent, the Grapevine registry information is convergent: if updates are stopped, the system eventually reaches a consistent state.

Grapevine is a well-engineered system; however, much of its design seems intended to avoid having to build a transaction system, with the attendant cost of distributed agreement. In some ways, the distributed mail repository is an exercise in building Grapevine with transactions. The distributed repository provides a repository service instead of a mail transport service; allows for replicated delivery of mail, where a message is actually stored in several mailbox replicas; and prevents anomalies such as adding the same name with two different bindings.

The repository presents a user view that is simpler and more reliable than the corresponding user view of Grapevine. In part, this is because the distributed repository must provide the same interface as an existing centralized system, whereas Grapevine was defined as a distributed system. The transaction system built into Argus makes emulation of a centralized system a reasonable task.

6.2.2 CES

Seliger implemented a Collaborative Editing System (CES) in Argus as a multi-person collaborative editor that allowed different users to cooperate on the creation and modification of a document [41]. CES had a *name server* that corresponded roughly to the repository's registries. The CES name server was replicated for higher availability, using Gifford's weighted voting as the replica control scheme. However, CES considered autonomy of nodes to be more important than the availability of the system. Each author's text was stored on that author's node, and an author could withdraw from collaboration. When an author withdrew, that author's text was no longer accessible to the other authors. Replication of text for availability would reduce this autonomy, by putting copies of text at other nodes where the owning author might not have complete control. The distributed repository does not address issues of autonomy of the nodes, assuming that all of the nodes are conceptually part of a single large system.

CES was an application that was designed to fit its environment, like Grapevine. The distributed repository faced a different set of issues since it had to emulate an existing system.

There was no notion of maintaining availability during a reconfiguration of CES. The configuration techniques that existed were largely static, focusing on building the system. Similarly, there was no attempt to make CES scalable like the distributed repository: there are no provisions for dividing the name server information into multiple name servers.

The implementation of CES was inefficient beyond what might be expected for a prototype system. The distributed repository has focused on efficiency throughout the design process, and Seliger's experience with CES suggested certain design choices that should be avoided. CES used very short topactions, essentially a topaction for each keystroke of an editor. Since each topaction also involved a write to stable storage (itself not an extremely fast operation), this meant that nearly every keystroke of the editor required a significant delay for the write to stable storage. In addition, the system created new guardians to service each new user and destroyed the guardians when the user logged out. In the distributed repository, actions are short but always involve a reasonable amount of computation and one or more remote calls, ensuring that stable storage is not a bottleneck in terms of response time.

6.3 Future Work

There are a number of areas that suggest themselves for future work. One is implementation of other replication techniques in Argus: for example, the Views algorithm [13] or Herlihy's General Quorum Consensus [19]. The distributed repository provides a framework for comparing the techniques in a real application. Unfortunately, as mentioned earlier, the replication technique is intertwined with all of the repository's operations, so that a change of replication technique would be complicated.

Another possible direction for future work is the use of the presented replication and reconfiguration techniques in other applications. It seems likely that these techniques can be used to provide high availability and scalability of an airline reservation system or a banking system, but only the construction of such systems will provide solid evidence.

Finally, future work might include developing new language concepts and constructs for dealing with replication and reconfiguration within the Argus language itself. It seems important to gain more experience with replication and reconfiguration first, and to gain more experience with distributed programs in general. As with the distributed repository, distributed programs need not implement a conspicuously distributed application; instead, a distributed program can implement a centralized application with improved availability and scalability.

Appendix A

DMSP Specification

Distributed Mail System Protocol Specification v3.3

DMSP uses the Unified Stream Protocol (USP). See USP documentation (RFC 272) for description of block transport and data types. The transport protocol used below USP is TCP/IP.

Protocol transactions: "⇒" is agent to repository, "⇐" is repository to agent.

A.1 General Protocol Transactions

Change a password
⇒ SET_PASSWORD[old:string, new:string]
⇐ OK[] | FAILURE[reason:string]

Start debugging facilities at repository
⇒ START_DEBUG[]
⇐ OK[] | FAILURE[reason:string]

Stop debugging at repository
⇒ END_DEBUG[]
⇐ OK[]

Log an informational message at the repository
⇒ LOG_MESSAGE[msg:string]
⇐ OK[] | FAILURE[reason:string]

Compare protocol versions with the repository
⇒ SEND_VERSION[version:cardinal]
⇐ OK[] | FAILURE[reason:string]

Send a message
⇒ SEND_MESSAGE[contents:sequence[string]]
⇐ OK[] | FAILURE[reason:string]

A.2 User Protocol Transactions

Start a connection to the repository.
If "create" is true, then create the named agent.
⇒ LOGIN[user:string, password:string, agent:string, create:boolean]
⇐ FAILURE[reason:string] | OK[] || FORCE_CLIENT_RESET[]

End a connection to the repository
⇒ LOGOUT[]
⇐ OK[]

Create a new user
⇒ CREATE_USER[user:string]
⇐ FAILURE[reason:string] | OK[]

Delete an existing user
⇒ DELETE_USER[user:string]
⇐ FAILURE[reason:string] | OK[]

6.3.1 Agent protocol transactions

Create an agent for the logged-in user
⇒ CREATE_AGENT[agent-name:string]
⇐ OK[] | FAILURE[reason:string]

Delete an agent of the logged-in user
⇒ DELETE_AGENT[agent-name:string]
⇐ OK[] | FAILURE[reason:string]

Reset the specified agent so that it has not seen any changes
⇒ RESET_AGENT[agent-name:string]
⇐ OK[] | FAILURE[reason:string]

List the agent_ids of the logged-in user
⇒ LIST_AGENTS[]
⇐ AGENT_LIST[sequence[
record[name:string,
status:boolean]]]
| FAILURE[reason:string]

A.3 Mailbox Protocol Transactions

Create a mailbox for the logged-in user
⇒ CREATE_MAILBOX[mailbox-name:string]
⇐ OK[] | FAILURE[reason:string]

Delete a mailbox of the logged-in user
⇒ DELETE_MAILBOX[mailbox-name:string]
⇐ OK[] | FAILURE[reason:string]

Permanently destroy all messages marked as deleted in the specified mailbox
⇒ EXPUNGE_MAILBOX[mailbox-name:string]
⇐ OK[] | FAILURE[reason:string]

Reset the mailbox so that no agents appear to have read it
⇒ RESET_MAILBOX[mailbox-name:string]
⇐ OK[] | FAILURE[reason:string]

List all of the logged-in user's mailboxes
⇒ LIST_MAILBOXES[]
⇐ MAILBOX_LIST[sequence[record[
name:string,
next-avail-uid:long-cardinal,
number-of-descs:long-cardinal,
number-of-unseen-descs:long-cardinal]]]
| FAILURE[reason:string]

Create an address for an existing mailbox
⇒ CREATE_ADDRESS[mailbox-name:string, address:string]
⇐ OK[] | FAILURE[reason:string]

Remove an address from a mailbox
⇒ DELETE_ADDRESS[mailbox-name:string, address:string]
⇐ OK[] | FAILURE[reason:string]

List the addresses attached to a mailbox
⇒ LIST_ADDRESSES[mailbox-name:string]
⇐ ADDRESS_LIST[sequence[string]] |
FAILURE[reason:string]

A.4 Message Protocol Transactions

Fetch flags of message
⇒ FETCH_MSG_FLAGS[uid:long-cardinal]
⇐ MESSAGE_FLAGS[flags:sequence[boolean]] |

```
  EXPUNGED[uid:long-cardinal] | FAILURE[reason:string]
```

Fetch several changed descriptors

```
⇒ FETCH_N_CHANGED_DESCS[number-to-send:cardinal]
⇐ sequence[oneof[desc: record[uid:long-cardinal,
                              flags:sequence[boolean],
                              from:string,
                              to:string,
                              date:string,
                              subject:string,
                              chars:long-cardinal,
                              lines:cardinal]
                 expunged: record[uid:long-cardinal]]]
| FAILURE[reason:string]
```

Fetch a particular descriptor by msg_id

```
⇒ FETCH_DESCRIPTOR[uid:long-cardinal]
⇐ DESCRIPTOR[uid:long-cardinal,
             flags:sequence[boolean],
             from:string,
             to:string,
             date:string,
             subject:string,
             nbytes:long-cardinal,
             nlines:long-cardinal] |
  EXPUNGED[uid:long-cardinal] | FAILURE[reason:string]
```

Fetch a particular message by msg_id

```
⇒ FETCH_MESSAGE[uid:long-cardinal]
⇐ EXPUNGED[uid:long-cardinal] | FAILURE[reason:string] |
⇐ MESSAGE[contents:sequence[string]]
```

Copy a message from one mailbox to another

```
⇒ COPY_MESSAGE[target-mailbox:string, source-uid:long-cardinal]
⇐ OK[] | FAILURE[reason:string] | EXPUNGED[uid:long-cardinal]
```

Send a message to a printer

```
⇒ PRINT_MESSAGE[uid:long-cardinal, printer-name:string]
⇐ OK[] | FAILURE[reason:string] | EXPUNGED[uid:long-cardinal]
```

Set one of the boolean flags on a message

```
⇒ SET_MESSAGE_FLAG[uid:long-cardinal, flag-number:cardinal, truth:boolean]
⇐ OK[] | FAILURE[reason:string] | EXPUNGED[uid:long-cardinal]
```

Reset the changed descriptors for this agent

```
⇒ RESET_N_DESCS[number-to-reset:cardinal]
⇐ OK[] | FAILURE[reason:string] | EXPUNGED[uid:long-cardinal]
```

Appendix B

Specifications of Guardians

B.1 Specification of Alien Mailer

```
alien_mailer = guardian is create
                handles deliver

 background
    % The alien mailer periodically checks for new incoming mail
    % and puts it into the appropriate mailbox

    end

 deliver = handler(msg:message) returns(invalid_address_list)
    % Deliver takes a message and attempts to deliver it to
    % the appropriate address(es). For internal mail, it sends
    % directly to the receivers' mailboxes; if an address is
    % intended for internal delivery but does not exist, it is
    % added to the invalid_address_list that is returned.
    %
    % Messages intended for external delivery are sent out of the system;
    % any errors at external mail systems are returned to the sender
    % as incoming messages later.

    end deliver

 end alien_mailer
```

B.2 Specification of Mailsite

```
mailsite = guardian is create
            handles create_mailbox,
                  delete_mailbox,
                  expunge_mailbox,
                  store_message,
                  fetch_message,
                  fetch_descriptor,
                  create_agent,
                  delete_agent,
                  changed_descriptors,
                  reset_updates,
                  get_new_msg_id,
                  next_message_with_flag,
                  set_flag

  create = creator() returns(mailsite)
    % Creates a blank-slate mailsite and returns it to the caller.

    end create

  create_mailbox = handler(name:box_id, agents:list[agent_id])
              signals(mailbox_exists)
    % Creates a new mailbox with the specified id, using agents as
    % initial list of agents with access to mailbox.
    % If mailbox with given name already exists, signal mailbox_exists.

    end create_mailbox

  delete_mailbox = handler(name:box_id) signals(no_such_mailbox)
    % Destroys the mailbox with the given id. If no such mailbox exists
    % at this mailsite, signals no_such_mailbox.

    end delete_mailbox

  expunge_mailbox = handler(name:box_id) signals(no_such_mailbox)
    % If a mailbox with the given id doesn't exist at this mailsite,
    % signals no_such_mailbox.
    % Otherwise, destroys every message which is flagged as "deleted".
    % Once the expunge commits to the top, the messages are
    % permanently deleted.

    end expunge_mailbox

  fetch_descriptor = handler(box:box_id, m:msg_id) returns(descriptor)
              signals(no_such_mailbox, no_such_message, expunged)
    % If a mailbox with the given id doesn't exist at this mailsite,
    % signals no_such_mailbox.
    % If the mailbox exists but no message with the given id has ever been
    % entered into the mailbox, signals no_such_message.
    % If the message has existed but has been expunged, signals expunged.
    % Otherwise, returns a descriptor for the message corresponding to
    % the box_id, msg_id pair.

    end fetch_descriptor

create_agent = handler(box:box_id, agent:agent_id)
               signals(no_such_mailbox, agent_exists)
    % If mailbox doesn't exist, signal no_such_mailbox.
    % If agent already exists, signal agent_exists.
    % Otherwise, create specified agent for specified box.

    end create_agent

delete_agent = handler(box:box_id, agent:agent_id)
               signals(no_such_mailbox, no_such_agent)
    % If mailbox doesn't exist, signal no_such_mailbox.
    % If agent doesn't exist, signal no_such_agent.
    % Otherwise, delete specified agent for specified box.

    end delete_agent

changed_descriptors = handler(box:box_id, agent:agent_id, n:int)
                      returns(list[descriptor])
                      signals(no_such_mailbox,
                              no_such_agent,
                              bounds)
    % If mailbox doesn't exist, signal no_such_mailbox.
    % If agent doesn't exist, signal no_such_agent.
    % If n < 0 then signal bounds
    % Otherwise return list of descriptors of length n or list of all
    % descriptors, whichever is shorter. The candidate descriptors
    % are those which have changed since the last connection by the
    % specified agent.

    end changed_descriptors

reset_updates = handler(box:box_id, agent:agent_id, n:int)
                signals(no_such_mailbox, no_such_agent, bounds)
    % If mailbox doesn't exist, signal no_such_mailbox.
    % If agent doesn't exist, signal no_such_agent.
    % If n < 0 or n > number of descriptors then signal bounds
    % Otherwise reset update flags on first n descriptors
    % so that they do not appear "changed" for this agent.
    % The candidate descriptors
    % are those which have changed since the last connection by the
    % specified agent.

    end reset_updates

fetch_message = handler(box:box_id, m:msg_id) returns(message)
```

```
            signals(no_such_mailbox, no_such_message, expunged)

   % If a mailbox with the given id doesn't exist at this mailsite,
   % signals no_such_mailbox.
   % If the mailbox exists but no message with the given id has ever been
   % entered into the mailbox, signals no_such_message.
   % If the message has existed but has been expunged, signals expunged.
   % Otherwise, returns the message corresponding to the box_id, msg_id
   % pair.

   end fetch_message

send_message = handler(msg:message)
   % Send msg based on its "To:" field(s).

   end send_message

messages_with_flag = handler(box:box_id, flag_num:int)
                     returns(list[msg_id])
                       signals(no_such_mailbox,
                               no_such_flag)
   % If a mailbox with the given id doesn't exist at this mailsite,
   % signals no_such_mailbox.
   % If the flag_num given is not the index of a flag (currently
   % 1 <= flag_num <= 16 ), signals no_such_flag.
   % Otherwise, returns a list of the id's of messages in the specified
   % mailbox with the specified flag "true".

   end next_message_with_flag

set_flag = handler(box:box_id, msg:msg_id, flag_num:int, val:bool)
           signals(no_such_mailbox,
                   no_such_message,
                   no_such_flag,
                   expunged)
   % If a mailbox with the given id doesn't exist at this mailsite,
   % signals no_such_mailbox.
   % If the flag_num given is not the index of a flag (currently
   % 1 <= flag_num <= 16 ), signals no_such_flag.
   % If the mailbox exists but no message with the given id has ever been
   % entered into the mailbox, signals no_such_message.
   % If the message has existed but has been expunged, signals expunged.
   % Otherwise, sets the specified flag of the specified message in the
   % specified mailbox to the value given.

   end set_flag

end mailsite
```

B.3 Specification of Registry

```
% Registry manages everything pertaining to the mailsystem's multi-user state
% not directly connected to messages. It maintains the list of users, the
% list of addresses, and the list of clients (different mail states of users).
% It also provides the mapping from a user-friendly mailbox name to an internal
% mailbox number which can be subsequently used to actually manipulate the
% mailbox. Actual mailboxes reside at mailsites, not at registries.
%

registry = guardian is create,
                handles create_user,
                        delete_user,
                        create_mailbox,
                        delete_mailbox,
                        validate_user,
                        list_mailboxes,
                        get_mailbox_by_name,
                        get_mailbox_by_address,
                        create_address,
                        delete_address,
                        create_agent,
                        delete_agent,
                        list_agents

  create = creator() returns(registry)
    % Normal creator, simply creates a blank-slate registry and
    % returns it to the caller.
    end create

  create_user = handler(name:user_id) signals(user_exists)
    % If the name is already known, signals user_exists.
    % Otherwise, creates new user entry for given name.
    end create_user

  delete_user = handler(name:user_id) signals(no_such_user)
    % Delete_user deletes a name.  If the name is not known,
    % the routine signals no_such_user.
    % Deleting the user removes all associated information,
    % so that no further deletion is required to get rid of it.
    end delete_user

  create_mailbox = handler(u:user_id, b:box_name)
                signals(no_such_user,
                        mailbox_exists)
    % If u not found, signals no_such_user
    % If b is the name of an existing mailbox for this user,
    % signals mailbox_exists.
    % Otherwise creates a mailbox for user "u" with a box_name of "b".
    end create_mailbox
```

```
delete_mailbox = handler(u:user_id, b:box_name)
              signals(no_such_user,
                      no_such_mailbox)
   % If u not found, signals no_such_user
   % If b not found, signals no_such_mailbox
   % Otherwise, removes a mailbox named "b" for a user "u"
   end delete_mailbox

list_mailboxes = handler(u:user_id) returns(list[box_name])
              signals(no_such_user)
   % If u not found, signals no_such_user.
   % Otherwise, returns a list of mailbox names for user u.
   end list_mailboxes

get_mailbox_by_name = handler(u:user_id, b:box_name) returns(dist_mbox)
                  signals (no_such_user, no_such_mailbox)
   % If u not found, signals no_such_user
   % If b not found, signals no_such_mailbox
   % Otherwise, maps a user_id and box_name  to a dist_mbox
   % (mailbox uid + mailsite list)
   end get_mailbox_by_name

get_mailbox_by_address = handler(addr:address) returns(dist_mbox)
                     signals(no_such_address)
   % if addr not found, signals no_such_address
   % Otherwise, maps an address to a dist_mbox (used for delivering
   % mail).
   end get_mailbox_by_address

create_address = handler(addr:address, u:user_id, b:box_name)
              signals(address_exists,
                      no_such_mailbox)
   % if addr already in use, signal address_exists
   % if b not found, signal no_such_mailbox
   % Otherwise, adds addr as an address for a user's mailbox
   end create_address

delete_address = handler(addr:address)
              signals(no_such_address)
   % If addr not found, signals no_such_address
   % Otherwise, removes an address from the system.  This does not
   % delete any other addresses or destroy any mailboxes.
   end delete_address

create_agent = handler(u:user_id, agent:agent_id)
             signals (no_such_user,
                      agent_exists)
   % If u not found, signals no_such_user
   % If agent is already an agent of u, signals agent_exists
   % Otherwise, adds agent to agent_list of user u, creates
   % agent information at mailsite.
```

```
    end create_agent

delete_agent = handler(u:user_id, agent:agent_id)
                 signals(no_such_user,
                         no_such_agent)
    % if u not found, signals no_such_user
    % if agent-not found, signals no_such_agent
    % Otherwise, removes agent from agent_list of user u,
    % removes any agent information from mailboxes at mailsite.
    end delete_agent

list_agents = handler(u:user_id) returns(list[agent_id])
                signals(no_such_user)
    % If user u is not known at this registry, signals no_such_user.
    % Otherwise, returns a list of the names of all of u's agents.
    end list_agents

end registry
```

Appendix C

Semi_Version_Table Implementation

```
% An ordered atomic table abstraction with version numbers.
% Based on semi_table, written by Craig Chambers

semi_version_table = cluster[KEYTYPE, DATATYPE: type]
                        is new, store, fetch, delete,
                           member, gc, equal, transmit

   where KEYTYPE has equal: proctype(KEYTYPE, KEYTYPE) returns(bool),
                     lt: proctype(KEYTYPE, KEYTYPE) returns(bool)

   OPS_BEFORE_GC = 25
   version = int
   rep = mutex[rec]
   rec = record[contents:datalist,
             gaps: gaplist,

             % upper limit for value of operations before gc
             limit: int,

             % number of potentially garbage-producing ops done
             operations: int]

   datalist = array[datathing]
   gaplist = array[gapthing]

   datathing = record[key: KEYTYPE,
                  state: here_gone]
   gapthing = version

   here_gone = atomic_variant[here: info,
                              gone: version]

   info = struct[data: DATATYPE,
              vers: version]
```

```
% Rep invariants:
% r:rep, list:datalist = r.value.contents, gaps:gaplist = r.value.gaps
%      -
% datalist$size(list) ≥ 0
% gaplist$size(gaps) = datalist$size(list) + 1
%
%          datalist$low(list) = 1
%          gaplist$low(gaps) = 0
%
% datalist$low(list) ≤ i < j ≤ datalist$high(list),
%             ⇒
%                list[i].key < list[j].key
%
%
% limit:int := r.value.limit, ops:int := r.value.operations
%
%           limit > 0

% Abstraction function:
%
% Abstract list is a list alternating between data items (each with
% a version number) and gaps (each with a version number).
%
% Gap0  Data1  Gap1  Data2  Gap2  Data3  Gap3  ...
%
% Given a rep object r with list = r.value.contents, gaps = r.value.gaps,
%
% Each concrete data item has a corresponding gap (with the same index).
% The abstract list is given by dividing the list into alternating
% data items and sequences of gone items.  The abstract data
% items and their version numbers are the same as the concrete data
% items; each abstract gap is determined by
% 1. the corresponding gap of the lower data item
% 2. the gone items between the lower and higher items
% 3. the corresponding gap for each of these gone items
% (Exception: the gaps at the beginning and end of the list don't
% have data items on both sides, obviously).
%  The abstract gap has a version number which is the maximum
% of all the version numbers of the concrete gaps and gone items
% making up the gap.
%
```

```
% Implementation note:
%
% Since there are multiple rep gaps in between rep data items, lookups
% in that gap can return more than one version number. However, there
% is no way for a caller to determine whether or not there is a data
% item in the gap (there are no operations describing the size of the
% table or giving all of the elements), so it's OK: abstractly, it looks
% as though there are two gaps with a data item in between, not two gaps.
%

% Implementation note:
%
% WARNING! The gc operation can mutate the arrays containing the
% rep information so that an array index returned by find may not
% be correct after a pause is executed. All operations should either
%   1. Not depend on array positions after a pause, or
%   2. Recalculate array position after a pause.
%

% Operations

new = proc() returns(cvt)
   % create table
   return(rep$create(rec${contents:datalist$new(),
                     gaps:gaplist$[0: 1],
                     limit: OPS_BEFORE_GC,
                     operations: 0}))
   end new
```

```
store = proc(t: cvt, key: KEYTYPE, data: DATATYPE, vers:version)
        signals(not_stored(version))

    % store element in table

    tv:rec
    interest:here_gone

    % Seize table
    seize t do
        tv := t.value
        maybe_gc(tv)

        % add element to table
        pos:int := find(tv, key)
            except when not_found(ins:int, miss:version):

        % insert entry into table
        % if new version is lower than old, ignore and signal
        if vers < miss then signal not_stored(miss) end

        % build object to insert
        memo:here_gone := here_gone$make_gone(1)
        here_gone$change_here(memo, info${data:data,
                                          vers:vers})
        keyed:datathing := datathing${key:key, state:memo}

        % insert new entry in table
        array_insert[datathing](tv.contents, ins, keyed)
        array_insert[version](tv.gaps, ins, miss)

        % write to stable storage
        rep$changed(t)
        return
        end

    interest := tv.contents[pos].state

    % routine continued on next page...
```

```
   % Wait for data to become available.
   while true do

      % Try to get write lock and update data
      tagtest interest
         wtag here(i:info):
            if i.vers < vers then
               here_gone$change_here(interest,
                                     info${data:data,
                                           vers:vers})
               return
            else
               signal not_stored(i.vers)
            end

         wtag gone(v:version):
            if v < vers then
               here_gone$change_here(interest,
                                     info${data:data,
                                           vers:vers})
               return
            else
               signal not_stored(v)
            end
         end

      % Release mutex while waiting for lock
      pause

      end % while
   end % seize
end store
```

```
fetch = proc(t: cvt, key: KEYTYPE) returns(DATATYPE, version)
     signals(not_in_table(version))

  % returns element of table

  tv:rec
  interest:here_gone

  % Seize table
  seize t do
    tv := t.value

    % find element of table
    pos:int := find(tv, key)
      except when not_found(ins:int, miss:version):
        % insert empty entry into table

        % build object to insert
        memo:here_gone := here_gone$make_gone(miss)
        keyed:datathing := datathing${key:key, state:memo}

        % insert new entry in table
        array_insert[datathing](tv.contents, ins, keyed)
        array_insert[gapthing](tv.gaps, ins, miss)

        % write to stable storage
        rep$changed(t)
        signal not_in_table(miss)
        end

    interest := tv.contents[pos].state

    % Wait for data to become available.
    while true do

      % Try to get read lock and update data
      tagtest interest
        tag here(i:info): return(i.data, i.vers)
        tag gone(v:version): signal not_in_table(v)
        end

      % Release mutex while waiting for lock
      pause

      end % while
    end % seize

  end fetch
```

```
delete = proc(t: cvt, key: KEYTYPE, vers: version)
        signals(not_deleted(version))

   % delete element from table

   tv:rec
   interest:here_gone

   % Seize table
   seize t do
      tv := t.value

      % delete element from table
      pos:int := find(tv, key)
         except when not_found(ins:int, miss:version):
            % insert placeholder into table

            % if new version is lower than old, ignore and signal
            if vers < miss then signal not_deleted(miss) end

            % build object to insert
            memo:here_gone := here_gone$make_gone(vers)
            keyed:datathing := datathing${key:key, state:memo}

            % insert new entry in table
            array_insert[datathing](tv.contents, ins, keyed)
            array_insert[gapthing](tv.gaps, ins, miss)

            % write to stable storage
            rep$changed(t)
            return
            end

      interest := tv.contents[pos].state

      % routine continued on next page...
```

```
    % Wait for data to become available
    while true do

       % Try to get write lock and remove data
       tagtest interest
          wtag here(i:info):
             if i.vers < vers then
                here_gone$change_gone(interest, vers)
                return
             else
                signal not_deleted(i.vers)
                end
          wtag gone(v:version):
             if v < vers then
                here_gone$change_gone(interest, vers)
                return
             else
                signal not_deleted(v)
                end
          end

       % Release mutex while waiting for lock
       pause

       end % while
    end % seize
end delete
```

```
member = proc(t: cvt, key: KEYTYPE) returns(bool, version)
   % returns true iff key is in table

   tv:rec
   interest:here_gone

   % Seize table
   seize t do
      tv := t.value

      % find element in table
      pos:int := find(tv, key)
         except when not_found(ins:int, miss:version):
            % insert placeholder into table

            % build object to insert
            memo:here_gone := here_gone$make_gone(miss)
            keyed:datathing := datathing${key:key, state:memo}

            % insert new entry in table
            array_insert[datathing](tv.contents, ins, keyed)
            array_insert[gapthing](tv.gaps, ins, miss)

            % write to stable storage
            rep$changed(t)
            return(false, miss)
            end

      interest := tv.contents[pos].state

      % Wait for data to become available
      % Note that we no longer have possession of the table,
      % just access to the object "interest".
      while true do

      % Try to get read lock on data
      tagtest interest
         tag here(i:info): return(true, i.vers)
         tag gone(v:version): return(false, v)
         end

      % Release mutex while waiting for lock
      pause

      end % while
   end % seize
end member
```

```
gc = proc(t:cvt) signals(gc_failed(string))
   % Garbage collect any aborted bindings
   % This operation should be called at guardian recovery.
   % Otherwise, the garbage should pretty much take
   % care of itself during normal operation.

   % Seize table
   seize t do
      tv:rec := t.value
      contents:datalist := tv.contents
      gaps:gaplist := tv.gaps
      internal_gc(contents, gaps)
         resignal gc_failed
      end

   end gc

equal = proc(t1, t2:cvt) returns(bool)
        where DATATYPE has equal:proctype(DATATYPE, DATATYPE)
                                      returns(bool)
   % returns true iff t1 is the same object as t2
   return(t1 = t2)
   end equal
```

```
% Internal Routines

find = proc(t: rec, key: KEYTYPE) returns(int)
       signals(not_found(int, version))

    % returns position in array of element if found;
    % or signals not_found with the position to insert the element.

    contents:datalist := t.contents
    gaps:gaplist := t.gaps

    low:int := datalist$low(contents)
    high:int := datalist$high(contents)
    middle:int

    % Binary search

    while low < (high - 1) do
        middle := (low+high)/2
        if contents[middle].key = key then
            return(middle)
            elseif contents[middle].key < key then
                % middle is too low, raise search
                low := middle
            else
                % middle is too high, lower search
                high := middle
            end
        end

        % routine continued on next page...
% Fallen through without finding item.
% Either high = low or high = low + 1.
% In all, there are 5 cases to consider:
%
% 1) The item doesn't exist and is just below low
% 2) low points to the item
% 3) The item doesn't exist and should be in the gap between low & high
% 4) high points to the item
% 5) The item doesn't exist and should be in the gap just above high.
%
% Note that in case 1, low never moves from the bottom of the list;
% similarly in case 5, high never moves from the top of the list.

if key < contents[low].key then
   signal not_found(low, gaps[low-1])  % case 1
   elseif key = contents[low].key then
      return(low)                   % case 2
   elseif key < contents[high].key then
      signal not_found(high, gaps[high-1]) % case 3
   elseif key = contents[high].key then
      return(high)                  % case 4
   else
      signal not_found((high + 1), gaps[(high)])   % case 5
   end
     except when bounds:
        % empty list
        signal not_found(low, gaps[low-1])
        end

end find
```

```
internal_gc = proc(contents:datalist, gaps:gaplist)
                   signals(gc_failed(string))

    % Must have already seized the mutex surrounding contents and gaps
    % Get external view of the array
    enter topaction

        % For each binding...
        index:int := datalist$low(contents)
        newvers:int
        while true do
            % If aborted...
            tagtest contents[index].state
            wtag gone(v:version):

                % Get new (highest) version for gap --
                % maximum of ...
                newvers := version$max(
                                      version$max(
                    % the previous gap,...
                    gaps[index - 1],
                    % the deleted object's version,...
                    v),
                    % and the following gap.
                    gaps[index])

                % store new version in gap
                gaps[index - 1] := newvers
                % destroy garbage
                array_delete[datathing](contents, index)
                array_delete[gapthing](gaps, index)

            others:
                % not an aborted binding, move on
                index := index + 1
                end
            end
              except when bounds:
                  % all done
                  end

         end % topaction
           except when unavailable(why:string):
               signal gc_failed(why)
               end

     end internal_gc
```

```
maybe_gc = proc(r:rec)
   % Must already have possession of mutex surrounding r
   % Don't write changes to stable storage.
   r.operations := r.operations + 1
   if r.operations > r.limit then
      internal_gc(r.contents, r.gaps)
      r.operations := 0
      end
   end maybe_gc

% Transmission Routines

encode = proc(t: cvt) returns(rep)
         where KEYTYPE has transmit,
               DATATYPE has transmit
   return(t)
   end encode

decode = proc(t: rep) returns(cvt)
         where KEYTYPE has transmit,
               DATATYPE has transmit
   return(t)
   end decode

end semi_version_table
```

References

[1] Alsberg, P., Belford, G., Day, J., & Grapa, E.
Multi-Copy Resiliency Techniques.
CAC Document 202, Center for Advanced Computation, University of Illinois, May, 1976.

[2] Balkovich, E., Lerman, S., & Parmelee, R.P.
Computing in Higher Education: The Athena Experience.
Communications of the ACM 28(11):1214-1224, November, 1985.

[3] Bernstein, P.A. and Goodman, N.
Multiversion Concurrency Control — Theory and Algorithms.
ACM Transactions on Database Systems 8(4):465-483, December, 1983.

[4] Bernstein, P.A. and Goodman, N.
An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases.
ACM Transactions on Database Systems 9(4):596-615, December, 1984.

[5] Birrell, A.D., Levin, R., Needham, R.M., & Schroeder, M.D.
Grapevine: An Exercise in Distributed Computing.
Communications of the ACM 25(4):260-274, April, 1982.

[6] Bloom, T.
Dynamic Module Replacement in a Distributed Programming System.
Technical Report MIT/LCS/TR-303, MIT Laboratory for Computer Science, March, 1983.

[7] Clark, D.D. and Lambert, M.L.
PCMAIL: A Distributed Mail System for Personal Computers.
RFC 984, SRI Network Information Center, May, 1986.

[8] Crocker, D.H.
Standard for the Format of ARPA Internet Text Messages.
RFC 822, SRI Network Information Center, August, 1982.

[9] Daniels, D. and Spector, A.Z.
An Algorithm for Replicated Directories.
Technical Report CMU-CS-83-123, Department of Computer Science, Carnegie-Mellon University, May, 1983.

[10] Eager, D.L. and Sevcik, K.C.
Achieving Robustness in Distributed Database Systems.
ACM Transactions on Database Systems 8(3):354-381, September, 1983.

[11] Ein-Dor, P.
Grosch's Law Re-Revisited: CPU Power and the Cost of Computation.
Communications of the ACM 28(2):142-151, February, 1985.

[12] El Abbadi, A., Skeen, D., & Cristian, F.
An Efficient, Fault-Tolerant Protocol for Replicated Data Management.
In *4th ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems.* 1985.

[13] El Abbadi, A. & Toueg, S.
Maintaining Availability in Partitioned Replicated Databases.
In *Proceedings of the 5th ACM SIGACT/SIGMOD Conference on Principles of Database Systems*. March, 1986.

[14] Eswaran, K.P., Gray, J.N., Lorie, R.A., & Traiger, I.L.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-633, November, 1976.

[15] Gifford, D.K.
Violet, An Experimental Decentralized System.
Technical Report CSL-79-12, Xerox Palo Alto Research Center, 1979.

[16] Gifford, D.K.
Weighted Voting for Replicated Data.
In *Proceedings of the 7th ACM Symposium on Operating System Principles*. December, 1979.

[17] Gifford, D. and Spector, A.
The TWA Reservation System.
Communications of the ACM 27(7):650-665, July, 1984.

[18] Herlihy, M. and Liskov, B.
A Value Transmission Method for Abstract Data Types.
ACM Transactions on Programming Languages and Systems 4(4):527-551, October, 1982.

[19] Herlihy, M.P.
Replication Methods for Abstract Data Types.
Technical Report MIT/LCS/TR-319, MIT Laboratory for Computer Science, May, 1984.

[20] Herlihy, M.
A Quorum-Consensus Replication Method for Abstract Data Types.
ACM Transactions on Computer Systems 4(1):32-53, February, 1986.

[21] Hwang, D.J.
Constructing Highly-Available Services in a Distributed Environment.
October, 1986.
S.M. Thesis Proposal, MIT EECS.

[22] Isloor, S.S. & Marsland, T.A.
The Deadlock Problem: An Overview.
IEEE Computer 13(9):58-70, September, 1980.

[23] Johnson, P.R., and Thomas, R.H.
The Maintenance of Duplicate Databases.
RFC 677, SRI Network Information Center, January, 1975.

[24] Joseph, T.A. and Birman, K.P.
Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems.
ACM Transactions on Computer Systems 4(1):54-70, February, 1986.

[25] Kramer, J. & Magee, J.
Dynamic Configuration for Distributed Systems.
IEEE Transactions on Software Engineering SE-11(4):424-436, April, 1985.

[26] Lampson, B.W., and Sturgis, H.E.
Crash Recovery in a Distributed Data Storage System.
Technical Report, Xerox Palo Alto Research Center, 1979.

[27] Liskov, B. *et al.*.
CLU Reference Manual.
Springer-Verlag, 1981.

[28] Liskov, B. and Scheifler, R.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[29] Liskov, B. *et al.*
Preliminary Argus Reference Manual.
Programming Methodology Group Memo 39, MIT Laboratory for Computer Science, October, 1983.

[30] Liskov, B. & Weihl, W.
Specifications of Distributed Programs.
Programming Methodology Group Memo 46, MIT Laboratory for Computer Science, September, 1985.

[31] Liskov, B., and Ladin, R.
Highly-Available Services and Fault-Tolerant Distributed Garbage Collection.
In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing.* August, 1986.

[32] Liskov, B. and Guttag, J.
Abstraction and Specification in Program Development.
MIT Press, 1986.

[33] Minoura, T. and Wiederhold, G.
Resilient Extended True-Copy Token Scheme for a Distributed Database System.
IEEE Transactions on Software Engineering SE-8(3):173-188, May, 1982.

[34] Moss, J.E.B.
Nested Transactions: An Approach to Reliable Distributed Computing.
MIT Press, 1985.

[35] Parker, Jr., D.S., *et al.*.
Detection of Mutual Inconsistency in Distributed Systems.
IEEE Transactions on Software Engineering SE-9(3):240-247, May, 1983.

[36] Postel, J.B.
Simple Mail Transfer Protocol.
RFC 821, SRI Network Information Center, November, 1982.

[37] Pu, C., Noe, J.D., and Proudfoot, A.
Regeneration of Replicated Objects: A Technique for Increased Availability.
Technical Report TR-85-04-02, University of Washington Department of Computer Science, April, 1985.

[38] Restivo, J.P.
Adding Type Information to the Argus Debugging System.
Master's thesis, Massachusetts Institute of Technology, May, 1985.

[39] **Scheifler, R.**
Private Communication.
July, 1985.

[40] Schroeder, M.D., Birrell, A.D., and Needham, R.M.
Experience with Grapevine: The Growth of a Distributed System.
ACM Transactions on Computer Systems 2(1):3-23, February, 1984.

[41] Seliger, R.
Design and Implementation of a Distributed Program for Collaborative Editing.
Master's thesis, Massachusetts Institute of Technology, September, 1985.

[42] Stonebraker, M.
Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES.
IEEE Transactions on Software Engineering SE-5(3):188-194, May, 1979.

[43] Thomas, R.H.
A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases.
ACM Transactions on Database Systems 4(2):180-209, June, 1979.

[44] Weihl, W. and Liskov, B.
Implementation of Resilient, Atomic Data Types.
ACM Transactions on Programming Languages and Systems 7(2):244-269, April, 1985.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-376	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125

6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139		7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Replication and Reconfiguration in a Distributed Mail Repository

12. PERSONAL AUTHOR(S)
Day, Mark S.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	March 1987	110

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Data replication, software reconfiguration, availability, reliability, scalable systems, distributed programs, electronic mail repositories, programming languages

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*
Conventional approaches to programming produce centralized programs that run on a single computer. However, an unconventional approach can take advantage of low-cost communication and small, inexpensive computers. A distributed program provides service through programs executing at several nodes of a distributed system. Distributed programs can offer two important advantages over centralized programs: high availability and scalability. In a highly-available system, it is very likely that a randomly-chosen transaction will complete successfully. A scalable system's capacity can be increased or decreased to match changes in the demands placed on the system.

When a node is unavailable because of maintenance or a crash, transactions may fail unless copies of the node's information are stored at other nodes. Thus, high availability requires replication of data. Both the maintenance of a highly-available system and scalability require the ability to modify and extend a system while it is

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little	22b. TELEPHONE *(Include Area Code)* (617)253-5894 — 22c. OFFICE SYMBOL

19. running, called dynamic reconfiguration or simply reconfiguration.

This thesis considers the problem of building scalable and highly-available distributed programs without using special processors with redundant hardware and software. It describes a design and implementation of an example distributed program, an electronic mail repository. The thesis focuses on how to design and implement replication and reconfiguration for the distributed mail repository, considering these questions in the context of the programming language Argus, which was designed to support distributed programming.

The thesis makes three distinct contributions. First, it presents the replication techniques chosen for the distributed repository and a discussion of their implementation in Argus. Second, it describes a new method for designing and implementing reconfigurable distributed systems. The new method allows replacement of software components while preserving their state, but requires no changes to the underlying system or language. This contrasts with previous work on guardian replacement of Argus. Third, the thesis evaluates the utility of Argus for applications involving replication and reconfiguration.