T 1229
P 18466

MIT/LCS/TR-350

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED PROGRAM FOR COLLABORATIVE EDITING

Robert Seliger

January 1986

Design and Implementation of a Distributed Program for Collaborative Editing

by

Robert Seliger

Submitted to the
Department of Electrical Engineering and Computer Science
on September 30, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

This thesis presents the design and implementation of a distributed program for the support of multi-author collaboration on shared documents. The Collaborative Editing System, CES, provides an environment in which authors working on a document can cooperate and coordinate their individual contributions to a single document. Authors are provided with a dynamic view of the document being edited so that the contributions made by other authors are shared, virtually as they are made. Features of the system include:

- No inherent limitation on the geographical locale of collaborating authors.
- No inherent limitation on the size of a document's author population; similarly, no inherent limitation on the number of authors that can concurrently edit a document.
- Simple protocols for obtaining exclusive access to portions of a document.
- A user interface specifically designed to accommodate both a private workspace and shared views of a document.
- Facilities for document critiquing and altering document viewing perspectives.

- Support for the distributed storage of a single document such that individual contributions are maintained on the contributing author's host computer.

- Ability to make documents available for public viewing or editing.

CES is comprised of three subsystems: the document registry, the document library, and the editor front-end. The subsystems, which differ in functionality and in their requirements for responsiveness, resiliency, and long-livedness of data, are implemented in the programming language Argus. Argus is designed to facilitate the development of distributed programs. The language has features that address many of the subsystems' requirements while providing a supportive development environment. Analysis of the CES implementation includes a review of Argus and a critique of the language's appropriateness for the development of interactive applications.

Keywords: distributed computing systems, distributed programming, distributed applications, replicated data, concurrency control, collaborative editing.

Acknowledgments

It is difficult to know who to thank first for helping to make this year at M.I.T. possible. I would like to thank the folks at Hewlett-Packard, Corporate, who instituted the Hewlett-Packard Resident Fellowship program. Without this program, I would most certainly not have had such uncompromising support for the pursuit of graduate studies. I would also like to thank the folks at Hewlett-Packard, Waltham, who made the Fellowship program a reality for me. These people include Jean Barron, Tim Blancke, Fred Campbell, George Elenbaas, Paul Goldman, Jack Harrington, Ben Holmes, Drew Koschek, Mike Kramer, Al Nardizzi, Stu Pearce, Russ Peschier, Ron Rankin, and Susan Simmons. I would also like to thank Judi Carmichael, Bob Femino, and Carl Walsh, who reminded me that I had the resources of Hewlett-Packard behind me.

Charles Fendrock and Professor William Heetderks both also deserve my thanks for graciously writing recommendations on my behalf to M.I.T.

Getting accepted by the Fellowship program and then by M.I.T. were only part of the challenge. The intense year of academics and research would not have nearly as rewarding were it not for the guidance, help, and friendship of so many at M.I.T. I would like to thank Professor John Guttag for teaching me how to think, M.I.T. style. I would also like to thank fellow students Mark Day, Jim Gibson, Gary Leavens, Tony Ng, and Mark Tuttle, whose insights, wit, and support have been most appreciated. Also, Sean Routhier was always there to help out with the latest crash and disk problems. In addition, a special thanks and acknowledgment are deserved by the Argus designers, including Barbara Liskov, Bill Weihl, Paul Johnson, and Bob Scheifler. It is unlikely that CES would ever have been implemented as extensively as it was without Argus.

Of course, the focal point of the past year has been this thesis. From its inception to its completion, Irene Greif, my research advisor, has been supportive, patient, full of ideas, and most importantly, enthusiastic about my work. The successful completion of this thesis would not have been possible were it not for her contributions and concern.

Finally, I would like to thank my wife, Elaine. This past year together has been wonderful.

Table of Contents

Chapter One: Introduction 13

1.1 Cooperative Work 13
1.2 Distributed Application Issues 15
1.3 Argus 17
1.3.1 Distributed Application Paradigms 18
1.3.2 Linguistic and System Support 19
1.4 Goals 21
1.5 Related Work 22
1.5.1 ZOG 23
1.5.2 Augment 24
1.5.3 Textnet 26
1.5.4 JEDI 27
1.5.5 Grapevine 29
1.6 Thesis Organization 30

Chapter Two: Collaborative Editing System Design 31

2.1 Structuring of Documents 33
2.2 User Interface 35
2.2.1 Display 35
2.2.2 User Input 37
2.3 Fundamentals of Editing a Document 38
2.3.1 Creating a Document 39
2.3.2 Viewing a Document 39
2.3.3 Modifying a Document's Structure 43
2.3.3.1 Outline Parser 43
2.3.3.2 Editing an Outline 46
2.3.4 Modifying a Document's Contents 47
2.3.5 Maintaining Consistency Among Views 49
2.3.6 Logging and Saving Changes 49
2.3.7 Orphan Nodes 51
2.3.8 Altering Document View Perspectives 52
2.3.9 Document Critiquing 53
2.4 Support for Collaboration 55
2.4.1 Creating a Document 56
2.4.2 Accessing an Existing Document 57

2.4.3 Modifying a Document's Contents 58
2.4.3.1 Obtaining Exclusive Access 58
2.4.3.2 Maintaining Exclusive Access 59
2.4.3.3 Sharing Changes 60
2.4.4 Modifying a Document's Structure 61
2.4.4.1 Editing an Outline 61
2.4.4.2 Parsing an Outline 62
2.4.5 Orphan Nodes 63
2.4.6 Document-Wide Edits 63
2.4.7 Observations 64
2.5 CES Architecture 65
2.5.1 Document Registry 67
2.5.1.1 Function 67
2.5.1.2 Performance Requirements 69
2.5.2 Document Library 70
2.5.2.1 Function 70
2.5.2.2 Performance Requirements 72
2.5.3 Editor Front-End 73
2.5.3.1 Function 73
2.5.3.2 Performance Requirements 74
2.5.4 Observations 75

Chapter Three: The Programming Language Argus 76

3.1 Fundamental Linguistic Constructs 77
3.1.1 Basic Abstraction Facilities 77
3.1.2 Guardians 78
3.1.3 Atomic Actions 80
3.1.4 Atomic Objects 82
3.1.5 User Defined Types 84
3.2 Using Argus 85
3.2.1 Guardians 85
3.2.2 Atomic Actions 87
3.2.2.1 Remote Procedure Calls 87
3.2.2.2 Specifying Concurrent Actions 88
3.2.2.3 When to Use Concurrent Actions 89
3.2.2.4 Managing Action Structures 90
3.2.2.5 Short Actions 91
3.2.3 User Defined Types 93
3.2.3.1 When to Use a User Defined Type 93
3.2.3.2 Increasing Concurrency 94
3.2.4 Summary 97

Chapter Four: Collaborative Editing System Implementation 99

4.1 Implementation Overview 99
4.2 Document Registry 101
4.2.1 Managing the Document List 101
4.2.1.1 Data Structures 101
4.2.1.2 Handlers, Background, Recovery, and Action Structures 104
4.2.2 Managing Document Registry Replicands 105
4.2.2.1 Gifford's Weighted Voting Algorithm 105
4.2.2.2 Data Structures 107
4.2.2.3 Creating Guardian Replicands 107
4.2.2.4 Obtaining a Quorum 114
4.2.2.5 Avoiding Deadlock 115
4.3 Document Library 116
4.3.1 Document Node Catalog 117
4.3.1.1 Basic Data Structures 118
4.3.1.2 Version Stack 120
4.3.1.3 Tickle-Lock 123
4.3.1.4 Handlers and Action Structures 125
4.3.1.5 Background and Recovery 133
4.3.2 Document Structure Catalog 134
4.3.2.1 Distributed Tickle-Lock 134
4.3.2.2 Coordinating Updates 135
4.4 Editor Front-End 136
4.4.1 I/O Guardian 137
4.4.1.1 Emulating and Atomic Display 138
4.4.2 Editor Front-End Guardian 140
4.4.2.1 Basic Data Structures 141
4.4.2.2 Action Structure 141
4.5 CES System Catalog 143
4.6 Observations about the Role of Argus in the CES Implementation 145
4.6.1 Argus Linguistic Support 146
4.6.2 Argus Implementation Subtleties 149

Chapter Five: Summary and Conclusions 153

5.1 Summary and Conclusions 153
5.1.1 Functionality 153
5.1.2 Architecture and Implementation 155
5.1.3 Conclusions 156
5.2 Extensions 159
5.3 Future Work 161
5.3.1 Cooperative Applications 161

5.3.2 CES Performance 162
5.4 Observations 163

Table of Figures

Figure 2-1: Layout of CES User Interface Display 36
Figure 2-2: Viewing a Document with Outline Status and Node Demarcations 41
Figure 2-3: Viewing a Document without Outline Status or Text Image Node Demarcations 42
Figure 2-4: Example of an Outline Item 44
Figure 2-5: Examples of Unparsed Outline Indentation 45
Figure 2-6: Outlines from Figure 2-5 After Parsing 46
Figure 2-7: Example of Node Copying 48
Figure 2-8: Example of Clipping a Document View Perspective 53
Figure 2-9: Example of Critique Nodes 54
Figure 3-1: Example of an Action Structure using Coenter 89
Figure 3-2: Example of an Action Structure using Coenter and Iterator 90
Figure 4-1: Specification of Registry and Related Types 104
Figure 4-2: Specification of Reg_status and Reg_list Types 107
Figure 4-3: Informal Specification of Document Registry Handlers 108
Figure 4-4: Chart of Quorum Sizes in Document Registry Guardian 114
Figure 4-5: Specification of Node_catalog and Related Types 119
Figure 4-6: Specification of Astack Representation 125
Figure 4-7: Informal Specification of Astack Cluster 126
Figure 4-8: Informal Specification of Node Catalog Handlers 128

Chapter One

Introduction

This thesis presents the design and implementation of the Collaborative Editing System, CES. CES is a distributed application that supports the cooperative work of a possibly geographically separated community of authors working together towards the common goal of writing a document. With CES, authors can concurrently work on a shared document and make individual contributions without disturbing the logical integrity of the document. A key feature of the system is that changes made by an author to a document are shared in a timely fashion with the document's other authors, thereby providing a context for synergism and the exchange of ideas.

1.1 Cooperative Work

A system that supports the work of a cooperating community of users is called a *cooperative* system. Underscoring the design of a cooperative system is an assumption about the behavior of the community that is being supported, and an assumption about the kind of support that a community desires. For example, consider a cooperative system that supports a real-time conferencing session in which a community of users electronically converse using keyboard input from physically separate terminals that display a shared work-space. The basic problem is how to allow users to converse so that inputs do not interleave, thereby becoming unintelligible. If it is assumed that the users are all polite, then it might be reasonable to simply allow them to recognize interleavings and resolve the problem themselves. If it assumed that the users are polite, but that

they would like automatic assistance to prevent the interleaving of inputs, then the system can require that users request permission to provide an input. If it is assumed that users are extremely opinionated and forceful, then the system might not only require that users request permission to provide input, but it might also limit the amount of time that a user has to do so. The nuances of these assumptions are the basis for categorizing cooperative systems as follows:

- Purely Cooperative

  Purely cooperative systems are targeted for communities in which the goals of each member of the community are the same as the overall goals of the community. A purely cooperative system is the least restrictive of the three categories because such systems assume that community members want a high degree of interaction and will always benefit by the effects of each member's work.

- Coordinating

  Coordinating systems are targeted for communities where the goals of different community members may be at conflict with one another, though each community member is working towards common community goals. A coordinating system is the most restrictive of the three categories because such systems assume that the activities of group members need to be tightly controlled to minimize destructive interference.

- Collaborative

  Collaborative systems are hybrids of cooperative and coordinating systems, and provide members of a community with enough flexibility to encourage willful cooperation, while offering restrictions in order in order to minimize destructive interference.

CES is designed to be a collaborative system. CES assumes that a community of authors using the system do so because they want to willfully cooperate on the joint authoring of a document. CES also assumes that the authors want to minimize the amount of manual management required to coordinate authors' activities. The assumption is that authors would always rather have access to a

document with the assistance of automated coordination, instead of having to pass drafts of a document around for review (either on paper or electronically). Therefore, authors will always be able to read and edit the latest draft of a document (and not have to wait until it is passed around). In conjunction with this flexibility, authors are expected to respect the contributions made by other authors (and not "unedit" them) and assist the system in resolving conflicts that the system cannot resolve on its own.

1.2 Distributed Application Issues

Applications designed for a distributed system offer a realm of possibilities that would most likely be extremely uneconomical, complicated, or unimplementable if designed for a centralized system. Consider the case in which a system allows geographically separated users to work together on a common project. Though it is conceivable that each user's terminal could be connected directly to a centralized system, such a system would be hardwired and difficult to expand and reconfigure. (For an example of such a system, see [Gifford & Spector 84].) In addition, such a configuration does not adapt to individual processing demands by providing local services locally. In contrast, a distributed system facilitates placement of processing resources where they are required, yet allows resources and data to be shared. Also, a distributed system offers features and behaviors which may not be possible with a centralized system. These include:

- Increased instruction throughput, achieved by performing computations in parallel on several computers.
- Resiliency to failures through the replication of data on redundant hardware.
- Extremely large data spaces which are a function of system size, and which are not restricted by the addressing capabilities of an individual machine.

- Support for *locality of ownership*, achieved by allowing system resources, and the data serviced by the resources, to physically reside at the owner's facility.

The challenge faced by designers of distributed applications is how to incorporate the features and benefits listed above into the application. This is both a specification and implementation concern, and poses the following questions:

- Should the application mask its distributed nature?

  Masking the distributed nature of an application gives users the impression that they are working in a centralized environment. A distributed resource manager is an example of this kind of application, where a client is only concerned that a task be performed and is not interested in how it is done. Alternatively, the distributed nature of an application can be made apparent to the user. An electronic mail system is an example of this kind of application, where clients explicitly use the mail system for communication across physical expanses.

- How should the concurrency introduced by parallel computation be exploited?

  An application can be designed so that activities are rapidly executed in a serial fashion. Alternatively, an application can be designed to accommodate the concurrent execution of several activities, with the possible cost of delaying the rate at which activities are completed.

- How critical is the resiliency of the application to failures?

  The primary motivation for designing a distributed application may be to take advantage of data replication on redundant hardware in order to create an application that can tolerate failures. In such applications, the cost of down-time may be substantial, thereby warranting special measures to minimize the impact of failures. In other applications, the hardware and performance costs of replicating data may be too expensive compared to the cost of losing the application's services during failures. With these applications, periodic back-ups may be the preferable alternative for preserving work performed by the application. Some applications may lie in the middle of these extremes and provide limited functionality during

failures.

- How does the application name data in an extremely large, distributed name space?

  If the names are embedded in the application, then the application may be sensitive to underlying system reconfigurations that cause names to change. Alternatively, the underlying system may provide a lookup service in order to associate logical names with physical names, thereby buffering the application from system reconfigurations. In addition, the application may have to reveal the naming mechanism to the user. If this is the case, then the application may be forced to reveal it distributed nature.

- Is there any political, proprietary, or administrative reason to support the locality of ownership?

  Many distributed applications are specifically designed to facilitate the sharing of data, but the accessibility of data in such systems can violate notions of ownership. This concern is further accentuated by the loss of autonomous control of data when data is replicated on redundant hardware for resiliency. Consequently, measures may have to be taken to provide the owners of data with the ultimate ability to restrict or prevent access to the data.

The CES design either implicitly or explicitly answers each one of these questions.

1.3 Argus

Even if distributed applications designers are able to easily answer the specification concerns itemized above, implementing the application is typically difficult and time consuming. This is because even though technology has made it feasible to build distributed systems, the tools for implementing distributed applications have tended to remain in the domain of operating systems. This has meant that truly distributed applications have been tightly coupled with lower level mechanisms, making the process of implementing such applications

extremely complicated.

Recently, the programming language Argus [Liskov *et al.* 83], conceived as a language for the implementation of distributed applications, matured to the point where it is a usable tool. By attempting to suppress low level issues, Argus is intended to allow applications designers to construct programs which can take full advantage of the features of a distributed environment, without getting overwhelmed by underlying details. The implementation of CES is the first use of Argus as the development tool for a novel, truly distributed, application.

1.3.1 Distributed Application Paradigms

There are a several paradigms of distributed systems applications. On one extreme are systems designed primarily for speed. These systems are intended to offer high speed performance by exploiting parallelism in algorithmic processing. On the other extreme are real-time systems designed primarily for process control. These systems rely upon dedicated processors to provide rapid response to stimuli from the system environment. In addition, these systems often manipulate replication of processor responsibilities to enhance system reliability in the event of an individual processor failure. In some aspects, the applications for which Argus is intended lie between these extremes. Argus offers programmers the opportunity to use parallelism to increase computational throughput, while developing system configurations which provide behaviors not possible with a centralized system, particularly with regard to resiliency to failures.

Distributed application paradigms can be further classified by the roles of the underlying system's component physical processors. In a system designed for speed, the processor roles tend to be extremely similar. Each processor is part of a larger whole, so the overall system can be thought of as a network of co-

processors, each performing some aspect of a computation. In a real-time system, the processor roles tend to be autonomous - they communicate perhaps only when attempting to detect a failure. Again, Argus aims at a middle ground by supporting a processor relationship that is commonly referred to as *client/server.* The client/server paradigm assumes that system work is accomplished through communication between autonomous processors. Typically, the role of a processor is not fixed as client or server, but varies depending upon which processor seeks information and which processor has the information. In the client/server paradigm, the sharing of data is explicit.

Argus further refines the client/server model to include the fact that not only is data shared, but data is expected to be long-lived. This additional consideration has substantial implications, because it constrains the Argus system to be able to consistently preserve data. This means that data must not only remain consistent as it is processed by a number of independent, communicating processors, but that it must remain consistent in the event of a failure. To this end, Argus offers mechanisms which assure data consistency during distributed computations, and provide, with a probability commensurate with the cost of implementing the Argus system, data consistency in the event of a failure.

1.3.2 Linguistic and System Support

Argus linguistic and system support facilitates the software implementation of a distributed application built as a virtual network of virtual processing nodes, called *guardians.* Installed on top of an actual hardware configuration, the network of guardians is an abstraction of the underlying hardware. A guardian, acting as a virtual processor, is capable of communicating with other guardians as well as performing local multi-processing chores. In addition, a guardian encapsulates state information which is accessible only by the guardian, but which can be made available to other guardians through inter-guardian

communication. The result is that a guardian is a centralized resource that, by communicating with other guardians, becomes a component in a distributed system.

In order to accommodate parallel computations, Argus explicitly supports concurrent processing. The usual concerns which are involved with concurrent processing, such as process synchronization, are further complicated in a distributed domain where processes span computers and data is long-lived. In order to synchronize processes and ensure data consistency across computations and failures, Argus uses *atomic actions*. An atomic action is an action which is *serializable* and *recoverable* [Weihl 84]. Serializability means that the effects of the potentially interleaved component steps of concurrently executed atomic actions are the same as if each action was executed in some serial order. Recoverability means that either all of the effects of an atomic action are made visible to other actions, or none of the effects are visible. Thus, the effects of an atomic action are indivisible. An action that runs to completion and makes its effects known is said to have *committed*. An action which is not able to successfully complete all of its steps undoes the effects of any completed steps by *aborting*.

By allowing actions to span guardians, Argus provides a mechanism for encapsulating the activities of individual processors into a single logical activity. As an atomic action, inter-processor activity exhibits atomic behavior, so the failure of a guardian (perhaps due to an underlying hardware failure) simply causes actions running at the guardian to abort. In order to structure the flow of actions and to modularize the scope of an action, Argus provides primitives for developing hierarchies of parent actions and associated children subactions. Work in the system is thereby accomplished by the concurrent processing of various threads of control, each rooted in a parent action, that spread as actions throughout the system. Guardians are capable of multiprocessing, so an

individual guardian can be involved with a number of related and unrelated actions. These actions are serializable and recoverable, so data consistency is maintained during active processing of actions. In addition, elements of the guardian state can be made stable so that a guardian can recover its state after a failure. This is achieved by keeping stable copies of the state data on highly reliable storage devices. Consequently, the guardian state can be maintained across failures as long as the failure does not involve a stable storage device.

In summary, Argus provides the linguistic and system support required to build distributable entities (using guardians), to perform concurrent computations (using actions), and to make applications resilient to failures (using stable storage).

1.4 Goals

The primary goal of the CES project is to design and implement a functional prototype of a collaborative editing system. The resulting system should be more than just an otherwise centralized application that has been built in a distributed environment. In order to do this, the design will attempt to unify techniques for designing distributed systems to produce an application that takes full advantage of its distributed nature. In addition, the CES design will opt for alternatives that help make users aware that they are individuals participating in a collaborative effort. Unlike the distributed resource manager example mentioned earlier, which is specifically designed to let users work in isolation, the CES design should promote ways for groups of users to work cooperatively.

There are three subordinate goals which must be confronted in order to reach the primary goal:

- Design the CES application level functionality.

This design should specify the user interface, the functionality of document editing chores that do not involve collaboration, and in particular, the functionality of document editing chores that do involve collaboration.

- Engineer the aspects of CES that concern the issues of system responsiveness, resiliency, locality of ownership, and configurability.

  It is with respect to this goal that a firm understanding of Argus is critical, so that the language can be used to its fullest potential. To increase system responsiveness, the design will use highly concurrent data types and an action structure that takes advantage of the resulting concurrency. To achieve resiliency, methods which enhance the availability of the system, such as the replication of critical resources, will be employed. Also, Argus stable storage will be judiciously used to ensure that critical data survives failures. To preserve locality of ownership, a system architecture that provides the desired overall system behavior without sacrificing ownership concerns will be designed. Finally, a scheme for configuring and maintaining the system will be proposed and implemented.

- Explore the viability of CES by making the prototype of the system available to a community of users.

  This will be done not only to identify and correct implementation errors, but will be particularly useful for evaluating many of the CES design decisions. It has been stated that CES does not shelter its users from each other's activities. This decision has the potential for making CES a useful, powerful, cooperative service, or could result in confusing distractions and information overload for the user. Unless the system is actually used, gauging the effectiveness of the various design decisions will be a speculative process.

1.5 Related Work

There are three areas of related work upon which this thesis draws - structured text editors, cooperative systems, and distributed systems and applications design. Examples of structured text editors include Zog [Akscyn & McCracken 84], Augment [Englebart 82] and Textnet [Trigg 83]. Sarin's proposed JEDI

system [Sarin 84] is an example of a cooperative system. An example of a distributed application is Grapevine [Birrell *et al.* 82].

1.5.1 ZOG

ZOG is a distributed data base system built to offer high performance access to data. The system is implemented with enough generality to support structured text editing as well as more conventional data base activities. A ZOG data base, or *net*, is a network of objects, called *frames*, that can be linked together in tree-like hierarchies to form a single, logical structure. Each frame has a number of fields. These range from title and text fields to frame-links fields. The frame abstraction was designed to serve a variety of uses, with the result that simplicity is achieved by requiring both the system and the user to deal with only one object type.

ZOG is designed to be an information representation system. By *navigating* through ZOG nets, a user can view, frame by frame, the net contents. In addition, various functions can be performed on and by the frame being viewed, such as invoking a program, altering the frame's contents, or creating links to other or new frames. The ability for users to be able to readily access a frame and alter its contents, or alternatively modify the net, is one of the primary reasons for ZOG's use as a documentation tool. When ZOG frames are used to encapsulate the blocks of text for a larger entity, the result is a structured document. The net provides structure and the frames provide content. In this regard, ZOG fits the description of a structural editor.

One potential shortcoming of ZOG is that the system apparently does not allow the user to isolate and view the net structure. Through navigation, the user can traverse the structure, but it is not clear whether it is possible to create a single schematic view of the net. This limitation would seem to complicate the task of

designing and maintaining a net for a document. Another potential shortcoming is that the user cannot interactively view the frames of a document in a cohesive fashion. Frames are viewed individually, and there is no mechanism, except by producing hard-copy, that allows the user to display clusters of frames. Because the user is not able to display neighboring frames simultaneously, a mental image of the document must be maintained in order to provide context for the frame being viewed. Preserving context is also of concern when a user is creating a new frame. For example, the flexibility that ZOG offers in allowing the user to link new frames to existing frames facilitates such activities as critiquing. A critique is not part of the document proper, but is nevertheless linked into the document subordinate to the frame being critiqued. It is presumably more difficult to critique from memory than it is when the text being critiqued is being viewed as the critique is written.

1.5.2 Augment

Augment is a structured text editor. Augment documents are represented as a file composed of a hierarchical collection of *nodes*. Each node encapsulates a block of text, graphical data, or other data. The structure of a document is explicitly represented by the organization of the nodes, thereby providing a framework in which an author can design, modify, and manipulate a document based on structural considerations as well as content. In this regard, Augment provides a host of capabilities and features that distinguish it from more common unstructured text editors.

Each node within a document is addressable in a variety of ways. A node can be addressed by a unique identifier, by a label indicating the node's hierarchical position in the document, by an author defined name, or by physically pointing to it in the Augment user interface display. Node addressing is the basis for targeting Augment commands to specific nodes as well as for embedding links to

nodes within the text of a node. For example, a reference link to a node containing a bibliographic citation can be embedded in the referencing text by specifying the address of the target node. Augment provides the capability to traverse such links so that readers can easily view the text of the target of an embedded reference.

Other Augment features include text, content, and inter-file addressing capabilities and commands that manipulate the display of a document based on structural considerations. With text and content addressing, a link can be made to a specific word, phrase, or letter within the text of a node. With inter-file addressing, a link can be prefaced by a document name so that nodes in different documents can be linked together. The commands that manipulate document views based on structural considerations allow Augment users to create non-linear filtered displays of a document. For example, all of the document nodes that lie above or below a specified hierarchical position can be clipped from the display of the document. User content filters can be used to further manipulate the display by filtering out nodes with content that does not satisfy the filter specification.

Augment is important because of its historical significance - it is the first viable structured text editor - and because of the comprehensive set of tools that it provides. In the twenty years since the development of Augment was first begun, the system has evolved into a powerful document preparation environment. Augment's fundamental editing facilities have grown to include an electronic journal in which documents are electronically published and available for referencing and perusal. In addition, shared-screen conferencing support provides a medium through which authors can collaborate. Electronic mail, with the capability to "record dialogue", provides a device with which communication can be documented. There is little doubt that Augment has influenced virtually all of its successors and still sets a standard for the integration of complimentary

tools.

1.5.3 Textnet

Textnet is the foundation for a *hyper-text* system[1] in which the structuring of documents is used to provide novel ways of presenting, reading, critiquing, and cross-referencing documents. Textnet documents are composed of three basic kinds of objects - *chunks*, *tocs*, and *links*. A chunk is the encapsulation of a primitive piece of text. A toc is similar to an entry in a table of contents and provides an overview of underlying content, but unlike a chunk, is not explicitly associated with a piece of text. A link is used to join chunks and tocs with other chunks and tocs. Using tocs and links, an author gives structure to a document, while chunks are used to provide content.

The structure of a Textnet document is not constrained to be strictly hierarchical, but can evolve into a directed acyclic graph. In the graph, tocs and chunks are nodes, and links are edges. The document structures that can be represented can be tree-like hierarchies, or more complicated, but flexible, organizations that allow multiple links to the same node. The document structures that can be represented in Textnet are the basis for many of the system's features, including critiquing and the linking of other complete or partial documents at any point in a particular document. Textnet attempts to offer more than the ability to create and manipulate an individual document by

---

[1] A *hyper-text* system is an environment in which substantial bodies of literature are archived "on line", readily available for electronic perusal and referencing. In academic and scientific communities it is rare to find a document that does not contain references, and it is almost as unusual for a document not to be referenced [Price 67]. The dependence of a document on other documents implies that literature within and across disciplines form enormous bodies of logically related documents. To organize these documents in a fashion that puts them "on line" for easy access by the communities served, hyper-text systems have been proposed. The scope of hyper-text systems is the national, even world-wide, electronic archiving and linking of documents into a readily accessible, easily navigated network of literature.

providing an environment in which networks of documents can be constructed and subsequently perused in a manner that compliments the interests of the reader.

Textnet explores the use of some extremely powerful ideas that put it at the vanguard of existing structured document/hyper-text systems. One such idea is the typing of link objects. The type of a link conveys semantic information by establishing a relationship between the nodes joined by the link. For example, the link type "support" implies that the source node contains an argument that supports the argument put forth in the target node. Another idea is the use of *paths* to capture a walk along a document graph. Paths can be tuned to a particular author or reader audience as a way to navigate through and across documents in a manner that focuses on the portions of interest to the audience.

1.5.4 JEDI

JEDI is a real-time joint document editing system. The system provides an environment in which authors can either independently edit a document or synchronously collaborate on editing a document. Concurrent, but independent, edits are coordinated by one of two schemes. *Validation* allows an author to save changes to a document as a new document version providing that the version saved was created from the last saved version of the document. *Reservations* allow an author to obtain exclusive access to a document for several minutes. At the end of the reservation interval, the changes made by the author are automatically saved as a new document version. Synchronous collaboration on a document is coordinated by a set of protocols that allow an author to obtain exclusive access to a document that is simultaneously viewed by a group of authors.

JEDI's support for collaborative editing provides authors with *shared* and *private*

workspaces. Access to the shared workspace is coordinated, but an author can work independently and exclusively in a private workspace. The shared workspace is implemented as a set of windows, each displaying a view of a document. The contents of these windows is the same for each author's display. The private workspace is also implemented as a set of windows. The contents of these windows is created by copying text displayed in the shared windows.

During a joint editing session, authors must obtain a reservation on a shared window before it can be accessed. A window-reservation can be freely obtained providing that it is not already held. Built-in monitors ensure that a window-reservation is held only while an author is actively using the window - the reservation expires otherwise. The monitors also ensure that a reservation that is held is eventually canceled if another author requests a reservation for the window. The monitors implement JEDI's philosophy of encouraging sharing. Once a reservation on a window is obtained, *region-reservations* are used to select a portion of the document's text for editing. Region-reservations prevent authors who have reservations on different windows from interfering with each other because through the windows, each author has access to the document.

JEDI provides a variety of other features, including *super-reservations* for exclusive access to the entire shared workspace, and *negotiation* facilities for reaching agreement on issues of common interest. The super-reservation allows all of the authors to participate in an editing session but only one author at a time is allowed to edit the document. Negotiation allows authors to reach an agreement about such issues as the resizing of the display space occupied by the shared windows. The granting of a super-reservation is mediated by the editing session chairperson - the author who initiated the session. The negotiation process is mediated by the author who placed the issue to be negotiated on the forum. Both of these activities are intended to be augmented by real-time verbal communication devices such as a telephone.

1.5.5 Grapevine

Grapevine is a distributed application designed to serve an internet - a collection of local area networks. The system offers electronic mail and a variety of facilities that can be used for locating people, machines, and other services on the internet. In addition, Grapevine is a highly reliable system. This is achieved by replicating data and services on different computers throughout a Grapevine network. Only one of the computers that performs the desired service or has a copy of the desired data needs to be functional in order for a user to have access to the service or data.

Grapevine's functionality is partitioned into a *delivery service* and a *registration service*. The delivery service is built upon a set of message servers, each responsible for a set of message *inboxes*. An inbox is a message buffer for a user or computer resource (e.g. printer, file server). Inboxes are replicated on a subset of the Grapevine message servers. The registry service is built upon a set of registration servers, each responsible for a set of user *registries*. A registry maps a user's name (either human or computer) to an authentication password, the internet address of the user's host computer, and the list of message servers at which the user has an inbox. Like the inboxes, the registries are replicated on a subset of Grapevine registration servers.

Grapevine provides clients with a set of primitives for accessing registries and inboxes. The primitives provide an abstraction of Grapevine's functionality, and, in particular, the replication of data and services. For example, the sender of a message need only specify the recipient's name; Grapevine determines where the sender's inboxes are located and then delivers the message to one of these inboxes. Similarly, Grapevine reads messages by examining the contents of all of a user's inboxes. For most uses, Grapevine effectively masks its distributed nature yet still provides clients with the benefits of enhanced system reliability and access to physically remote resources.

At times, Grapevine will allow its distributed nature to be revealed. This occurs during the updating of copies of replicated data. Rather than using an atomic update procedure in which most of the copies of the data must be accessible in order to perform an update, Grapevine employs a convergence algorithm. This algorithm, which requires that only one copy of the data be accessible, assures that eventually all copies will be updated. However, in the convergence interval it is possible for users to see inconsistencies. By opting to let Grapevine users deal with these transient inconsistencies, the Grapevine designers have made an interesting decision to enhance performance by weakening the Grapevine abstraction.

1.6 Thesis Organization

This chapter introduced some of the basic issues that confront the CES design. In Chapter 2, the CES design is detailed. This includes a discussion of the system architecture as well as the functionality provided, with an emphasis on CES support for collaboration. In Chapter 3, the programming language Argus is introduced. This chapter provides an overview of the language and aspects that are particularly relevant to the CES implementation are explored in detail. In Chapter 4, the CES implementation in Argus is described. This chapter focuses on portions of the implementation that provide basic system functionality. The chapter concludes with a critique of Argus with respect to the role that the language played in the CES implementation. Chapter 5 concludes the thesis with a review of how well the goals itemized in this chapter were met, and presents ideas for further research.

Chapter Two

Collaborative Editing System Design

CES is a document editor that supports the collaboration by a group of authors on a shared document. The system provides the following features:

- Full-screen interactive structured document editor.
- Document structures can be specified to reflect standard document organizations (i.e. chapters, pages, sections, etc.) or custom organizations.
- Different authors can work independently on separate portions of a document.
- Individual authors editing a portion of a document have exclusive access to the portion.
- All authors can view any portion of a document even if it is being edited by another author.
- Authors can work at personal workstations or multi-user computers connected by a network.
- Documents are physically distributed so that the contributions made by an author reside on the author's computer, yet a document can still be accessed as a single, logical entity.
- Authors can comment or critique a document. The comments and critiques can be subsequently filtered from the document view.
- Documents can be made available for public viewing; they can also be made available for public editing.

CES is best suited for large jointly-authored documents for which the co-authors

agree that individual authors are responsible for individual portions of the document. This allows authors to obtain exclusive access to a portion of a document based on the prior scoping of the document by its authors into non-overlapping sub-portions. In addition, it provides a basis for identifying individual contributions and enforcing the privilege of the "owner" of the contribution to ultimately control whether or not the contribution is shared. In order to implement the capability to partition a document as described, CES employs a structured representation of documents. A structured document's structure, or organization, is explicit and separate from its content. The document's content is developed from individual textual components which are linked together as dictated by a structural specification to provide a readable image of the document.

This representation of a document is in contrast to that of an unstructured document, typically represented as one or several large blocks of text. In an unstructured document, the author must distinguish between chapters, pages, paragraphs, and other structural entities. Although these elements can be delimited by the author for formatting purposes, the resulting structure is implicit in the text and is not part of the document's representation. Therefore, the document's structure can neither be viewed nor manipulated independent of textual content.

The structured approach is used in CES because it provides a framework for coordinating authors' activities. The document components can also be physically distributed, yet logically linked based upon the document's structural specification. CES authors use the explicit document structure in order to design documents and plan contributions before actually beginning to write, and to make structural modifications during writing. CES uses the fact that an author's writing activities are targeted at the level of the document's textual components to provide concurrent, but coordinated, access to a shared, distributed document.

2.1 Structuring of Documents

A CES structured document is composed of two types of objects. For each document, a single *structure* object is a schematic representation of the document's structure, and each *node* object in a set of node objects encapsulates a portion of the document's content. The structure object is essentially a set of pointers to node objects, so that a document is completely described by its set of nodes and the ordering of its nodes as described by the structure object. Node objects capture content; structure objects capture context. The structure and content are stored in different objects to allow changes to a document's structure or content to be made separately.

Structure objects describe a strictly hierarchical tree-like document organization that can be used to represent the the organization of most documents. A simple document such as a business letter might consist of a small number of items (e.g. heading, body, closing), each at the same hierarchical level. On the other extreme, the organization of a report might contain a large number of nested items (e.g. chapters containing sections, sections containing subsections, etc.).

Structure objects are comprised of an linked list of items that represent the nodes of the document's tree structure. When presented to the user by CES, the list of items is displayed in indented outline fashion. Thus, the order and nesting of items represent the edges of the document's tree structure. Each item uniquely corresponds to a node object. This association is specified by the item's *node unique identifier* which is a pointer to a node object. In addition, an item contains a brief *description* or *name* for its associated node, and a *label* indicating the node's hierarchical position in the document's organization. The label is specified as a sequence of integers. The number of integers in the sequence indicates the breadth-wise position of an item and the magnitude of the integers indicates the depth-wise position of an item. For example, a node with

the label "1.22.3." is the third child of the node with the label "1.22" but parent to the node with the label "1.22.3.1.".

A node object corresponds to one item in the document's outline representation and is identified by a CES generated unique identifier. Node objects encapsulate an arbitrary amount of text. Therefore, node objects can describe a single idea or several related ideas. For example, the text of a node object can be a single bibliographic reference, or it can be several paragraphs in a chapter of a document. Node objects can be thought of as a set of variable length text strings. In this regard, a node object is similar to a standard text file.

CES provides a readable image of a document. This is done by linking together text contained in a document's node objects as specified by the document's structure object to produce a *text image* of the document. The "pasting" together of the contents of node objects in this fashion creates a single, linear presentation of a document as it would appear on paper. The text image can be viewed as a single block of text, with the option to display demarcations between the text of individual node objects.

The interface between a user and a CES document is a real-time screen editor. Text editing is also the basic mechanism for modifying a document's structure - authors edit the text of the outline representation of a document's structure. Using editor commands, outline items can be relocated, new items can be added, and existing items can be deleted. The subsequent parsing of the outline text by CES reorganizes the document's structure to match the outline changes. The text image is similarly reorganized.

2.2 User Interface

2.2.1 Display

The CES user interface displays one document at a time for reading and editing. The display allows a user to simultaneously view a document's outline, text image, and the text of an isolated node in three windows. The windows are horizontally stacked and do not overlap. The window widths are the same size; the heights are different, though fixed relative to one another. The top two windows each occupy approximately one-quarter of the total display, while the bottom window occupies the remaining half. Each window is labeled with its name. Above the top window are two display lines. The top line is used by CES to display errors and messages; the bottom line is used to display status information such as the name of the document being viewed. The layout of the user interface display is shown in Figure 2-1.

The displays in each window are CES interpretations of the document being viewed. The top, or *outline*, window displays the outline representation of the document's structure. The middle, *document node*, window displays the text of one of the document's node objects. The bottom, *text image*, window displays the document's text image. Typically, the text for an outline, node, or text image cannot be displayed in its entirety within the confines of a window. For the case in which there are more lines of text than lines in a window, the windows provide a set of operations that allow the user to page or scroll through the text. For the cases in which the width of a line of text exceeds the width of a window, an exclamation mark "!" in the right-most window column indicates the overflow, and a window primitive operation can be invoked that will display the "missing" text on the interface message line.

The interface display also supports a variant of windows called *overlays*. An

overlay is a window superimposed on part or all of a window's display. The size of an overlay cannot exceed the size of the window being overlayed. An overlay is a way to temporarily create additional windows, though only one overlay per window is allowed. There are two kinds of overlays - *question* overlays, and *display* overlays. A question overlay occupies the two bottom lines of the window being overlayed. The display overlay can occupy a variable number of overlayed window lines. The question overlay is intended to be a mini-window for use by CES with prompts that require a user response. The display overlay is intended for use by CES so that the results of a user's query can be presented.

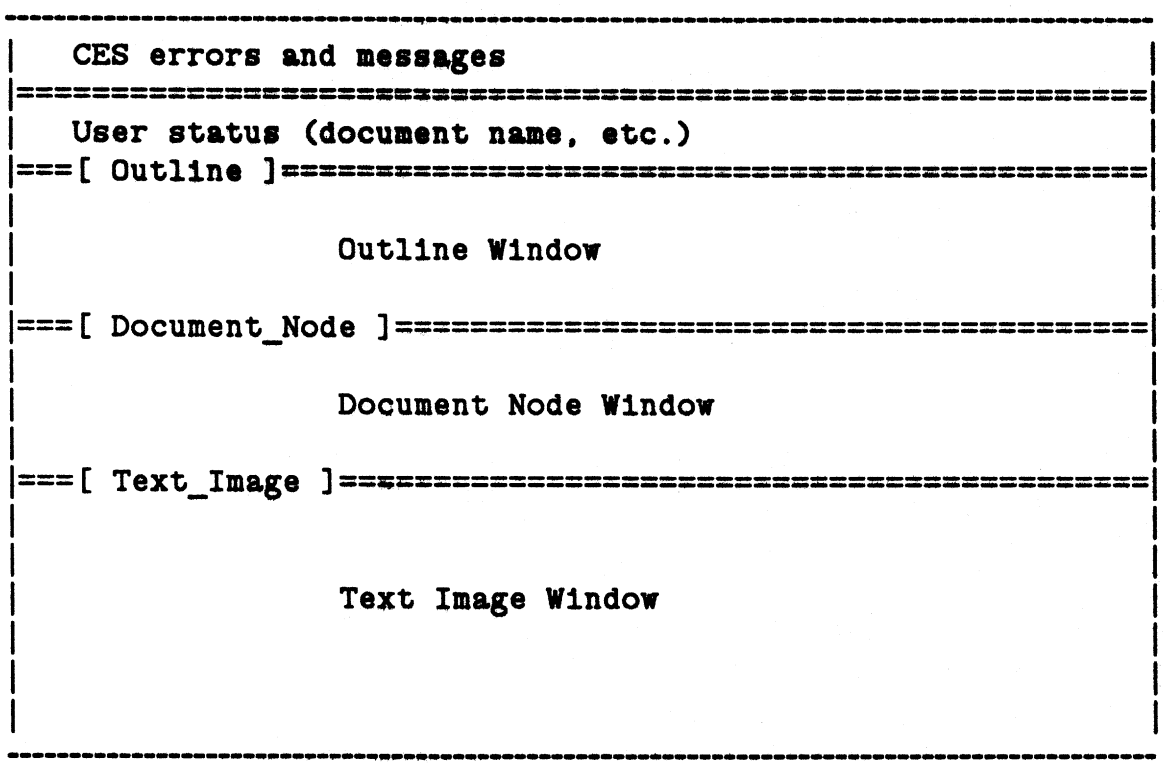

Figure 2-1: Layout of CES User Interface Display

provides a view can be tagged as read-only. An attempt by the user to enter text in a read-only portion of a window results in the display of an error message and the read-only portion of the window is not affected. The top line of a question overlay is implicitly read-only; a display overlay is implicitly completely read-only.

In windows configured to accept user text and in the bottom line of a question overlay, alphanumeric text is inserted at the keyboard cursor position, and an interactive screen editor provides a full range of text manipulation functions. This editing functionality is similar to that offered by the popular EMACS editor [Stallman 81], though most features have been derived from the editor TED [Atkinson 84]. Editing features include a variety of commands to insert, delete, find, and replace characters or entire strings.

2.3 Fundamentals of Editing a Document

CES explicitly separates the process of modifying a document's structure from the process of modifying its content. To achieve this, the three windows introduced in Section 2.2 serve distinct roles. The outline window is used for the display and editing of a document's structure. The document node window is used for the isolated editing of document nodes. The text image window is used to provide a supporting view of the document that provides a context for node editing. Both the outline and document node windows freely accept user commands and textual input (though at times this might be restricted), but the text image window is for viewing purposes only and accepts only non-editing related commands. The following sections provide additional details about these windows.

2.3.1 Creating a Document

In order to create a new document, the author begins by writing an outline in the outline window. This initial outline is specified with items composed only of node descriptions or names, ordered and indented to convey the desired document structure. A simple business letter might contain a handful of outline items, each with the same degree of indentation. Similarly, the basic framework for a report might contain several items, with items for chapters indented the least, sections next, then paragraphs, and so on.

When the initial outline has been specified, the command to parse the outline can be executed. This results in the initialization of the new document's structure object. The initial structure of the document thereby reflects the structure described by the parsed outline. For each item in the outline, a new node object is created. Also, an empty text image with node demarcations is displayed in the text image window.

2.3.2 Viewing a Document

The window primitives mentioned in Section 2.2 for scrolling or paging through text provide coarse ways to view windowed text. Additional commands are provided by CES that allow the manipulation of windowed views based on the structure of the document being viewed. By using the keyboard or mouse cursor to point to items in an outline and issuing the appropriate commands, the alignment of the text image in the text image window can be altered. This technique acts as a node text "go to" and will align the text image so that the text for the node associated with the outline item pointed to will start at the top of the text image window. Similar node text "go to" commands can be issued from any of the windows and accept either a node identifier, an item label, or a

signed count of nodes[2] as arguments. For example, when issued from the outline window, the commands "go to node 2@oa", "go to 1.1.", and "scroll +1" will all align the text of the second node displayed in the text image shown in Figure 2-2 with the top of the text image window.

Using the node text "go to" commands is one way to use the text image in order to locate the boundaries for a particular node's text. An alternative is to turn on the display of the text image node *demarcations*. The demarcations are inserted into the text image display at the start of each node's text. The demarcations consist of a horizontal separator preceded by the text for the outline item associated with the node. Therefore, the demarcations not only separate node text, but identify the nodes as well. The display of node demarcations facilitates a variety of editing operations by allowing viewers to readily isolate several nodes at once for viewing. Turning off the display of node demarcations restores an uninterrupted display of the text image.

A single demarcation is also used to identify the node displayed in the document node window. Like the text image node demarcations, the display of the single document node demarcation can be turned on and off. When displayed, the text for the demarcation is configured as a read-only portion of the document node window. An example of a document display with text image node demarcations is shown in Figure 2-2; an example of a document display without text image node demarcations is shown in Figure 2-3.

[2]When issued from the outline or document node windows, the signed count is relative to the node at the top of the text image display in the text image window. When issued from the text image window, depending upon the command, the signed count is relative to the node pointed to by the keyboard cursor or mouse cursor.

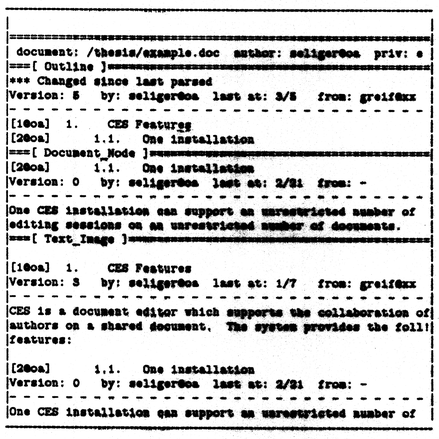

Figure 2-2: Viewing a Document with Outline Status and Node Demarcations

```
|                                                                |
|================================================================|
| document: /thesis/example.doc  author: seliger@oa  priv: e     |
|===[ Outline ]==================================================|
|[10oa]  1.    CES Features                                      |
|[20oa]       1.1.   One installation                           |
|[30oa]       1.2.   Simultaneous views                         |
|[40oa]       1.3.   Real-time screen editor                    |
|[50oa]       1.4.   Locality of ownership                      |
|===[ Document_Node ]============================================|
|[20oa]       1.1.   One installation                           |
|Version: 0   by: seliger@oa  last at: 2/21  from: -            |
|- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - |
|One CES installation can support an unrestricted number of      |
|editing sessions on an unrestricted number of documents.        |
|===[ Text_Image ]===============================================|
|CES is a document editor which supports the collaboration of    |
|authors on a shared document.  The system provides the foll!    |
|features:                                                       |
|                                                                |
|One CES installation can support an unrestricted number of      |
|editing sessions on an unrestricted number of documents.        |
|                                                                |
|Each author participating in a CES session can simultaneous!    |
|view a shared document while editing a portion of the docum!    |
|                                                                |
|Actual editing chores are performed using a real-time screen    |
|editor.                                                         |
```

Figure 2-3:
Viewing a Document without Outline Status or Text Image Node Demarcations

2.3.3 Modifying a Document's Structure

Modifying a document's structure involves editing the document's outline representation in the outline window and parsing the outline. After parsing, the changes made to the outline are incorporated into the document's structure object and are reflected in the text image.

2.3.3.1 Outline Parser

The outline parser scans an outline line-by-line, checks the syntax of each line, and if no errors are found, alters the document's structure object so that it corresponds to the outline specification. When an outline is parsed, blank lines and comments (i.e. statements following the "%" character) are ignored. All other lines are treated as outline items. Each item contains information about a different document node, including the unique identifier for the node (delimited by a pair of square brackets), a label conveying the node's position in the document's structure, and a description or name for the node. For specification purposes, only the node identifier and description are required, as the parser computes a label after it determines the node's position. An example of an outline item is shown in Figure 2-4.

The depth-wise position of nodes in a document's logical structure is specified by the degree of indentation in the outline of the item associated with a node. This indentation is determined by the column in which the node description or name starts. The greater the indentation, the lower the depth of the node in the document hierarchy. The breadth-wise position of nodes is specified by the ordering of outline items.

In order to simplify the specification of document structures, indentation is relative. The depth of the node is conveyed by whether or not its associated outline item is indented with respect to the item's predecessors and successors,

[23@oa] 4.5.17. How to get help

Figure 2-4: Example of an Outline Item -
Each outline item contains three fields. The first field is the unique identifier of the node associated with the item. This field is delimited by a pair of square brackets. The second field is the "infinite" precision label which conveys the associated node's position in the document's structure. The third field is the description or name for the associated node.

and not by how much the item is indented. For example, the outlines shown in Figure 2-5 will result in the same document structures when parsed, even though the items in the second outline which are at the same hierarchical depth are not uniformly indented.

When an outline item is parsed, the parser attempts to extract the node identifier, label, and node description. If a node identifier is found, it is used to associate the outline item with an existing node. If a node identifier is not found, then a new node is created and its identifier is inserted into the outline. If a node label is found, it is ignored, as labels are recomputed during parsing. The remaining text on the line is treated as the node description. As previously described, it is the indentation of this text that is used to compute the depth of the node in the document hierarchy. Exceptions to these rules, such as node identifiers that refer to non-existent nodes, improperly formatted node identifiers or labels, or identifiers or labels that are not followed by a node description or name, are flagged as errors by the parser and reported to the user, and no further parsing action is taken.

```
Abstract                    Abstract
Introduction                Introduction
  Motivation                      Motivation
  Overview                        Overview
Conclusion                  Conclusion
  Improvements                Improvements
  Future Work                 Future Work
```

Figure 2-5: Examples of Unparsed Outline Indentation - The items in the outline on the left are uniformly indented; the items in the outline on the right are not. When parsed, though, both outlines will result in the same structures because the outline parser is sensitive to relative, and not absolute, indentation.

When the parser has parsed an outline and there are no errors, it updates the structure object and redisplays the outline. The redisplayed outline is "cleaned-up" so that all of the node identifier fields start in the same column, and the indentation for the labels and descriptions of nodes at the same hierarchical depth start in the same display columns. Figure 2-6 shows the outlines in Figure 2-5 after parsing.

The parsing rules allow a good deal of flexibility. First, the author need not worry about the values of node labels, as they are recomputed by the parser. Second, except for the cases when a node is to be copied or moved (See Section 2.3.3.2), the author need not be concerned with node identifiers, as they are assigned by the parser. Third, except for the fact that node descriptions or names should not start with text that could be confused as a node identifier (e.g. 3@oa) or label (e.g. 2.1.), node names are unconstrained and can have meaningful length and content. The result is that the need for special commands to create

```
[1@oa]  1.  Abstract
[2@oa]  2.  Introduction
[3@oa]    2.1.  Motivation
[4@oa]    2.2.  Overview
[5@oa]  3.  Conclusion
[6@oa]    3.1.  Improvements
[7@oa]    3.2.  Future Work
```

Figure 2-6: Outlines from Figure 2-5 After Parsing

or alter an outline, other than those used for basic text editing and invoking the parser, is obviated.

2.3.3.2 Editing an Outline

In order to add a node to the document's structure, just the description or name for the new node is added to the outline, and then the outline is reparsed. The outline semantics described in Section 2.3.3.1 govern the resulting hierarchical location of the node in the document's structure. When the outline is reparsed, the new node is assigned a unique identifier and a label indicating its hierarchical position is computed.

In order to delete a node from the document's structure, the outline item associated with the node is simply removed from the outline, and then the outline is reparsed. Reparsing the outline will alter the document's structure object and text image so that the deleted node is no longer included. As will be discussed in more detail in Section 2.3.7, the deleted node is not destroyed. At this point it has merely been removed from the document's structure, but is completely recoverable.

Nodes can also be moved or copied. In order to move a node, the outline item associated with the node is moved to another position in the outline, and then the outline is reparsed. The node unique identifier is left unaltered. The node description or name can be changed if desired. In addition, the depth of the node in the document hierarchy can also be changed by altering the indentation of the node description or name. When the outline is reparsed, a label indicating the new hierarchical position of the node is computed

The process for copying a node is similar to that used for moving a node - the outline is modified and then reparsed. Node copying creates a new node that is initialized with a copy of the text of the node being copied. The text copied is not shared between the nodes, and subsequent alterations to the text of one node will not affect the text of the other node. In order to make a copy of a node, a copy of the outline item associated with the node to be copied is entered in the outline. In fact, all that really needs to be copied is the unique identifier of the node to be copied; the description or name for the new node can be different and its label will be computed by the outline parser. The copy of the node can be positioned anywhere in the document structure by appropriately positioning the copied outline item. Figure 2-7 illustrates the copying of a node.

2.3.4 Modifying a Document's Contents

The modification of a document's contents is performed on a node-by-node basis. Using commands similar to the node text "go to" commands described in Section 2.3.2, a node can be specified and isolated for editing in the document node window. If the node being edited also happens to appear in the text image window, the text image is updated as the changes are made to the node in the document node window.

For example, by pointing to an outline item with the keyboard or mouse cursor

```
[1@oa]  1.  Abstract
[2@oa]  2.  Introduction
[3@oa]    2.1.  Motivation
[4@oa]    2.2.  Overview
[7@oa]          Copy of Node 7@oa
[5@oa]  3.  Conclusion
[6@oa]    3.1.  Improvements
[7@oa]    3.2.  Future Work
```

Figure 2-7: Example of Node Copying -
Assuming that the outline has been previously parsed, when it is reparsed the node named "Copy of Node 7@oa" will have its identifier changed from 7@oa to a unique identifier, it will have the label "2.3." computed for it, and its text will be a copy of the text for node 7@oa ("Future Work").

and issuing the appropriate command, the text of the node associated with the outline item can be displayed and edited in the document node window. The full compliment of text editing commands can be used in this capacity to edit node text.

Although most text editing operations are targeted at individual nodes, there are times when it is desirable to perform document-wide edits. Document-wide editing applies a single editing operation to the text of several, or all, of the nodes in a document. Therefore, document-wide editing commands accept arguments that specify a range of nodes to which the command should be targeted, so that the scope of the resulting operation can be flexibly restricted. For example, it is not unusual to use the string find-and-replace command in order to change a commonly used string throughout a document. This command scans through the document nodes locating each occurrence of the target string

and prompts the author to confirm the replacement of the string.

In order to preserve the "for-viewing-only" role of the text image window, document-wide commands operate node-by-node like other editing commands, except that they automatically sequence nodes through the document node window rather than requiring the manual selection of each targeted node. In addition, as nodes are sequenced through, the windowed view of the text image is realigned to include each node being sequenced through as the node appears in the document node window. For commands that require the author to decide on whether or not to actually perform the command, the realignment of the text image provides a context for the author's decision.

2.3.5 Maintaining Consistency Among Views

One potential problem with presenting the document's structure as a component separate from the document's contents is maintaining consistency between the outline and the text image. The text image presents the contents of the document nodes in an order that reflects the structure of the document, so that inconsistencies can potentially arise if the outline is changed, but the text image is not. This problem is resolved by the outline parser which updates the text image by appropriately reordering the blocks of text for the document nodes whenever the outline is parsed without errors. After an outline is parsed and the text image is reconstructed, both the outline window and text image window displays are automatically updated.

2.3.6 Logging and Saving Changes

Changes to a document's structure and content can be saved on *version stacks*. For each document, there is one version stack for the document's structure object and one version stack for each of the document's node objects. Version stacks provide a device for archiving changes made to a document, though, if necessary,

these changes can be undone.

The version stacks allow changes made to the *current* version of a document's structure, outline representation, or any of its nodes, to be separately saved. The current version of a structure object is reflected by the last parsed outline *and* any changes that have been made to the outline since it was parsed. The current version of a node object is indicated by the last change made to its text *and* the node label and description or name that appeared in the outline item associated with the node when the outline was last parsed.

A copy of the current version of a document's structure object, or a copy of the current version of any of a document's node objects, can be *pushed* onto the object's version stack to create a new version. Alternatively, previously pushed versions can be *popped* off (in last-pushed-first-popped order). In addition, a version stack can be reset with the version at the top of the stack installed as the base version of the structure or node object. When a version stack is popped or reset, the version of the structure or node object that is on top of the affected version stack is installed as the current version of the object.

Associated with each version of an object on a version stack is a sequentially ordered version number (starting with zero), the name of the author who pushed the version onto the stack, and the row and column of the author's keyboard cursor when the version was pushed on the version stack. For a document structure object, the cursor row and column are relative to the text of the outline representation of the structure; for a document node, the cursor row and column are relative to the text encapsulated by the node. This information allows authors to locate where changes were made to a document structure or node and is included as part of the node demarcation in the *version*, *by*, and *last at* fields of the demarcation display, as shown in Figure 2-2.

Version stacks are intended to provide authors with the capability to log changes made to a document during the course of an editing session. An author can push a copy of the current version of a structure or node object onto the object's version stack, or examine or remove existing versions. For example, any version of any object can be viewed in a display overlay. In addition, CES automatically logs versions of a structure object by pushing a copy of the structure object on its version stack after the outline representation of the object is parsed. Similarly, CES automatically logs versions of nodes by pushing a copy of a node object on its stack when the first change is made to the node's text after it has been selected for editing. At the conclusion of an editing session, the version stacks for all of a document's component objects can either be reset, thereby saving a new version of the document, or cleared by purging all but a version stack's base version, in which case the changes made to the document are undone.

2.3.7 Orphan Nodes

The deletion of a node from a document's structure does not destroy the node. Instead, the node becomes an *orphan* - the node is no longer part of the document's structure. It is desirable to keep deleted nodes intact, as future changes to the document might require that such a node be reincorporated. The list of orphan nodes for a document and the text of an individual orphan node can be viewed at any time in a display overlay. This list includes the node's unique identifier and description or name. Conversely, the author can choose to destroy one, several, or all of a document's orphan nodes. The destruction of an orphan node is a non-recoverable action.

2.3.8 Altering Document View Perspectives

A benefit of being able to deal explicitly with a document's structure as well as its content is the ability to manipulate perspectives of the document view. CES provides a set of commands which iteratively act upon the current view of the document to produce a new perspective of the view. These commands do not change the document's structure, but do change the display of the outline and text image, thereby allowing users to quickly and easily focus in on parts or subsections of the document's structure. Half of the perspective-altering commands act by removing, or clipping, nodes at a specified hierarchical level of the document's logical structure. The clipped perspective is reflected in the outline and the text image through the removal of outline items and the text of associated node. The other half of the perspective-altering commands act to restore, or unclip, nodes which have been clipped.

For example, all of the nodes that are at the highest hierarchical level in the current view perspective can be removed from the view perspective. A similar command will remove all of the nodes that are at the lowest hierarchical level in the current view perspective from the view perspective. Complimentary commands will restore these nodes to the view perspective. Other commands take into account the hierarchical position of the node associated with the outline item pointed to with the keyboard or mouse cursor, and clip all nodes from the view perspective that have a higher or lower hierarchical position. Finally, there is a command which restores the view of the entire document. An example of the results of a command which clips the nodes at the lowest hierarchical level in a view perspective is shown in Figure 2-8.

```
[1@oa]  1.  Abstract
[2@oa]  2.  Introduction
[3@oa]    2.1.  Motivation
[4@oa]    2.2.  Overview
[5@oa]  3.  Conclusion
[6@oa]    3.1.  Improvements
[7@oa]    3.2.  Future Work
```

After clipping nodes at the lowest hierarchical position ...

```
[1@oa]  1.  Abstract
[2@oa]  2.  Introduction
[5@oa]  3.  Conclusion
```

Figure 2-8: Example of Clipping a Document View Perspective

2.3.9 Document Critiquing

Another benefit of being able to deal explicitly with a document's structure is the ability to critique a document without violating the integrity of the body of the document. CES allows authors to include *critique* nodes in the document. A critique node is one that is not part of the proper document structure, but is included at the desired points to serve as a commentary or critique. Critique nodes can be added like regular nodes, wherever they are required. Both the document's structure and text image will be updated to include the critique nodes, but when the author wants to view the document without the critiques, they can be readily filtered from both the outline and text image. These nodes serve a role similar to the Scotch™ Post-it™ note pads. The adhesive-backed pages from these pads can be commented and affixed to paper documents

wherever desired, yet can be easily removed to reveal the original document. Critique nodes can similarly be "affixed", "commented", and "removed".

```
[1@oa]  1.  Abstract
[2@oa]  2.  Introduction
[3@oa]    2.1.  Motivation
[4@oa]    2.2.  Overview
[5@oa]  3.  Conclusion
[8@oa]  3.C Ideas for clarifying test results
[9@oa]  3.C Suggestions from research advisor
[6@oa]    3.1.  Improvements
[7@oa]    3.2.  Future Work
```

Figure 2-9: Example of Critique Nodes

The process for adding a critique node is similar to the process for adding a document node in general. One difference is that when the outline item for the critique node is added to the outline, the node description or name is preceded by an upper or lower case "C", for "critique". If a label is present, then the "C" should immediately follow the label (as in "1.2.C"). Another difference is that the indentation of the name for the critique node is ignored, so that the critique node is placed in the document's structure at the same hierarchical level as the node whose item immediately precedes the critique node's item in the outline. In addition, after the outline is parsed, the critique node is assigned the same label as the preceding node, except for the appended "C". In the example of a parsed outline shown in Figure 2-9, the nodes with identifiers 8@oa and 9@oa are both critique nodes for the node with identifier 5@oa.

2.4 Support for Collaboration

CES support for collaboration provides solutions for the following problems:

- Coordination of the activities of a document's authors.
- Sharing of document changes with all of a document's authors.
- Support for *public* access to a document, where for a particular document, the public is any CES user who is not one of the document's authors.

CES is a distributed application. It is installed on a network of computers. For such networks, CES support for collaboration assumes that the network has the following capabilities:

- The ability to uniquely name each author in the network.
- The ability to uniquely name each computer in the network.
- The ability for all of the computers in the network to communicate and share data with each other.

Given that these capabilities exist, CES provides the following resources and additional capabilities:

- A CES installation-wide *document library* cataloging all of the document structure objects and node objects for every document supported by the installation.
- A CES installation-wide *document registry* listing all of the CES documents supported by the installation.
- For each document in the registry, the ability to dynamically maintain a list of its authors.
- For each author in each document's author list, the ability to specify an *access privilege* with respect to the document. The access privileges are:
    - No-access Author is not allowed to view or edit the

document.

- View-only — Author is allowed to view, but not edit, the document.
- Edit — Author is allowed to edit the document.

- For each document, the ability to specify a public access privilege.

In the remainder of this section, the details of collaborative support are discussed. The discussion focuses on how authors' activities are coordinated, and how contributions are shared. As will be described, once the authors' activities have been coordinated at a coarse level, the editing tools and techniques described in the previous section are all directly applicable.

2.4.1 Creating a Document

Document creation is a two-phase process. In the first phase, an author adds the name of the new document to the document registry. The name of this author is automatically included with the document registry entry for the new document, and the author is permanently distinguished as the document's *creator*. The creator of a document has permanent edit access. The author is then asked to provide the list of authors who will initially collaborate on the document. This list can be left empty, or it can contain an unconstrained number of authors. Each entry on the list must include the collaborating author's name and access privilege. The author must then specify the public access privilege. At the completion of this phase, the entry for the document in the document registry is permanent and available for examination by other authors or the public.

During the second phase of document creation, the author specifies the structure of the document in outline form in the outline window, and the outline is then parsed. This is as was described in Section 2.3.3.1. If there are no errors, the

resulting document structure is cataloged in the document library. In addition, a document node is created for each entry in the outline, and each node is cataloged in the document library. A text image of the document, with demarcations turned on, is then displayed. At the completion of this phase, the initial structure of the document and its initial compliment of document nodes are available for use by other authors or the public.

2.4.2 Accessing an Existing Document

When a user requests that a document be displayed, CES compares the user's name against the list of names for the document's authors. If the user is one of the document's authors, then the user is assigned the access privilege specified in the list. If the user is not one of the document's authors, the user is assigned the public access privilege. The access privilege for the user with respect to the document is displayed in the *priv* field of the user's interface display status line. The privileges are abbreviated as:

- e = Edit
- v = View-only
- n = No-access

An example of a user with edit access is shown in Figure 2-2.

A user with no-access can neither view or modify the document. A user with view-only access can view a document's outline representation and text image, and can change view perspectives, but cannot modify or critique the document. A user with edit access can view and modify the document.

A document's public access privilege or the access privileges of the document's authors can be changed at any time, though only a user with edit access to the document can make such changes. A user's access privilege can be changed while

the user is accessing a document, though it can take up to twenty seconds before the change is effected. Another way to change a user's access privilege is to add the user to, or remove the user from, the document's list of authors. This can be done at any time, though only by a user with edit access to the document, and will take up to twenty seconds before the change is effected.

Access privileges serve three basic purposes. First, they can enforce a distinction between a document's authors and the public. Second, they can make documents available to the public for reference or for contributions. For example, public comments on a document can be obtained without altering the document by requesting critiques[3]. Third, they can be used to restrict access to a document within the document's community of authors. This can be particularly useful during the early development of a document when it might be desirable to allow only a few of the document's authors to modify the document while allowing the other authors to monitor the development.

2.4.3 Modifying a Document's Contents

2.4.3.1 Obtaining Exclusive Access

In order to edit a node, a user (referred to in this discussion as the "querying author") must first obtain *exclusive edit access* to the desired node by selecting it for editing, as described in Section 2.3.4. When the desired node has been specified, CES attempts to obtain exclusive edit access to it, unless the querying author does not have edit access to the document, in which case an error message is displayed in the querying author's interface display.

[3]This use of access privileges suggests that perhaps a fourth access privilege, *critique-access*, be provided. This access privilege would allow a user to critique a document but not modify the document proper.

If no other authors are currently editing the node, or the querying author is already editing the node, or another author was editing the node but has not done so for at least the last twenty seconds, then the querying author is granted exclusive edit access. If the node is being edited by another author, then the querying author is so informed by a message in the querying author's interface display. This message includes the name of the author editing the node. The querying author can then choose to cancel the edit request, or to let CES wait until the request can be granted. If several authors have requests pending for access to the same node, then the longest waiting request is granted.

When exclusive edit access to a node is granted, the most recent version of the node is displayed in the querying author's document node window, and the author who last had edit access to the node is informed by a CES message that another author has been granted access. This message contains the name of the author to whom access has been granted. In addition, when the author with the newly granted exclusive edit access first changes the node, then the change of exclusive edit access is logged by CES by pushing a new version of the node onto the node's version stack.

2.4.3.2 Maintaining Exclusive Access

With exclusive access granted, an author will be allowed to maintain access as long as the node is being "used". The "use" of a node means that an author is actively adding or deleting text from the node, moving the keyboard cursor to different positions within the node, or repositioning the node's text within the document node window. If more than twenty seconds lapses between any of these activities targeted at the node, then exclusive edit access to the node expires. CES will automatically re-grant edit access for the author when the author's next editing activity occurs, unless another author intervenes and acquires access. An author is not informed about the temporary loss of exclusive

edit access.

2.4.3.3 Sharing Changes

An author with exclusive edit access to a node can see the effects of the commands targeted at the node in the document node window as they are issued. If the node being edited also happens to appear in the text image window, then the text image is also updated as the commands are issued.

The effects of commands that alter the node text are also eventually seen by all of the authors viewing the document if the affected node is displayed in their text image or document node windows. The text images are updated a node at a time. The display updates include node demarcations as well as node text, so that authors can see who has been making changes to a node as well as what the changes are. In order to compliment the node demarcation *version*, *by*, and *last at* fields, the *from* field of the node demarcation is used to indicate the name of the author who had exclusive edit access to the node prior to the author named in the *by* field. Consequently, these combined fields inform authors about a node's recent history, and are an alternative to examining a node's version stack. Figure 2-2 shows an example of node demarcations for a node that has three versions on its version stack, and a node that has only one version on its version stack.

When an author newly acquires exclusive edit access to a node, and while exclusive edit access is held, the version of the node displayed in the author's document node window is guaranteed to be the most recent. Otherwise, an author's text image and document node windows are updated in a timely fashion as changes are made by other authors. In order to provide the timely sharing of changes, display updates are performed at a rate fast enough to keep authors informed about document updates but slow enough to prevent the updates from

becoming distracting. The actual value for this rate will be determined by testing user's reactions; alternatively, the rate may be configurable within some limits by the user. CES does not attempt to synchronize the updates of different author's interface displays, nor is an attempt made to order the display of changes across nodes in the text image, although changes made to a single node are displayed in the order in which they were made.

2.4.4 Modifying a Document's Structure

2.4.4.1 Editing an Outline

With regard to editing the outline representation of a document's structure, the structure is managed like a node. In order to edit the outline representation of a document's structure, an author must first obtain exclusive edit access to the structure. This process is the same as for a node, described in Section 2.4.3. In addition, authors are informed about a structure's recent history using a structure status display that is similar to the node demarcation. The status display is displayed in the outline window above the outline text and includes *version*, *by*, *last at*, and *from* fields like a node demarcation. The structure status field is illustrated in Figure 2-2.

The only difference between editing an outline and editing a node is that when an edit change is made to a previously parsed outline, a message indicating that the outline has not been parsed since it was last changed is displayed in the outline window of every user viewing the document. Until the outline is parsed without errors, this message will be displayed. The message informs users that the outline does not necessarily describe the structure of the document. In particular, the outline may not agree with the document's text image.

2.4.4.2 Parsing an Outline

In order to parse an outline, exclusive edit access to the structure represented by the outline must be implicitly obtained by the author who issued the parse command. CES automatically does this when the parse outline command is issued. If exclusive edit access is already held because the author who issued the parse command was previously editing the outline, then exclusive access is simply reaffirmed. If exclusive edit access is not already held, then it must be obtained as described in Section 2.4.3. Once exclusive edit access is obtained, it is held for at least the duration of the parsing. After the outline is parsed, the outline and text images displayed in the interface display of the author who issued the parse command will be immediately updated. The interface displays of other authors will be updated in a timely manner.

It should be noted that outline parsing does not prevent authors from simultaneously editing nodes. This is because a document's structure can be updated independent of its content. There are two ramifications of this:

- Nodes which are actively being edited can become orphans when an outline is parsed. As described in Section 2.3.7, such nodes are not destroyed, although they are removed from the document's structure. An author editing a node that has become an orphan is informed in a timely manner about the change in the node's status. Once this happens, the node is truly an orphan, and the author loses exclusive edit access to it. Once a node is an orphan, it is not possible for any author to obtain exclusive edit access to it.

- Nodes that appear in text images can also become orphans. Because changes to a document's structure propagate independently of changes to nodes, it is possible that CES will attempt to update a text image display using orphan nodes. When this happens, the node is included in the text image because the timely propagation of updates ensures that such a transient state will be short in duration.

2.4.5 Orphan Nodes

The destruction of orphan nodes does not require exclusive access because orphan nodes cannot be edited (unless reincorporated into the document's structure); it does require edit access to the document, though. Therefore, orphan nodes are destroyed without author coordination. An attempt to destroy an orphan node that has already been destroyed will result in an error message to the author who issued the inappropriate command. Similarly, an attempt to select an orphan node that appears in a text image for editing will also result in an error message, due to the circumstances described in Section 2.4.4.2.

2.4.6 Document-Wide Edits

Several editing commands, such as the string find-and-replace command, are designed to operate on the text in all of a document's nodes. These document-wide editing commands require that exclusive access to the *entire* document be implicitly obtained by the author who issued such a command. This is in contrast to exclusive edit access which is targeted at a single document component object.

Exclusive access to an entire document is automatically obtained by CES whenever a document-wide edit command is issued, providing that the author who issued the command has write access to the document. In order to obtain exclusive access to an entire document, CES prevents other authors from obtaining exclusive edit access to the document's structure or nodes and waits until the exclusive edit access already held expires. Until the processing for the document-wide edit command is completed, authors other than the one who issued the document-wide edit command are prevented from obtaining access to the document. These authors cannot do any work on the document until the processing of the document-wide command is completed.

The time-consuming nature of many of the document-wide editing commands can deter their use in the CES shared document environment. Three alternatives are therefore provided:

- The scope of an otherwise document-wide editing command is limited to a single node. An author must first obtain exclusive edit access to the node.
- The scope of a document-wide editing command is restricted to only those nodes that are not already being exclusively edited by other authors. CES automatically sequences through nodes, bypassing nodes that it cannot immediately obtain exclusive edit access to.
- The scope of document-wide editing commands is restricted to a range of nodes, as described in Section 2.3.4. CES automatically obtains exclusive edit access to all of the nodes in the range, waiting as necessary before changing any of the nodes in the range.

These alternatives potentially compromise the utility of the document-wide editing commands while offering flexibility in the use of the commands.

2.4.7 Observations

CES provides support for collaboration without burdening authors with complex protocols. By implicitly incorporating structural information about a document into coordination and sharing mechanisms, authors need do little more to edit a shared document then they would if CES was a single user system. In fact, the primary difference between a single user variant of CES and the multi-author variant presented is that the multi-author variant requires that authors understand that there can be multiple authors working on a document. This means that at times, an author will not be able to have immediate access to a document. It also means that authors will have to cooperate and perhaps communicate in order to make decisions about the editing activities.

The approach that CES takes to supporting collaboration sets it apart from

other systems. CES neither provides real-time sharing of authors' contributions, thereby making each editing session into a real-time conference, nor does it allow authors to work in isolation, which is the hallmark of centralized, single user systems. Nevertheless, until CES is tested on a user population, the viability of the CES collaborative environment cannot be established.

2.5 CES Architecture

In Section 1.2, five questions that the designer of a distributed application should answer were posed. For CES, the answers to those questions are as follows:

- Should the application mask its distributed nature?

  CES supports the collaborative efforts of a geographically separated population of authors. In addition, the system supports the locality of ownership, which means that components of a shared document physically reside at different computers. Therefore, in order to make these features useful and apparent to the user, CES does not mask its distributed nature.

- How should the concurrency introduced by parallel computation be exploited?

  CES supports the concurrent editing of a shared document by multiple authors. A high degree of parallelism will be maintained as authors access different (i.e. non-overlapping) portions of a document. When authors attempt to simultaneously access the same portion of a document, parallelism will be reduced because only one author at a time can have access to a particular portion of a document. Only those authors attempting to simultaneously access the same portion of a document are affected by the restricted parallelism; other authors do not perceive any degradation of performance. The dynamics of document sharing dictate that CES supports documents as highly concurrent resources.

- How critical is the resiliency of the application to failures?

  CES is designed to serve a large population of authors. From a

hardware perspective, the probability that CES will fail is proportional to the number of computers in the network. It is intolerable that the isolated failure of an author's computer cause the entire system to fail. Therefore, CES employs schemes that make the system highly resilient to failures.

- How does the application name data in an extremely large, distributed name space?

  A detailed answer to this question will be described in the next chapter. From the user's perspective, though, the network addresses of the various computers in the network are used to specify name subspaces within the network.

- Is there any political, proprietary, or administrative reason to support the locality of ownership?

  The answer to this question is simply, yes. Even though CES supports collaborative work, authors who are working together one day may find the situation disagreeable the next day. Many hypothetical situations for how this can happen can be constructed. Consequently, CES supports locality of ownership.

The architecture that implements the answers to the questions above and the functionality described in the preceding two sections is composed of three subsystems:

- Document registry
- Document library
- Editor front-end

Each subsystem provides a set of logically related services. The *document registry* is the CES network-wide list of documents. The *document library* is the network-wide catalog of document structures and nodes. The *editor front-end* provides the user interface to CES. Each of these subsystems is a fundamental CES resource, and, as a component in a distributed system, each subsystem has a

different combination of performance requirements with respect to *responsiveness*, *resiliency*, and *long-livedness*:

- Responsiveness is a measure of a system's ability to complete a task before the user who initiated the task becomes impatient. This is a purposely vague definition because it is based on the imprecise concept of "impatience", which is typically application sensitive and modulated by user expectations.

- Resiliency is a measure of how tolerant a system is of failures. Resiliency can be discussed in terms of the system's *availability*. A resilient system is *highly available* if its functionality remains undiminished in the presence of failures. The availability of a resilient system *gracefully degrades* if its functionality declines as the number of failures increases.

- A long-lived system maintains state information and data, even during idle periods and between failures, for the life of the system.

In the following sections, the function and performance requirements of each of the three subsystems are discussed.

2.5.1 Document Registry

2.5.1.1 Function

The document registry subsystem is the CES network-wide list of documents which performs the following functions:

- It enforces the uniqueness of document names within a CES system. To provide naming flexibility, the document names can be of arbitrary length and syntax. This allows prefixes and suffixes to be appended to document names so that unique name subspaces can be defined.

- For each document, it is the source of author access privileges and the public access privilege.

- It is used to locate the various physical network sites at which a document's authors reside. This information is used by the other subsystems to focus document relevant communication to only those network sites at which a document's authors reside. This service is particularly important because a document's author population can change over time, and is thereby an efficient alternative to the network-wide broadcasting of messages.

Associated with each entry in the document registry are the following items:

- The name of the document.
- The login name and network address of the document's creator.
- A list of the document's authors, including each author's login name, network address, and document access privilege.
- The public access privilege.

The document registry provides a service that is the inverse of a typical file system. A typical file system maps from a user name to accessible objects. The document registry maps from an object (i.e. a document) to the names of users who have access to the object. Either scheme will allow objects to be shared, but the document registry approach explicitly maintains the list of an object's users. The existence of, and access to, this list is an important collaborative aspect of the CES environment because it allows both CES and authors to readily identify the authors collaborating on a document.

The document registry can be used by any CES user. Most operations service queries and accept as arguments a full or partially specified document name and the querying user's login name and network address. For example, the user "seliger" working on the computer with the network address "oa" is specified as "seliger@oa".

Typical queries check for the existence of a document, return the list of authors

for a particular document, or return the access privilege for the querying user with regard to a particular document. For example, given a document name, the registry is checked for an entry for the document. If an entry is found, then by comparing the querying user's login name and network address against the document's author list, the user's access privilege can be determined. If the user is not on the list, then the user is assigned the public access privilege.

Other operations allow entries for new documents to be added to the registry and existing document entries to be deleted. In addition, there are operations which modify a document's author list and public access privilege. The use of all of these operations is restricted by access privileges, except for the operation to add a new entry. Any user can add a new document to the document registry. As described in Section 2.4.1, this user becomes distinguished for the life of the document as its *creator*. A document's creator has permanent edit access to the document. The entry for a document can only be removed by an author who has edit access to the document. The same access privilege is required in order to modify a document's author list by the addition or deletion of an author, or by changing an author's access privilege. Edit access is also required in order to modify the public access privilege.

2.5.1.2 Performance Requirements

The document registry is a focal point for many CES activities, consequently it must be highly available. Also, it clearly must be long-lived. The requirements for responsiveness can be separated into two categories. Most document registry activities are queries and are often an invisible (to the user) component of a user initiated activity. Queries do not alter the registry. Registry alterations are less frequent than queries and are typically performed explicitly because of a user activity. Assuming that a user who is aware of an interaction with the registry will be more "patient" than a user for whom interaction with the registry is

mini-catalog by a mini-catalog assigned identifier. Mini-catalogs are distinguished from each other by the network address of the computer at which the mini-catalog resides. A network-wide node identifier is therefore the concatenation of these two names. For example, if the identifier for a node within a mini-catalog is "27", and the network address for the computer at which the mini-catalog resides is "xx", then the unique identifier for the node is "27@xx".

The document library supports version stacks, described in Section 2.3.6, for each document structure and node. Each version stack is protected by a mechanism that provides exclusive edit access, as described in Sections 2.4.3 and 2.4.4. Another feature of the document library is that updates to a version stack can be made by an author while other authors read the version stack. This read-while-writing concurrency is provided by only enforcing exclusive edit access to the most recent structure or node version on a version stack. Earlier versions cannot be changed (unless they are reinstated as the most recent versions - See Section 2.3.6), so serializability is not violated by allowing readers to concurrently read these versions while the most recent version is being updated.

A read of a version stack will always return a version of the document structure or node that is either the most recent (if another author does not have exclusive edit access to the version stack) or close to the most recent version on the stack. In addition, consecutive reads of a particular version stack by an author will always return the same version as last read or a more recent version, providing that versions on the stack have not been purged between the reads. Because new versions of a structure or node are automatically created when an author makes the first change to the structure or node after obtaining exclusive edit access to it (See Sections 2.4.3 and 2.4.4), readers will be able to read versions of a structure or node that are fairly indicative of the most recent version.

2.5.2.2 Performance Requirements

The document library must be a long-lived subsystem. This is obvious because documents, the most important data in a CES network, are stored in the library's structure catalog and node mini-catalogs. The responsiveness requirements for the structure catalog and node mini-catalogs are different because it is expected that most CES activities will involve the nodes, and not structures. This expectation is based on the observation that authors will tend to modify a document's content more frequently than a document's structure. Therefore, node mini-catalogs should be more responsive than the structure catalog.

In fact, for node mini-catalogs, it is tolerable to allow the responsiveness of a mini-catalog to decrease as the physical distance between the computer querying a catalog and the computer at which the catalog being queried resides increases. This is because it is expected that most node mini-catalog activities will be targeted at the node mini-catalog residing on an author's machine. This expectation is based on the observation that authors will tend to work on their own contributions. CES support for the locality of ownership stores an owner's node level contributions to a document in the node mini-catalog that resides on the owner's host computer, so that the responsiveness of activities targeted at local mini-catalogs is not an unreasonable design constraint.

With respect to availability, the structure catalog must be highly available because it is not possible to view a document without first reading its structure. If the structure catalog is not highly available, then the failure of the catalog will preclude access to every document in a CES network. In contrast, the failures of individual node mini-catalogs can be tolerated. The functionality of the document library subsystem can be implemented to gracefully degrade due the failure of an individual mini-catalog because access to all of the nodes not maintained in the failed mini-catalog is still possible. When a node mini-catalog

fails, an author will not be able to obtain exclusive edit access to a node in the failed mini-catalog, and the text image for the document will not include the text for nodes at failed sites, but work can still be performed on the accessible portions of the document.

2.5.3 Editor Front-End

2.5.3.1 Function

The editor front-end subsystem maintains the CES user interface and performs the following functions:

- For each CES user, it maintains a CES user interface.
- It monitors changes made to documents by periodically querying the document library and updating each author's interface display.
- It provides the real-time screen editing capabilities as well as CES specific functionality, such as outline parsing and the altering of document view perspectives.

Unlike the document registry and document library subsystems which are instantiated only once for each CES network and exist for the life of the network, an instance of the editor front-end subsystem is created each time a user initiates a CES session. At the end of the session, the instance of the subsystem is destroyed. The instance of the editor front-end serves as the CES processor for its user. When the user no longer requires CES services, the destruction of the user's editor front-end subsystem frees computing resources.

For a user, the editor front-end is the root of all activities. Virtually all of the flow of control in CES passes from an instance of the editor front-end to the document registry or document library. Instances of the editor front-end do not explicitly communicate with each other. Instead, the activities of different

instances of editor front-ends, which manifest as independent threads of control in a CES network, are coordinated by both the document registry and document library. Even though the document registry and document library support aspects of CES functionality, such as the enforcement of access privileges and exclusive edit access, these subsystems are subservient resources for the editor front-end.

Most of the details about the design of the editor front-end have been presented in previous sections. The user interface display maintained by the editor front-end was described in Section 2.2. The dynamics of display updates was described in Section 2.4. Editing functionality was described primarily in Section 2.3. The descriptions provided in each of these sections are the basis for the editor front-end subsystem design.

2.5.3.2 Performance Requirements

The editor front-end is long-lived in the sense that it must maintain its state between failures. It is not long-lived with respect to the life of a CES network because an instance of the editor front-end is implicitly destroyed at the conclusion of its user's CES session. Of the three subsystems, the editor front-end must be the most responsive. This is because it is the CES user interface and because it is the root of most of a user's activities. As the user interface, the editor front-end must provide tactile feedback for user inputs that involve typing text and making basic edits. As the root of most of a user's activities, the editor front-end should not be the subsystem that limits (by poor responsiveness) the completion of activities. The fact that an instance of the editor front-end resides on its user's host computer implies that implementing the subsystem so that it is highly responsive should not be a problem. Unfortunately, the centralization of the subsystem prevents making it more available than the computer on which an instance of it runs. Nevertheless, the failure of an instance of the editor front-

end will not prevent users, other than the instance's user, from accessing CES.

2.5.4 Observations

In Section 1.3.1, distributed applications paradigms were discussed. Based on the description of the CES architecture, it should be clear that CES fits the client/server paradigm. For example, for most activities the editor front-end subsystem is a client for document registry and document library services. The architecture also requires the capability to stably maintain data for extended periods of time, including between failures. In order to implement such an architecture, tools that support the development of long-lived client/server distributed systems are required. The programming language Argus, introduced in Sections 1.3.1 and 1.3.2, is one such tool. As will be discussed in detail in the next chapter, Argus not only facilitates the development of long-lived client/server distributed systems, but as a high level language, also offers the opportunity to expedite the implementation of a distributed system by masking low-level concerns.

Chapter Three

The Programming Language Argus

The programming language Argus is designed to facilitate the implementation of distributed programs for a constrained domain of applications. In particular, Argus is intended to be best suited for applications which do not have severe real-time constraints and in which the viability of long-lived, shared data is crucial. This implies that Argus is not expected to be well suited for high-speed computations or process control, but would serve best as a tool for implementing client/server activities. The Argus application domain is anticipated to be large, distributed, transaction-based systems, such as an airline reservation system. Applications like electronic circuit simulation, where pipelining is used to obtain high-speed performance, or airplane flight control systems, where real-time responsiveness must be guaranteed, do not fall within the Argus application domain. One question is whether Argus will support interactive applications, such as an editor, which have critical, but not real-time, time constraints.

Argus has been under development at MIT since 1977. Over the years, Argus has evolved from an extension to the language CLU [Liskov *et al.* 81] to become a language in its own right. Nevertheless, Argus borrows heavily from CLU, so that the sequential aspects of Argus are virtually identical to CLU. Argus is an object-oriented programming language which supports data as well as procedural and control abstractions. Where Argus and CLU diverge is in Argus's support of distributed applications. Paramount in the Argus design goals is that the language address and readily handle the issues that arise in distributed systems design. These issues concern concurrency, failures, and distribution.

3.1 Fundamental Linguistic Constructs

3.1.1 Basic Abstraction Facilities

Argus uses the basic abstraction facilities provided by CLU. These include:

- Procedures — Argus procedures are similar to procedures in most programming languages, and are the fundamental unit of procedural abstraction. A procedure performs an activity by executing a series of operations and references to other procedures, Procedures accept arguments and can return results or terminate with an *exception*. An exception is an alternate return path that is used to inform the procedure's caller that normal termination was not possible. Multiple exceptions can by specified and are distinguished by name. Exceptions can return results.

- Clusters — An instantiation of a cluster is called an *object*. A cluster encapsulates an internal representation of a data structure with externally accessible operations that examine and alter the state of the instantiated data structure. Cluster operations accept arguments and can return results or terminate with an exception. A cluster abstraction provides a barrier between the implementation of an object (i.e. its physical representation) and the application of the object. A programmer who uses an object is consequently buffered from implementation details and needs to understand only the semantics of the cluster operations.

- Iterators — An iterator is a control abstraction that produces a sequence of objects, one at a time, to its caller. The actual sequence of objects produced is based on arguments to the iterator. Control is shared between the iterator and its caller. Starting with the initial call, and upon each successive call until the iterator terminates, the iterator yields to its caller one of the objects in the sequence. In

addition to yielding objects, iterators can terminate with an exception.

Procedures, clusters, and iterators are employed in Argus to provide abstractions for the sequential aspects of a program. Argus also adapts these abstractions for use with specifying portions of a program that deal with concurrency. For example, a procedural abstraction can encapsulate several concurrent activities. In the following sections, the use of clusters, procedures, and iterators as abstractions for both sequential and concurrent aspects of an Argus program will be discussed in detail.

3.1.2 Guardians

In Argus, a distributed application is built on a virtual network of virtual processing nodes, called *guardians*. Installed on top of an actual hardware configuration, the network of guardians is an abstraction of the underlying hardware. A guardian, acting as a virtual processor, is capable of communicating with other guardians as well as performing local multi-processing chores. In addition, a guardian encapsulates state information which is accessible only by the guardian, but which can be made available to other guardians through inter-guardian communication. The result is that a guardian is an abstraction of a centralized resource that, by communicating with other guardians, becomes a component in a distributed system.

The abstraction for a guardian is built from five basic elements:

- Guardian State — The guardian state is contained in an object or objects that are the instantiations of one or several cluster types. In addition, a guardian state object can be declared to be *stable*, in which case the state of the object will survive, with high probability, a

crash[4] of the guardian.

- Recovery Code — After a crash and before it is on-line again, a guardian executes recovery code. The recovery code reconstructs or initializes volatile guardian state objects, typically by using recovered stable objects.

- Background Code — Guardian background code is executed while the guardian is active to perform background processing.

- Creator Operations — Creator operations are like cluster operations that create new objects. Creator operations are typically externally called, and are used to create a new guardian object and to initialize its state. Guardians typically contain one or just a few creator operations.

- Handler Operations — Handler operations are like cluster operations in that they are the only way to externally access a guardian object. Handler operations are usually externally called, though a guardian's handlers can call each other, and can transfer the thread of processing control to the guardian object. Guardians typically contain a variety of handler operations.

Though guardians, like clusters, are data abstractions, a major semantic difference between them is that guardian objects are active processing entities, whereas cluster objects are passive. The activeness of guardian objects is exemplified by the fact that they use multi-processing to simultaneously execute ongoing background tasks while responding to transient handler activities. In

[4] A guardian crashes when underlying hardware malfunctions, or when an application, Argus run-time system, or operating system software error occurs which is not or cannot be appropriately handled, thereby preventing further execution of the guardian code.

addition, guardians employ a dynamic process structure that allows an essentially unconstrained number and combination of handler operations to be concurrently processed. In contrast to static process structures (i.e. a fixed number of processes), the dynamic process structure results in highly responsive guardians, increased concurrency, and less overall latency in Argus programs [Liskov *et al.* 85].

3.1.3 Atomic Actions

Argus explicitly supports concurrent processing, but the requisite concern of process synchronization is further complicated in the distributed domain, where processes span computers. Additional complications are introduced by the Argus goal of attempting to ensure the consistency of long-lived data. Argus uses *atomic actions* to unify the mechanisms for dealing with each of these problems. Atomic actions synchronize processes and ensure data consistency and longevity across computations and failures.

An atomic action is an action which is *serializable* and *recoverable*. Serializability means that the effects of the potentially interleaved component operations of concurrently executed atomic actions are the same as if each action was executed in some serial order. Recoverability means that either all of the effects of an atomic action are made visible to other actions, or none of the effects are visible [Weihl 84]. Thus, the execution of an atomic action is logically indivisible. An action that runs to completion and makes its effects known is said to have *committed*. An action which is not able to successfully complete all of its operations undoes the effects of any completed operations by *aborting*.

By allowing actions to span guardians, Argus provides a mechanism for encapsulating activities of individual computers into a single logical activity. In order to control these activities, action structures can be specified that dictate

the flow of actions and modularize the scope of an action. In order to do this, Argus provides primitives for developing structures of parent actions and associated children ***subactions***. The root of an action is a *topaction*. A topaction's subactions are children of the topaction, and each subaction can in turn be a parent of children subactions. An action commits contingent upon the committal of its parent action. The aborting of an action will not force its parent action to abort, but the abort of a parent action does force the aborting of its children.

The atomicity of actions with respect to other actions is enforced on a strictly hierarchical basis, so that action structures are developed in a tree-like fashion. Within this framework, unrelated actions and sibling actions can run concurrently, but a parent action cannot run concurrently with its children. The result is that unrelated actions and concurrent sibling actions are atomic with respect to each other, but an action can view the effects of its ancestors. This makes sense when the semantics of commits and aborts are considered. An action that views the effects of its ancestors can only commit if its lineage back to an ancestral topaction commits, thereby preserving the atomicity of the action. An action that views the effects of its ancestors and then aborts is clearly not a concern.

Sub-trees, with independent roots, are also possible within the action hierarchy. This is achieved by using *nested topactions*. A nested topaction is independent and unrelated to the topaction and any of its relatives, within whose lineage the nested topaction resides. Thus, the view of a nested topaction cannot violate the atomicity of the surrounding action family. Nested topactions extend the modularization of the action system. In particular, nested topactions are intended to allow activities to run to completion as an adjunct to the action in progress.

Work in an Argus program is therefore accomplished by the concurrent processing of various threads of control, each rooted in a topaction, that spread as actions throughout the network of guardians. Guardians are capable of multiprocessing, so an individual guardian can be involved with a number of related and unrelated actions. These actions are serializable and recoverable, so data consistency is maintained during active processing of actions. In addition, the consistency of long-lived data, which resides in stable storage, is preserved by the action system by updating a stable object's state only if the action that modified the volatile copy of the object commits. The committing of an action and updates to stable storage are coordinated by a two-phase commit protocol, executed when a topaction terminates, which ensures that an action that commits updates stable storage, and that only if stable storage is updated will an action actually commit. The failure to successfully update stable storage results in the aborting of the action [Oki, Liskov, & Scheifler 85].

3.1.4 Atomic Objects

The atomicity of actions in Argus is actually implemented by *atomic objects* [Weihl & Liskov 82]. An atomic object synchronizes actions that invoke operations on a shared object so that the actions are serializable. Atomic objects also use a version stack (which is a form of an undo-log [Gray *et al.* 81]) that allows uncommitted changes made to an object to be undone, or recoverable. One reason for using atomic objects is that they both encapsulate the serial and atomic aspects of an object's behavior within a single abstraction. This makes it easier for the programmer to design for concurrency by locating the synchronization mechanism with the data requiring synchronized access [Bloom 79]. Another reason is that the atomicity of action structures is guaranteed when the shared object is atomic. This simplifies the design of action structures.

In order to synchronize actions at the object, a variant of the standard two-phase

locking scheme (not to be confused with two-phase commit) is used [Liskov & Scheifler 83]. Object operations are classified as read operations - operations which examine, without altering, an object's state, and write operations - operations which alter an object's state. Before performing a read operation, an action must acquire a read lock on the object. This lock will be granted if a write lock is not currently held by another action. Before performing a write operation, an action must first acquire a write lock on the object. This lock will be granted if no locks are currently held by another action. Thus, the lock semantics, hereafter referred to as "readers/writer", allows concurrent readers, or only one writer and no readers.

During the first phase of two-phase locking, an action acquires locks for its component operations as the operations are executed. During the second phase, locks are released. Once a lock has been released, an action does not acquire another lock. In standard two-phase locking, a lock can be released after the execution of the action's component operation for which the lock was acquired. This policy can cause aborts to cascade among actions because an action can potentially examine an object that has been modified by an operation executed by an earlier, but unterminated, action [Wood 79]. In order to preserve consistency, the examining action will have to be aborted if the unterminated action subsequently aborts. Argus consequently constrains the second phase of the protocol so that locks are released only when an action commits or aborts. This approach reduces concurrency by requiring that locks be held for the duration of an action at the benefit of preventing the potentially time-consuming cascading of aborts.

The action hierarchy described in Section 3.1.3 is superimposed on the object locking semantics in terms of dictating how locks propagate. Whether or not an action can acquire a lock that is already held is a function of the relationship between the action trying to acquire the lock and the action that holds the lock.

Locks held by the ancestor of an action are propagated to the action, but locks held by an unrelated or concurrent sibling action will require that an action wait for the lock's release. In conjunction with an action's acquisition of a write lock, a copy of the version of the object that is viewable to the action is made and pushed on the version stack for the object. All subsequent action operations are targeted at this new version. When the action commits, its parents inherit the new version. When the action's ancestral topaction commits, the new version of the object becomes the base version. If the action aborts, then its version of the object is discarded and the previous version is reinstated as the working version.

3.1.5 User Defined Types

Argus offers mechanisms and constructs that allow the programmer to specify new data types. Using cluster abstractions, non-atomic and atomic built-in types can be used as the basis for programmer defined object representations and operations. If all of the types used in the cluster definition are atomic or immutable[5], then the cluster type is guaranteed to be atomic. If non-atomic types are used in the definition of a cluster that is to be atomic, then the programmer must ensure the correctness of the design with respect to atomicity. A wide range of cluster types can be created by defining clusters using built-in types and other cluster types.

There are instances when the programmer requires more flexibility in defining atomic types than can be had by using built-in atomic types. In order to support these needs, Argus provides the *mutex* type generator. The mutex type generator binds synchronization primitives with another data abstraction to create a higher level abstraction. The mutex portion of the resulting abstraction is a semaphore-like construct. When using a mutex object, an action must first

[5]The state of an immutable object cannot change after the object is created, therefore, an immutable object does not require locks for atomicity.

seize the mutex in order to examine or alter the *value* of the mutex, which is the object to which the mutex is bound. Only one action can seize a mutex at a time, but once seized, an action can temporarily release possession of the mutex by executing a *pause*. The pausing action is suspended for a system-determined amount of time, allowing another action to seize the mutex. When awakened, the paused action attempts to regain possession of the mutex, which may not be possible if another action holds it. If there are multiple paused actions, then the order of reseizes is determined in a fair manner.

Mutex objects also have an operation that can be used to notify the system that the value of the mutex has changed. When this operation is invoked, the value of a mutex object declared to be stable will be guaranteed to be copied to stable storage before the invoking action commits. If the value of the mutex is not successfully copied to stable storage, then the action will be forced to abort.

3.2 Using Argus

Like most programming languages, Argus offers a set of linguistic tools, but does not explicitly prescribe the use of the tools. In this section, suggestions and observations, from the literature and first-hand experience, are provided that give insight into the application of Argus. In particular, the focus is on techniques and methods that influence program behavior with respect to concurrency and other time-based measures of performance.

3.2.1 Guardians

A potential source of confusion in Argus is the difference between guardians and clusters. One key difference is the active nature of a guardian. Clusters define objects that are passive and execute code only when an object operation is invoked. A guardian can execute code, in background, independent of calls to

the guardian operations (i.e. creators and handlers). Another key difference is that only guardian state objects can be declared as stable. This means that a guardian whose state is contained in an object of type A can declare the instantiation of A to be stable, but the cluster definition for A cannot specify that it be stable. In connection with this, recovery code - the code that rebuilds or reinitializes volatile state from stable state - is only explicitly run for guardians.

The semantic difference between guardians and clusters reflects the role of guardians as active processing entities, and clusters as passive data structures. For example, when a hardware failure occurs, guardians running on the computer crash and then recover when the hardware recovers. It makes sense to describe crash and recovery semantics for a virtual machine, such as a guardian, but it is not clear that equivalent semantics are appropriate for passive data structures, such as cluster objects.

Guardians are centralized, active objects that, by communicating with each other, are used to build a distributed program. The initial set of a program's guardians is created using topactions inside of procedures or cluster operations to call guardian creators. When the topaction that calls a creator commits, the guardian created becomes permanent. Guardians can in turn be used to create other guardians, as the result of code executed by the guardian in background, or in response to a handler call. Guardians can also be destroyed. In contrast to cluster objects which are effectively destroyed by the reclamation of memory when no other objects reference it, guardian destruction is explicit. The only way to destroy a guardian is to have the target guardian destroy itself. This is done by using a system call to a self-destruct routine. External control over guardian destruction is typically implemented by the use of a handler operation that performs the self-destruct.

Guardians can be designed for long-lived or ephemeral services. In either case, it may or may not be necessary for the guardian to have state information that can be recovered in the event of a crash. If a stable guardian state is required, this can be achieved by declaring the desired guardian state objects as stable. If volatile state information is maintained by the guardian, then a recovery routine can be specified. After a guardian crashes, any stable objects are copied into volatile memory, and the guardian's recovery code is executed to initialize the volatile state objects, perhaps by using the copies of the stable objects. If recovery fails, the guardian can crash and then try again. When recovery terminates, the guardian's background code is started-up and the guardian is ready again to receive handler calls.

The background code that a guardian executes can be an arbitrarily complex structure of operations or actions. Typically, this code executes in some kind of loop, causing background processing to periodically cycle through a set of states. Background is useful for maintaining action structures that originate in a guardian. It is also useful for various kinds of busy and non-busy waiting chores. For example, a guardian's background code can monitor handler calls, and can initiate activity independent of handler calls, perhaps in response to a particular handler call or a change in the guardian's state due to a handler call.

3.2.2 Atomic Actions

3.2.2.1 Remote Procedure Calls

Argus uses *remote procedure calls* to implement calls to guardian handler operations [Liskov & Scheifler 83]. A remote procedure call is a communication mechanism that implements a procedure call-like semantics. A remote procedure call consists of an invocation message and a reply message. These messages are analogous to a procedure call and return. Each remote procedure call invocation

message is logically sent and acted upon by its receiver exactly once. The reply for the invocation message is either logically received exactly once by the sender, or the sender is informed of the inability to complete the call.

Remote procedure calls are employed by Argus because of the similarity between the one-invocation/one-reply property of remote procedure calls and the recoverability property of atomic actions. In fact, Argus fuses the two mechanisms by implicitly encapsulating a remote procedure call inside of a subaction. Handler calls that fail can therefore be aborted without causing the action that made the call (which is the parent of the handler call action) to abort. Similarly, handler calls can be aborted without causing the calling action to abort. The result is that the implicit modularization of handler calls simplifies the design of action structures.

3.2.2.2 Specifying Concurrent Actions

Specifying concurrent actions can be done using two similar constructs. One construct, the *coenter*, can be applied to any combination of topactions and subactions. Each action is executed concurrently with the others but is constrained by the action hierarchy described in Section 3.1.3. The second construct is a variation of the coenter, and uses an iterator to create a set of concurrent sibling subactions. For each object returned by the iterator, a new action is created. When the iterator terminates, the actions are run concurrently, with each action possibly initiated using one of the iterated objects.

Concurrent actions derived from the same coenter can be terminated in two ways. All actions can be run to completion, independently committing or aborting, or the termination of an action can force the aborting of the other actions. An action that terminates and forces the aborting of the other actions need not abort itself.

3.2.2.3 When to Use Concurrent Actions

There are two basic scenarios in which the use of explicitly structured concurrent actions is best suited. The first scenario is one in which the actions are concerned with independent tasks that do not share data and are not related to each other by a sequential ordering. For example, a task that updates several distinct data bases might be a candidate for concurrency. The action structure shown in Figure 3-1 implements concurrent updates to three separate data bases where the order of the update completion is not important.

```
enter topaction
  coenter
    action
      update data base #1
    action
      update data base #2
    action
      update data base #3
  end
end
```

Figure 3-1: Example of an Action Structure using Coenter - The three actions shown are executed concurrently. Each action commits or aborts independently of the others.

The second scenario is one in which there are a number of independent resources that can perform a task, but only a subset of the resources are actually required to complete the task. For example, consider a situation in which there is a task that monitors a number of input devices. The task is interested in processing the input of the first device that responds to the task's query. To do this, the task issues concurrent queries to all of the input devices, but processes the input of the first device to respond, while aborting the other actions. The action

structure shown in Figure 3-2 characterizes this scenario.

```
enter topaction
  coenter action
    for each input device iterated
      issue query for input action
      when get first reply, abort other queries
    end
  end
end
```

Figure 3-2: Example of an Action Structure using Coenter and Iterator -

For each input device iterated, a query action is created. When all of the input devices have been iterated, the query actions are executed concurrently. The actions abort independently of each other, but if an action commits, then it will force the aborting of the other actions.

3.2.2.4 Managing Action Structures

The primary cost of using actions is the number, or size, of messages required to query about the state of locks or to propagate information about the committing of actions. Messages are required when a topaction commits, because all of the guardians on which the topaction's descendant actions ran must communicate as part of a two-phase commit protocol. This protocol implements the action hierarchy commit/abort semantics, described in Section 3.1.3. Similarly, messages are required to propagate lock information as dictated by the semantics of the action hierarchy.

The Argus implementation has been designed to minimize the number of

messages required, particularly with regard to lock propagation. Locks acquired by a guardian handler are not automatically propagated back to the parent, nor are locks propagated to handler descendant actions that are executed on other guardians. Rather, explicit lock propagation queries are generated when an action needs to use an object that is locked by a non-local action. The use of explicit queries reduces the message overhead that would otherwise be required to propagate locks [Liskov 84].

Despite Argus implementation optimizations, the span of an action structure, as measured by the number of guardians involved in processing the action, is a measure of the potential performance cost of the action structure. The programmer must be aware of this fact in order to develop efficient programs. This requires that the roles of guardians be clearly developed to minimize the unnecessary physical separation of a logical function. In addition, action structures need to be scrutinized to make sure that only those guardians required to perform a task are actually involved in the task.

3.2.2.5 Short Actions

As an Argus topaction is processed, it spreads throughout the system acquiring locks. The atomicity of the topaction, enforced by the locks, can potentially prevent the progress of other topactions due to lock conflicts on shared data. Although *deadlock*, the irreconcilable conflict of unrelated actions where each action holds a lock required by the other, is a concern, the problem of interest in this discussion is the rate of overall system progress.

One approach to the problem is to use highly concurrent data types. This will be discussed in Section 3.2.3, but it is often the case that even with highly concurrent data types, it may still be difficult or impossible to ensure that conflicts that prevent system progress will not occur. Another approach is to

restructure an action so that it is composed of a set of shorter (in duration) actions. There are several ways to do this [Gifford & Donahue 85]:

- It may simply be the case that the scope of the action is too broad, and that decomposition into shorter action may be entirely appropriate.

- System invariants can be relaxed to reduce the coupling between actions. For example, consider a system invariant that specifies that the sum of the money in the bank accounts contained in account data bases must be zero dollars. The steps to transfer money from an account in one data base to an account in another data base must occur within a single atomic action, or the system invariant could be violated. An alternative to the global preservation of the system invariant is to locally enforce the invariant. Each bank data base could use a transfer account that keeps a running tally of the total amount of money in transfer between banks. The transfer of money between banks could then take place as two separate actions. One action would debit the source account while adding the amount debited to the local transfer account. The second action would credit the target account while subtracting the amount credited from the local transfer account.

  The cost of relaxing the original invariant is that funds will effectively become inaccessible during the time between execution of the debit and credit actions. Therefore, a client who transfers money from one account to another may temporarily be prevented from accessing the money.

- *Persistent actions* can be used. A persistent action is an action that is a collection of independent component actions. A component action may in turn be a persistent action. A persistent action terminates when all of its component actions, guaranteed to run only once, have been run to completion. By using component actions to move the system from one consistent state[6] to another, persistent actions can view the partial effects of other persistent actions without violating atomicity, system invariants, or causing cascading aborts.

[6] A consistent state is one in which all of the system invariants are preserved.

Argus does not provide explicit linguistic support for persistent actions, though the adaptation of Argus to accommodate persistent actions is a topic of current research. The decomposition of large actions and the alteration or relaxation of system invariants do not require additional linguistic support, and should be readily employed in Argus programs.

3.2.3 User Defined Types

3.2.3.1 When to Use a User Defined Type

There are four major reasons why a programmer would want to design a new data type:

- To create a new abstraction for a specific application (e.g. an atomic queue with create, enqueue, and dequeue operations in lieu of the atomic array built-in type).
- To create an object that is not atomic but provides mutually exclusive access with respect to actions (e.g. there are instances in which it necessary to violate atomicity and allow concurrent actions to communicate; such communication might require synchronized access to a non-atomic object).
- To create a stable atomic object that is partitioned in a way that minimizes the amount of object state that must be copied to stable storage after an action that affects the object commits (e.g. a large data base where it is desirable to copy only an altered record, and not every record, to stable storage when the altering action commits).
- To create an atomic object that has more than the readers/writer concurrency (e.g. an atomic queue that allows concurrent enqueues and dequeues).

The specification of new atomic types for the first reason is a straightforward process of designing the object representation from Argus built-in atomic or immutable types, and providing relevant operations. Thus, a queue abstraction

could be simply built using an atomic array, with the atomic array add-element and remove-element operations serving as the basis for the queue's enqueue and dequeue operations. To address the second reason, a non-atomic object bound to a mutex offers a mechanism which provides mutually exclusive access to a non-atomic object. To address the third reason, abstractions that incorporate a mutex or a set of mutex objects can be used to partition the object representation and exploit the mutex notify-value-changed operation to control the amount of an object's state that gets copied to stable storage. The specification of new types for the fourth reason is more complicated, and is discussed next.

3.2.3.2 Increasing Concurrency

There are three basic methods for designing highly concurrent (i.e. more than readers/writer concurrency) atomic types:

- The abstraction representation can be structured to minimize the probability of conflict.
- The abstraction can be defined so that operations inherently offer more concurrency.
- Deterministic constraints on the abstractions operations can be relaxed.

The first two methods can be classified as ways of manipulating the granularity of locks on an object [Gray 81], whereas the third method assumes that the semantics of a type's application can be examined and adjusted (e.g. the semantics of an atomic array "top" operation may be relaxed so that some element near the top of the stack, rather than the element actually at the top of the stack, is returned.) By manipulating lock granularity, the programmer deals primarily with internal aspects of the type abstraction (i.e. the object representation), but relaxing determinism impacts both internal and external

aspects of the abstraction. Thus, relaxing determinism to increase the concurrency possible with an existing abstraction can be costly in terms of the impact that the resulting semantic changes have on other portions of the program.

The first method is fairly simple to implement and is best suited for objects with a large, regular superstructure composed of smaller substructures (e.g. an MxN dimensional array can be implemented as an M element array superstructure where each element of the array is an N element array substructure). By using an algorithm (e.g. hashing) that causes, with high probability, concurrent operations to be targeted at different substructures, the average amount of concurrency for the object can be enhanced. Objects of this type are typically composed of immutable superstructures with atomic substructures. There is no need for locks on immutable objects, so the superstructure does not restrict concurrency and the atomic substructures provide atomicity. The second and third methods require non-trivial implementation techniques that merge non-atomic types with atomic types and the mutex type. The non-atomic types provide unrestricted concurrency, the atomic types provide recoverability[7], and the mutex type provides synchronization and operations for signalling updates for stable storage.

Though Argus provides a variety of built-in atomic types, Weihl demonstrates that the built-in atomic type *atomic variant* can be extremely useful for the design of user-defined atomic types [Weihl 84]. Atomic variant types are specified as a set of tags, each associated with a type. In an atomic variant, only one tag can be the "active" tag, and the object bound to this tag is the *value* of the atomic variant. Operations exist that create new atomic variants in some

[7] In Argus, the incorporation of built-in atomic types into non-atomic structures is the only way to provide recoverability for a user-defined atomic type [Weihl 84].

base state (i.e. tag and value specified as arguments to the operation), that change the state of an atomic variant (i.e. target object, new tag and new value specified as arguments to the operation), and that return the state of the tag or the value associated with the tag. The operation to change the state of an atomic variant is classified as a writer, the other operations are all classified as readers.

A property of Argus atomic types that seems to violate atomicity is the fact that if an action that creates an atomic object aborts, the object will survive in the state in which it was created. However, the Argus designers claim that the persistence of objects is easier to deal with than if the objects were "uncreated" [Weihl 84].[8] Weihl proposes that the atomic variant type can be used to mask the persistence of newly created objects and provide recoverability for user-defined atomic types. The strategy is to build the framework for the new type using a mutex type with a non-atomic type value, and use an atomic variant type to encapsulate the state of the resulting object and provide recoverability. For example, concurrent writes to an array-like object can be provided by a type that is built from a mutex encapsulated non-atomic array with an atomic variant as the array element type.

In Weihl's strategy, the atomic variant type contains at least two tags - one tag is used to identify the base state of the atomic variant, the other tags are used to identify "active" states of the atomic variant. An action that adds a new atomic variant to the non-atomic framework first creates the atomic variant to be added

[8]This claim is reasonable if the following is considered. There are only two ways that an action can view an object that has been created by another action that subsequently aborted. The first way is if the new object is bound to a non-atomic object, shared by the two actions. If this is the case, the shared object may not exhibit correct atomic behavior, but the language only guarantees the atomicity of actions if shared objects are atomic. The second way is if the new object is passed by the aborted action to its parent action by an exception. If this is the case, the parent action presumably "understands" the semantics of the exception and the fact that it is viewing the effects of an aborted action.

in the base state, and then uses the change operation to set the atomic variant's tag (and associated value) to an active state. The mutex for the framework object is then seized, the new atomic variant is added, and the mutex notify-value-changed operation is invoked. If the action commits, the altered framework object is copied to stable storage (if the object is declared as stable), the write lock acquired by the change operation on the atomic variant added is released, and the state of the atomic variant is the state resulting from the change operation. The added atomic variant can then be examined by other actions. If the action aborts, then the version of the atomic variant created by the change operation is discarded, returning the atomic variant to its base state. The operations for the new type know that atomic variants in the base state are remnants of aborted actions, and simply ignore them.

An inherent inefficiency with the above technique is the "remnant" objects that are left by aborted actions. With time, the number of remnant objects can grow to substantial proportions, perhaps overwhelming the number of "active" objects, thereby wasting memory space. This problem can be solved by using a "garbage collecting" operation that is internal to the type's abstraction. The garbage collector would be periodically invoked by an externally accessible operation, but would run in its own topaction so that its view of the atomic variant will not include changes made by the action that called the externally accessible operation.

3.2.4 Summary

In this chapter, the programming language Argus was introduced and the language's fundamental linguistic constructs were explored:

- Guardians - the active entities in an Argus program.
- Atomic actions - the construct for specifying control structures, atomic objects - data types that implement the atomicity of actions.

- User defined-types - data types that offer the user flexibility, were presented with detail relevant to the remainder of this thesis.

In addition, some insight into the application of these constructs was provided as a foundation for the CES implementation, discussed in the next chapter.

Chapter Four

Collaborative Editing System Implementation

4.1 Implementation Overview

CES is implemented as a single distributed program written in Argus. Argus currently runs on a network of DEC VAXs equipped with VAXstation 100 terminals. The CES implementation is not VAX-specific except in its use of Argus - as the Argus environment becomes more heterogeneous, CES should port unchanged except for terminal I/O. CES terminal I/O is currently implemented for the VAXstation 100 terminal.

The three CES subsystems described in Chapter 4 are implemented as four guardians. The editor front-end subsystem is composed of an editor front-end guardian and a guardian for terminal I/O, the latter referred to as the I/O guardian. The document registry and the document library are each composed of a single guardian. In addition, a fifth guardian, the CES system catalog guardian, implements the CES system catalog. The CES system catalog is used by CES guardians to locate other CES guardians. Finally, a configuration utility provided by the CES system catalog is used by CES administrators to configure a CES installation. The configuration utility allows administrators access to any site in a CES network in order to create or destroy guardians, and to update the CES system catalog as appropriate.

Using the CES configuration utility, a CES *installation* can be created. A CES

installation consists of a set of CES guardians, possibly physically distributed across an underlying network. A CES installation requires at least one instantiation of the document registry and CES system catalog guardians, and one instantiation of a document registry guardian at each computer in the underlying network that supports a CES author. Computers that are intended to provide read-only CES services need not have a document registry guardian. Instantiations of the editor front-end and I/O guardians are created by CES each time a user initiates a CES session. Both of these guardians are created on the same computer. At the end of a user's CES session, these guardians are destroyed by the system.

The CES configuration utility can also be used to expand CES installations in several ways:

- New author sites can be added by creating a document library guardian at the new site.
- Copies of the document registry guardian can be arbitrarily scattered throughout the underlying network. These copies behave as a logically centralized resource, but enhance the availability and responsiveness of the document registry through replication.
- Copies of the CES system catalog guardian can be arbitrarily scattered throughout the underlying network. As is the case with the document registry guardian, copies of the CES system catalog guardian enhance the availability and responsiveness of the resource.

Each of the guardians implements the subsystem abstractions described in Chapter 2. In the remainder of this chapter, the implementations of each subsystem will be described. Section 4.2, on the the document registry guardian, emphasizes the replication techniques employed to make the document registry highly available. Section 4.3, on the document library guardian, focuses on the user defined data types that are used to provide highly concurrent access to the document library. Section 4.4, on the I/O and editor front-end guardians,

describes the emulation of atomic terminal I/O and the action structure of the editor front-end. The CES system catalog is discussed in Section 4.5.

4.2 Document Registry

The document registry guardian implements the document registry subsystem. Using replication techniques, the guardian can be replicated throughout the installation, thereby providing a highly available and responsive resource. The guardian design offers a moderate amount of read-write and write-write concurrency, as most document registry operations are read-based queries. The availability and responsiveness of the service provided by the document registry resource are a function of the number of guardian replicands and the availability and responsiveness of individual replicands.

There are two aspects to the functionality of the document registry: the management of the list of documents and the management of document registry replicands. The document list is contained in a single, stable object of type *registry*. A set of handlers provides query and registry altering operations. The replicands are managed with stable objects, one of type *reg_status*, and another of type *reg_list*. A set of handlers coordinates the activity of the replicands.

4.2.1 Managing the Document List

4.2.1.1 Data Structures

The registry type is a two-dimensional structure consisting of an immutable array of subregistries, where each subregistry is an atomic array. The entry for a document is found by using the document name to hash into a subregistry, and then searching the subregistry for the desired entry. This structure increases read-write and write-write concurrency because the immutable array reduces the

granularity of locks from the registry level to the subregistry level. The hashing algorithm is designed so that with high probability, registry reads and writes will not hash into the same subregistry. (This probabilistic approach to designing highly-concurrent user-defined atomic types was discussed in Section 3.2.3.)

The immutable array of subregistries is actually a non-atomic array. When the document registry guardian is created, the array and its associated subregistries are created. Subsequent guardian operations on the array do not alter its size and do not change its subregistry bindings. The result is that the array is treated as an immutable object. The advantage of using an array in this fashion, as opposed to a *struct*, which is the Argus built-in immutable array type, is that the size of the array can be dynamically specified by the creator of the document registry guardian. The size of the array can be chosen so that for an anticipated population of documents, the average probability that two random writes to the registry will hash into the same subregistry will be low. Given the hashing function used in the document registry guardian, this probability is, on the average, 1/N for N subregistries, and decreases by $1/N^2$ upon the addition of the $N+1^{st}$ subregistry.

The specification of the registry and associated types is shown in Figure 4-1. The *timestamp* field of the *subregistry* type is used by the replication algorithm, and will be discussed later. The *empty* field of the *reg_entry* type is used for entry recycling purposes. As described above, in order to preserve the immutable properties of the registry array, its subregistry bindings cannot be altered. This restriction poses a minor obstacle to the process of deleting a registry entry because in Argus there is no efficient way, short of creating a new array object, to remove an element from an array. This approach would violate the immutability of the registry-to-subregistry bindings. One alternative is to compress the array object by iterating over the array elements, starting from just above the element to be removed, moving each element down by one, and then

removing the top-most element. The problem with this scheme is that it is inefficient when subregistries are large.

A better approach is to recycle deleted subregistry entries. This approach assumes that it is less costly to use memory for "deleted" subregistry entries than it is to take the time to squeeze out such entries using a compressing technique. When an entry is deleted, the entry is simply marked as "empty". When a new entry is added to a subregistry, the subregistry is first scanned to see if any of its entries are "empty". If such an entry is found, then it is appropriately overwritten and recycled as the new entry; otherwise, the new entry is appended to the end of the subregistry. The major advantage of this approach is that it is free. This is because when the entry for a new document is added to the registry, the subregistry into which the document name hashes must first be scanned to verify that a document with the same name does not already exist. With subregistry scanning as an inherent aspect of the operation to add an entry, it is a trivial matter to test each entry's empty field, and then remember an entry which is empty.

The remaining fields of the registry entry maintain the information, described in Section 2.5.1, that is associated with each document registry entry. The *creator_name* and *creator_site* fields are the login name and network address of the document's creator. The *authors* field is the list of the document's authors, including each author's login name, network address, and document access privilege, and the *public_priv* field is the public access privilege. The *sids* field is the list of identifiers for each copy of the document's structure. This list is essentially the list of network addresses at which the document's authors reside, though the structure identifier is used for rapid access to the associated copy of the document's structure. This will be further described in Section 2.5.2. It should be noted that network sites that support more than one of the document's authors is represented only once in this list. The *author_cnt* field of *struct_id_rec*

is the count of the number of authors at the site. When the count drops below one, the site's address is removed from the list.

```
author_info = record[name: string,
                     site: network_address,
                     priv: privilege]

list_authors_info = atomic_array[author_info]

struct_id_rec = atomic_record[sid:      struct_id,
                              author_cnt: int]

list_struct_ids = atomic_array[struct_id_rec]

reg_entry = atomic_record[empty:          bool,
                          doc_name:       string,
                          creator_name:   string,
                          creator_site:   network_address,
                          sids:           list_struct_ids,
                          authors:        list_authors_info,
                          public_priv:    privilege]

sub_registry = atomic_record[entries:    atomic_array[reg_entry],
                             timestamp: very_big_num]

registry = array[sub_registry]
```

Figure 4-1: Specification of Registry and Related Types

4.2.1.2 Handlers, Background, Recovery, and Action Structures

A variety of document registry querying and modification operations are supported by the document registry guardian handlers. The informal specification of these handlers appears in Figure 4-3. The action structures employed by the implementation of the handlers is very simple. In fact, except for the interactions with the replication algorithm, each handler simply executes a sequence of operations on the registry object. The guardian does not employ

background code, and recovery from crashes is implicit in the restoration of the guardian's volatile state from stable objects. A recovery procedure is not required.

4.2.2 Managing Document Registry Replicands

4.2.2.1 Gifford's Weighted Voting Algorithm

In order to enhance reliability and system responsiveness, multiple document registry guardians, each with a copy of the document registry, can be created by the CES system administrator. The coordination of document registry guardian replicands is managed using Gifford's Weighted Voting algorithm [Gifford 79]:

> Copies of replicated data are assigned a vote, or weight, greater than or equal to zero, with at least one copy assigned a non-zero weight. Each copy of the data has an initially nil timestamp field associated with it. Based on the the sum of the weights of each copy of the data, a read quorum size and a write quorum size are defined such that the sum of the sizes exceeds the total weight. A read of the data consists of reading from enough copies so that the sum of the weights of the copies read constitutes a read quorum. The copy that is retained as the value read is the copy with the most recent timestamp. A write of the data consists of writing to enough copies so that the sum of the weights of the copies written to constitutes a write quorum. In addition, the timestamp of each copy is set to the same time, where the time exceeds the time of the last write.

This algorithm works because every combination of copies of the data that constitutes a read quorum overlaps every combination of copies of the data that

constitutes a write quorum. This fact, coupled with the use of the timestamp to identify the most recently written copies, means that every read quorum is guaranteed to have at least one of the most recently written copies of the data. The algorithm has the following advantages over non-replicated schemes and simple replication schemes (e.g. read-one/write-all):

- Quorum sizes can be chosen so that neither quorum requires the total weight of the system. Reads and writes can therefore still be accomplished even though copies of the data are inaccessible, thereby enhancing the availability of the system.

- The weights of a copy can be chosen to reflect the availability or responsiveness of the computer at which the copy resides. For example, the availability of a system composed of moderately reliable and highly reliable computers can be enhanced by assigning larger weights to the copies of the data that reside on the highly reliable computers. Similarly, the responsiveness of a system can be improved by assigning larger weights to the copies of the data that reside on the faster computers. In addition, temporary or cached copies of the data, or copies of the data on computers that are unreliable, can be assigned a weight of zero.

- In the worst case, where N copies each have the same weight, N+1 inter-processor messages are required to execute the algorithm for a read followed by a write. This is the same number of messages required for the "simpler" and less powerful read-one/write-all scheme.

The disadvantages of the algorithm are:

- The inequality of the respective quorum sizes forces a choice to be made that favors reads over writes, or writes over reads. Though data is often either read-intensive or write-intensive, the restrictions caused by the quorum sizes can be undesirable in situations where the frequency of reads and writes are similar.

- In environments (such as Argus) where each copy of the data is a lockable entity, concurrent reads or writes can easily lead to deadlock. This happens when concurrent reads or writes, while attempting to gather a quorum, obtain locks on some of the copies of the data such

that no operation can obtain a quorum.

4.2.2.2 Data Structures

Gifford's algorithm is used by the document registry guardian to coordinate the replication of the registry at the subregistry level. Therefore, the algorithm is used whenever a subregistry is read from or written to. Support for the algorithm is provided by two stable objects in each document registry guardian replicand. One object is of type reg_status and is used to maintain replicand status information including the number of subregistries in the registry, the voting weight of the replicand, the sizes of the read and write quorums, and the total weight of all of the replicands. The other object is of type reg_list and maintains the names of each replicand in the set of replicands. The specifications of these data structures appears in Figure 4-2.

```
reg_status = atomic_record[num_sub_regs: int,
                           own_weight:   int,
                           total_weight: int,
                           read_quorum:  int,
                           write_quorum: int]

reg_list = atomic_array[document_registry]
```

Figure 4-2: Specification of Reg_status and Reg_list Types

4.2.2.3 Creating Guardian Replicands

During the initial configuration of a CES installation, one instance of the document registry guardian is created. This guardian is the seed guardian for any eventual set of guardian replicands. The site at which the seed guardian is

```
all_docs = handler (match_doc: string,
                    match_site: variant[site: network_address,
                                        wild_card: null],
                    match_author: string)
           returns (array[string])
           signals (no_quorum, no_match)
    %
    % Function:
    % Returns the list of document names that satisfy the
    % following constraints:
    %      1.  The match_doc argument is a substring of
    %          the document's name.
    %      2.  The match_site argument, if tagged as
    %          site, is the address of the site of at
    %          least one of the document's authors.
    %      3.  The match_author argument is a substring
    %          of the name of at least one of the
    %          authors that satisfy condition #2.
    %
    % The match_doc, match_site, and match_author arguments are
    % used as filters for which documents to include in the
    % list.  The match_doc and match_author arguments can be
    % the null string, in which case they are ignored, or it
    % can be a string with any combination of alphanumeric
    % characters and the wildcard string "*".  The match_site
    % is either a valid network address (tagged as site) or a
    % wild card (tagged as wild_card).
    %
    % For example, match_doc = "", match_site = oa, and
    % match_author = "*ll*" will return the list of all documents
    % with authors at computer oa, where the names of these
    % documents includes the substring "ll".  If both the
    % match_doc and match_name arguments are the null string, and
    % the match_site argument is tagged as wild_card, then the
    % list returned will include the name of every document in the
    % registry.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry.  Signals no_match if could
    % not find a match for the arguments specified.
    %
```

Figure 4-3: Informal Specification of Document Registry Handlers

Figure 4-3: (continued)

```
locate_doc = handler (doc_name, author_name: string,
                      author_site: network_address)
             returns (struct_id, privilege, authors)
             signals (no_quorum, not_found)
    %
    % Function:
    % Given a document name, scans through the document registry
    % to find the entry for the document.  If the entry is found,
    % then the list of the document's authors is examined to see
    % if the querying author is on the list.  If this is the case,
    % then the handler returns the privilege specified on the list
    % for the author.  If the querying author is not on the list,
    % then the handler returns the document's public access
    % privilege.  In either case, the structure identifier for the
    % copy of the document that resides in the structure catalog
    % physically closest to the author's computer is returned,
    % along with the list of the document's authors.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry.  Signals not_found if an
    % entry for the document named is not found in the registry.
    %

all_struct_ids = handler (doc_name: string)
                 returns (array[struct_id])
                 signals (no_quorum, not_found)
    %
    % Returns the list of all of the structure identifiers
    % associated with the document named.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry.  Signals not_found if an
    % entry for the document named is not found in the registry.
    %
```

Figure 4-3: (continued)

```
add_doc = handler (doc_name, creator_name: string,
                   creator_site: network_address,
                   public_priv: privilege, sid: struct_id)
          signals (no_quorum, already_exists)
    %
    % Function:
    % Adds to the registry a document with the name, creator's
    % name, creator's network address, and public access
    % privilege specified, and structure identifier for the
    % copy of the document's structure at the creators site.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry or a write quorum in order to
    % update the registry.  Signals already_exists if an entry
    % for the document named is found in the registry.
    %

del_doc = handler (doc_name: string, author: author_info)
          signals (no_quorum, not_found, invalid_priv)
    %
    % Function:
    % Deletes the document specified from the registry.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry or a write quorum in order to
    % update the registry.  Signals not_found if an entry for
    % the document named is not found in the registry.  Signals
    % invalid_priv if the author does not have edit access
    % to the document named.
    %
```

Figure 4-3: (continued)

```
add_author = handler (doc_name: string,
                      adder, addee: author_info,
                      sid: struct_id)
                signals (no_quorum, not_found, invalid_priv,
                         already_exists)
    %
    % Function:
    % For the document specified, adds information about the
    % author specified as the addee to list of authors for the
    % document.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry or a write quorum in order to
    % update the registry.  Signals not_found if an entry for
    % the document named is not found in the registry.  Signals
    % invalid_priv if the author does not have edit access
    % to the document named.  Signals already_exists if
    % the addee is already on the document's list of authors.
    %

del_author = handler (doc_name: string,
                      deler, delee: author_info)
                signals (no_quorum, not_found, invalid_priv,
                         is_creator)
    %
    % Function:
    % For the document specified, deletes information about the
    % author specified as the delee from list of authors for the
    % document.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry or a write quorum in order to
    % update the registry.  Signals not_found if an entry for
    % the document named is not found in the registry.  Signals
    % invalid_priv if the author does not have edit access
    % to the document named.  Signals is_creator if the delee
    % is the creator of the document.
    %
```

Figure 4-3: (continued)

```
mod_author_priv = handler (doc_name: string,
                           moder, modee: author_info)
                  signals (no_quorum, not_found, invalid_priv,
                           is_creator)
    %
    % Function:
    % For the document specified, changes the access privilege for
    % the author specified as the modee.  The new access privilege
    % is specified in the modee info.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry or a write quorum in order to
    % update the registry.  Signals not_found if an entry for
    % the document named is not found in the registry.  Signals
    % invalid_priv if the author does not have edit access
    % to the document named.  Signals is_creator if the modee
    % is the creator of the document.
    %
    %

mod_public_priv = handler (doc_name: string, moder: author_info,
                           public_priv: privilege)
                  signals (no_quorum, not_found, invalid_priv)
    %
    % Function:
    % For the document specified, changes the public access
    % privilege as specified.
    %
    % Exceptions:
    % Signals no_quorum if could not obtain a read quorum in
    % order to read the registry or a write quorum in order to
    % update the registry.  Signals not_found if an entry for
    % the document named is not found in the registry.  Signals
    % invalid_priv if the author does not have edit access
    % to the document named.
    %
```

to reside, its weight for the voting algorithm, and the number of subregistries (See Section 4.2.1.1) can all be specified. Using the *replicate_reg* handler of the seed guardian, replicands can be created. Similarly, the replicate_reg handler of any existing replicand can be used to create another replicand. The site at which the replicand is to reside and its weight can be specified.

Guardian replicand creation is a three-step process. During the first step, the new guardian is created. During the second step, the replicand's registry is initialized from copies of the most recently written subregistries. During the third step, the new guardian is added to each replicand's list of replicands (i.e. appended to each guardian's reg_list object), and the quorum sizes are recomputed to include the weight of the new guardian.

Though activities internal to steps two and three can be executed concurrently (e.g. during step three, replicand reg_list objects are simultaneously updated), the order in which the steps are performed is critical. In order to ensure the correct behavior of Gifford's algorithm after a replicand is created, the replicand cannot be allowed to participate as a member of a quorum, and the quorum sizes cannot be adjusted, until after its registry is initialized from the most recent copies of the subregistries. If this is not the case, then all possible pre-replicand write quorums are not guaranteed to overlap all post-replicand read quorums. Unless this situation eventually corrects itself[9], subsequent rounds of Gifford's algorithm might give erroneous results.

[9]The situation is self-correcting if the new replicand participates only as a member of write quorums until all of its subregistries become copies of the most recently written subregistries. Nevertheless, there is a finite probability that the situation will never correct itself. This probability is a function of the quorum sizes and of the number of replicands and their associated weights.

4.2.2.4 Obtaining a Quorum

Since the document registry is read intensive (See Section 2.5.1), the quorum sizes are computed so that for N copies of the registry, the read quorum is approximately N/3 and the write quorum is approximately 2N/3. The approximation is poor for small N, but improves as N increases, as shown in Figure 4-4.

N	W	R
1	1	1
2	2	1
3	2	1
4	3	2
5	4	2
6	4	3
7	5	3
8	6	3
9	6	4

N = number of copies, W = write quorum size, R = read quorum size

Figure 4-4: Chart of Quorum Sizes in Document Registry Guardian

The document registry implementation of Gifford's algorithm exploits the concurrency inherent in the algorithm. A subregistry is read by a guardian by simultaneously broadcasting read request handler calls to all of the guardian replicands, including the broadcasting guardian. Using the Argus action termination semantics described in Section 3.2.2.2, the broadcasting guardian tallies the weights of the replicands that have responded until a read quorum is obtained, at which time the outstanding handler calls are aborted. The copies of the subregistry returned by the replicands that did respond are scanned until the

copy with the most recent timestamp is found. This is the copy that is "read".

A write to a subregistry is analogous to a read. The difference is that the writer increments the subregistry timestamp[10] prior to broadcasting write request handler calls. This implies that there should be no blind writes to the subregistries, and this is the case. The broadcasting guardian tallies the weights of the responding replicands until a write quorum is obtained, at which time the outstanding handler calls are aborted.

4.2.2.5 Avoiding Deadlock

The potential of Gifford's algorithm for causing deadlock was mentioned in Section 4.2.2.1; also, Argus does not perform deadlock prevention or detection [Liskov & Scheifler 83]. Consequently, the document registry guardian implementation of Gifford's algorithm employs a deadlock avoidance scheme. Upon the receipt of a subregistry read request, a document registry guardian tests the state of the specified subregistry's read lock. An attempt is made every one-hundred milliseconds, for a maximum of five seconds, to acquire the read lock. The lock testing and timer are executed as concurrent sibling actions, and whichever completes first aborts the other. If the timer action completes first, then the read request terminates with an "avoiding possible read-write deadlock" exception; otherwise the subregistry's read lock is obtained and the subregistry is read. A write request is processed in the same fashion, except that the lock test attempts to acquire the specified subregistry's write lock.

Although this deadlock avoidance scheme is simple, it is not the best solution. This is for two reasons:

---

[10]The timestamp is implemented as a monotonically increasing number with "infinite" precision - such a number contains enough digits so that for the life of the system, it can increase without bounds. For example, a 32 bit binary number can be incremented once per second for approximately 2,000 years before it will overflow.

- There is no way to know, a priori, how long to wait before timing out. In Argus, locks are held for the duration of the topaction within which the lock was acquired. Unless some high-level clock forces topactions that have run for "too long" to abort, there is no way to be certain how long to wait for a topaction to terminate. Similarly, Argus lock queries take an unspecified amount of time, so it is not clear how long to wait for lock queries to propagate.

- Even if a deadlock is avoided, there is no guarantee about how, or if, progress will be made. If the action that initiated a subregistry read or write results in a deadlock prevention exception, there is no guarantee that retries will not also result in the same exception.

The scheme is nevertheless used in the document registry guardian because it can be simply and efficiently implemented. Deadlock detection is performed locally within a guardian, and uses Argus primitives to do most of the work. Complex algorithms and explicit inter-guardian communication are not required.

4.3 Document Library

The document library guardian implements the document library subsystem. Each instance of the document library guardian maintains a subset of the installation-wide structure catalog. A guardian's subset contains all of the document structures for documents which have authors at the guardian's site. Each instance of the guardian also maintains a mini-catalog of all of the document nodes which have been created by authors at the guardian's site. The node mini-catalogs are unique, but structure catalog subsets can overlap because the structures for documents with geographically dispersed authors are replicated at each author's site.

The replication of portions of the structure catalog enhances the resiliency of the catalog, though contrary to the document library subsystem design, the structure catalog is not highly available, but gracefully degrades. The graceful degradation

of the structure catalog manifests as the inability to update a document structure which is replicated at a failed site, even though it can still be read. The replication of portions of the structure catalog does improve its responsiveness, though. This is because an author's references to a document's structure will be local to the author's computer, which is presumably faster than inter-computer communication.

As discussed in the document library design specification, the uniqueness of node mini-catalogs preserves the locality of ownership for author node-level contributions. This means, though, that node mini-catalogs will not be resilient to failures. The failure of a document library guardian will therefore preclude access to all of the nodes maintained in its node mini-catalog. In order to mitigate the effects of the failure of a document library guardian, the overall functionality of the node catalog gracefully degrades. The graceful degradation of the node catalog manifests as the inability to access the document nodes at the failed site. With regard to responsiveness, the node mini-catalogs accommodate local references.

Following the functional distinction between the structure and node catalog, the document library guardian is implemented as two distinct pieces: the structure catalog and the node mini-catalog (hereafter referred to as the "node catalog"). The structure catalog is contained in a stable object of type *struct_catalog*. The node catalog is contained in a stable object of type *node_catalog*. Two essentially independent sets of handlers support the respective catalogs.

4.3.1 Document Node Catalog

4.3.1.1 Basic Data Structures

The node_catalog type is designed to support a high degree of read-write concurrency with respect to individual nodes and the catalog as a whole. The catalog design allows concurrent reads or writes in different nodes by independent actions, but the independent actions cannot concurrently add or delete nodes. The node_catalog type also implements the highly concurrent node version stack and exclusive edit access timer, both of which were described in Section 2.5.2.

The specification of the node_catalog and associated types is shown in Figure 4-5. The node_catalog type structure is a non-atomic array encapsulated in a mutex. The array elements are of type *node_cat_entry* which is a two field non-atomic record. One field of the record is of type *status* which is an atomic variant with two states - "present" and "empty." The null type is associated with both of these states. The other field of the record is of type *node_info* which is a mutex encapsulated non-atomic multi-field record of node specific information. Non-atomic structures and mutexes are used to provide a high degree of concurrency and to minimize the size of the object that must be written to stable storage when node versions are added, deleted, or updated (See Section 3.2.3.2).

In an instance of the document library guardian, each element in the node_catalog array is uniquely identified by its array index. This index is coupled with the network address of the guardian in an object of the cluster type *node_id* with the network address of the guardian to provide a CES installation-wide unique document node identifier. For example, the node with an index of "7" in the catalog of the guardian at address "oa" is specified as "7@oa". Developing the identifier in this manner has the advantage that the index portion of the identifier is used by the document library guardian to rapidly access the associated node. This direct mapping technique enhances the responsiveness of the guardian.

```
status = atomic_variant[present: null,
                        absent:  null]

node_info = mutex[record[
                        nid:       node_id,
                        last_time: very_big_num,
                        last_ed:   ed_front_end,
                        versions:  astack[node_version_info]
                        ]
                ]

node_cat_entry = record[status:    status,
                        node_info: node_info]

node_catalog = mutex[array[node_cat_entry]]
```

Figure 4-5: Specification of Node_catalog and Related Types

Node entries are added to the catalog by seizing the node_catalog mutex and appending the new entry. In order to make this operation recoverable, the status atomic variant is used. When a node_cat_entry object is created, its status is created as "empty." The status is then set to "present" and the node_cat_entry object is appended to the catalog. If the action executing the add-entry operation aborts, the entry remains as part of the node_catalog array, but is tagged as "empty". This entry is ignored by future guardian operations until it is eventually recycled (See Section 4.3.1.5). If the action commits, then the entry is "present" and available for use by other guardian operations.

The deletion of node entries compliments the addition process. An entry is deleted by setting its status to "empty". If the action executing the delete-node operation aborts, the status of the entry is recovered as "present". In addition to changing an entry's tag, the delete-node operation adds the entry's index to an atomic list of deleted node entry indices. This list is used to expedite the

recycling of deleted entries by allowing the add-node operation to simply recycle the entry at the head of the list. The advantage of using such a list is that it is a lot faster than scanning the entire node catalog until an "empty" entry is found. The disadvantage is that its atomic nature prevents actions from concurrently executing add-node and/or delete-node operations. In CES, the frequency of add-node or delete-node operations is small relative to other node catalog operations, so this disadvantage is not significant.

4.3.1.2 Version Stack

Node-specific information is contained in the node_info record associated with each node entry. The key piece of information maintained in this record is the node's version stack. Each node version on the stack is numbered, starting from zero, and includes the document node, login name, and network address of the author who last modified the version, and the row and column of the keyboard cursor when the modification was written to the node catalog. For all versions but the base version, the node version information also includes the login name, network address, and keyboard cursor row and column of the previous version. Consequently, when any single version is read, information about the previous version is also obtained.

The node version stack is essentially an atomic stack where stack items are of the cluster type *node_version_info*. The implementation of the stack is a cluster type called *astack* which provides such common stack operations as *push*, *pop*, and *top* [11] operations. In addition, three uncommon operations are provided which support the read-while-writing concurrency described in Section 2.5.2. These operations are:

- Reset — The reset operation removes all of the existing

[11] The astack top operation reads the top item on the astack without altering the astack.

items on the astack, then pushes the item specified as the argument to the operation onto the astack. The astack is *reset* with the argument as its sole item. This operation is an alternative to creating a new astack object.

- Newtop — The newtop operation overwrites the top item of an astack with the item specified as the argument to the operation. This operation is an alternative to first popping the current top item and then pushing a new item.

- Fasttop — The fasttop operation returns the value of an item that is either the top-most item on the stack, or one near the top. Actions using this operation can execute concurrently with an action that is altering the state of the target astack. Consecutive calls of fasttop without intervening calls of the push, pop, or reset operations will return the same astack item or an item closer to the top.

The representation of the node_version_info cluster is simply a non-atomic record. The representation of the astack is more complicated. The representation of the astack is shown in Figure 4-6, its informal specification is shown in Figure 4-7. The astack is composed of a mutex encapsulated non-atomic record. One field of this record is an atomic type (arbitrarily chosen to be an atomic record) that serves as an atomic lock on the astack. This lock ensures serialization of all actions on an astack object, except for actions executing the fasttop operation. The other field of this record is a non-atomic array with an atomic variant element type.

The atomic variant is used to provide recoverability much like the present/empty tag described in Section 4.3.1.1 for node catalog entries. The difference is that the object type associated with the "empty" state of the variant is null, while the object type associated with the "present" state is the astack element type (which, in this case, is the cluster type node_version_info). Therefore, the value of the

stack item is bound to an atomic variant. The reset, push, and pop operations all manipulate the state of astack atomic variants to "add" or "remove" astack items. In addition, rather than recycling "empty" items, a nested topaction is run as part of each reset, push, and pop operation to garbage collect "empty" items. This garbage collection is more efficient than recycling (See Section 4.2.1) because most of the "empty" items in an astack are created by the pop and reset operations. These items are at the physical top of the astack, so compression of the astack is simply accomplished by removing the top-most item or items.

The granularity of astack updates to stable storage is minimized for each astack operation. Top and fasttop do not alter the state of an astack, so stable storage updates are not required for these operations. In contrast, reset, push, pop, and newtop all alter the state of an astack object. In addition, all of these operations invoke the astack garbage collector, which potentially alters the state of an astack object. The reset operation causes the stable storage copy of the entire astack object to be updated because this operation alters every item on the stack. For the other operations, unless an item is actually garbage collected, only the stable storage copy of the single item that is the target of a push, pop, or newtop operation is updated. If an item is garbage collected, then the stable storage copies of the single target item and of the astack array object are updated. This is less costly than the reset operation, but more costly than if an item was not garbage collected. In all cases, though, only the stable storage copies of those astack component objects which have been altered are updated.

Actions including the fasttop operations can run concurrently with other actions that alter the same astack. The fasttop operation runs as a nested topaction of the calling action. The operation does not first acquire the astack lock. Instead, serializability is enforced by the locks on the atomic variants that encapsulate each astack item. Starting from the physical top of the astack object, the fasttop operation scans down through the astack, testing the read lock on each astack

atomic variant. The lock test attempts to acquire a read lock if it can be obtained without waiting. If a read lock is acquired, the state of the atomic variant is examined. If the state is "present", then the item encapsulated by the atomic variant is returned as the item read. Except for the duration of the operation (during which the astack mutex is seized), the fasttop operation is non-blocking because the nested topaction within which it runs commits at the completion of the operation, thereby releasing any read locks that were obtained.

4.3.1.3 Tickle-Lock

Exclusive edit access to a node's version stack is implemented by an explicit lock. This lock is called a *tickle-lock*. The possessor of the lock can reassert possession of it anytime before the access interval expires by resetting, or "tickling", the lock interval timer.

The tickle-lock consists of a component that is local to each node version stack, and a shared, monotonically increasing timer of "infinite" precision. The local component of the lock is maintained in the *last_time* and *last_editor* fields of a node entry's node_info record. The last_time field is the time at which the node version stack was last tickle-locked, and the last_editor field is the name of the editor front-end guardian (which is uniquely associated with a single CES user) which obtained the tickle-lock. The timer component of the tickle-lock is maintained in a document library guardian stable object. The tickle-lock semantics are as follows:

- If the current time exceeds the time marked in last_time by more than the tickle-lock timeout interval, then the tickle lock is obtained. Last_time is set to the current time, and last_editor is set to the name of the guardian that obtained the lock.

- If the current time does not exceed the time marked in last_time by more than the tickle-lock timeout interval, then the lock cannot be obtained.

The tickle lock is implemented in non-atomic portions of the node_cat_entry structure, so read and write locks do not interfere with the testing and setting of the tickle-lock. The tickle-lock behavior is nevertheless atomic, in a sense. In particular, if an action in which a guardian acquires a tickle-lock subsequently aborts, then the state of the associated node entry recovers after the timeout interval to a state that leaves the tickle-lock unlocked. This recovery is only partial because the guardian which lost the tickle-lock is not reinstated as the holder of the lock and another guardian can "sneak" in and obtain the lock. Therefore the system is not restored to the same state that it was in prior to the aborted action. In terms of the CES application demands, this partial recovery is acceptable because the user who lost possession of the tickle-lock can simply request that it be reacquired.

Another notable aspect of the tickle-lock is that its real-time nature is not explicitly supported by Argus. In fact, no guarantees can be made by the tickle-lock implementation to ensure that the time that it takes to acquire a tickle-lock is less than the tickle-lock timeout interval. If the time to acquire a tickle-lock is more than an order of magnitude less than the tickle-lock timeout interval, then the purpose of the tickle-lock is defeated. Even excluding failures, Argus implementations are not forced by the language's specification to satisfy any timeliness constraints. The only way in which an Argus programmer can truly impose real-time control on Argus programs is to use operating system provided timers to abort actions that have run for "too long." Forcing actions to run to completion and commit within a certain time is not possible. Nevertheless, the current Argus implementation is fairly efficient, and tests of the tickle-lock have not yet revealed the problem alluded to.

```
lock = atomic_record[x: null]

item = atomic_variant[present: item_type,
                      absent:  null]

astack = mutex[record[
                      lock: lock,
                      items: array[item]
                      ]
               ]
```

Figure 4-6: Specification of Astack Representation

4.3.1.4 Handlers and Action Structures

The node library provides a set of handlers that operate on the node version stacks. The informal specification of these handlers is shown Figure 4-8. The handlers are primarily abstractions of the astack operations discussed in Section 4.3.1.2. Except for *getr_node*, the handlers are all classified as writers and enforce tickle-lock and access privilege semantics (See Section 2.5.1). The getr_node handler only enforces access privilege semantics and avoids the tickle lock, so that a version of a node can be concurrently read from and written to.

The action structure of the node catalog handlers had to be carefully designed to avoid deadlock and starvation[12] . The problems to avoid are:

- The suggested style of using Argus mutex objects is that actions should not wait to acquire locks within the seize of a mutex, and pausing within nested mutex seizes should be avoided [Liskov *et al.* 83]. Not heeding this advice can result in programs that deadlock.
    - An action that waits inside of a mutex seize for a lock to be released can prevent the action that holds the lock from releasing it if the holding action needs to first seize the mutex.

[12]Starvation occurs when a part of a system is consistently prevented from making progress.

```
create = proc () returns (astack)
    %
    % Function:
    % Creates a new, empty astack object.
    %

createi = proc (item_value: element_type) returns (astack)
    %
    % Function:
    % Creates a new astack object, initialized with item_value
    % as the value of its first item.
    %

reset = proc (A: astack, item_value: item_type)
    %
    % Function:
    % For A = {I1, I2, .. In}, resets A so that A = {item_value}.
    %

push = proc (A: astack, item_value: item_type)
    %
    % Function:
    % For A = {I1, I2, .. In}, appends item_value to A so that
    % A = {I1, I2, .. In, item_value}.
    %

pop = proc (A: astack) returns (item_type) signals (empty)
    %
    % Function:
    % For A = {I1, I2, .. In}, removes In, the top-most item
    % on A, from A so that A = {I1, I2, .. In-1}.  Returns the
    % value In.
    %
    % Exceptions:
    % Signals empty if there are no items on A.
    %
```

Figure 4-7: Informal Specification of Astack Cluster

Figure 4-7: (continued)

```
top = proc (A: astack) returns (item_type) signals (empty)
    %
    % Function:
    % For A = {I1, I2, .. In}, reads the value of top-most item,
    % In, on A.  Returns the value of In.  A is not changed.
    %
    % Exceptions:
    % Signals empty if there are no items on A.
    %

newtop = proc (A: astack, item_value: item_type) signals (empty)
    %
    % Function:
    % For A = {I1, I2, .. In}, sets the value of top-most item,
    % In, on A to item_value so that A = {I1, I2, .. In-1,
    % item_value}.
    %
    % Exceptions:
    % Signals empty if there are no items on A.
    %

fasttop = proc (A: astack) returns (item_type) signals (empty)
    %
    % Function:
    % For A = {I1, I2, .. In}, reads the value of an item,
    % Ij, that is near or the same as the top-most item, In,
    % on A.  Returns the value of Ij.
    %
    % Consecutive calls of this operation without intervening
    % calls of the push, pop, or reset operations, will return
    % Ii such that j <= i <= n.
    %
    % Exceptions:
    % Signals empty if cannot find an item to read even
    % though there may in fact be items on A.
    %
```

```
add_node = handler (node: doc_node, author: author_info,
                    editor: editor_front_end)
           returns (node_id)
    %
    % Function:
    % Adds the node specified to the node catalog by creating an
    % entry and version stack for it.  The name and network
    % address of the author specified is distinguished in the
    % entry as the node's creator.  The version stack is
    % initialized with the node specified; the version number
    % for this base version is set to 0.  Returns the unique
    % identifier for the new node.  The node's tickle-lock is
    % implicitly obtained.
    %

del_node = handler (nid: node_id, author: author_info,
                    editor: editor_front_end)
           returns (node_version_info)
           signals (invalid_priv, not_found, in_use)
    %
    % Function:
    % If can immediately obtain the tickle-lock for the node
    % specified, then will remove the entry for the node from
    % the node catalog.  Returns version of the node at the top
    % of its version stack prior to removal.
    %
    % Exceptions:
    % If author's privilege is not edit access, then signals
    % invalid_priv.  Signals not_found if the node specified is
    % not found in the catalog.  If node is being used by another
    % author (i.e. its tickle-lock is held), then signals
    % in_use.
    %
```

Figure 4-8: Informal Specification of Node Catalog Handlers

Figure 4-8: (continued)

```
getr_node = handler (nid: node_id, author: author_info,
                     editor: editor_front_end)
            returns (node_version_info)
            signals (invalid_priv, not_found)
    %
    % Function:
    % Returns the most recent, or close to the most recent,
    % version on the version stack of the node specified.
    % Consecutive calls will return the same version, or a
    % more recent version, providing that version stack is
    % not altered between the calls.  The middle-lock for the
    % version stack is not tested, nor is it acquired.
    %
    % Exceptions:
    % If author's privilege is not edit or view-only access, then
    % signals invalid_priv.  Signals not_found if the node
    % named is not found in the catalog.
    %

getw_node = handler (nid: node_id, author: author_info,
                     editor: editor_front_end)
            returns (node_version_info)
            signals (invalid_priv, not_found)
    %
    % Function:
    % Returns the most recent version on the version stack of
    % the node specified.  Must first obtain middle-lock on
    % node specified; if held by another author, will wait
    % until it expires.
    %
    % Exceptions:
    % If author's privilege is not edit access, then signals
    % invalid_priv.  Signals not_found if the node specified
    % is not found in the catalog.
    %
```

Figure 4-8: (continued)

```
new_node_version = handler (nid: node_id, dnode: doc_node,
                            author: author_info,
                            row, col: int,
                            editor: editor_front_end)
                   signals (invalid_priv, not_found, in_use)
    %
    % Function:
    % If can immediately obtain the tickle-lock for the node
    % specified, then will push a new version of the node
    % onto the node's version stack.  The version info for the new
    % version is created from information supplied by the caller
    % as well as from information taken from the previous version.
    % Thus, the version information for the current version
    % includes version information about the previous version.
    % The current version number is not to one greater than the
    % previous version number.
    %
    % Exceptions:
    % If author's privilege is not edit access, then signals
    % invalid_priv.  Signals not_found if the node specified is
    % not found in the catalog.  If node is being used by another
    % author (i.e. its tickle-lock is held), then signals
    % in_use.
    %

update_node_version = handler (nid: node_id, dnode: doc_node,
                               author: author_info,
                               row, col: int,
                               editor: editor_front_end)
                      signals (invalid_priv, not_found, in_use)
    %
    % Function:
    % If can immediately obtain the tickle-lock for the node
    % specified, then will update the most recent version on the
    % node's version stack.  Only the last row and last column
    % fields, and the node itself, are actually changed; the rest
    % of the node version info is left intact.
    %
    % Exceptions:
    % If author's privilege is not edit access, then signals
    % invalid_priv.  Signals not_found if the node specified is
    % not found in the catalog.  If node is being used by another
    % author (i.e. its tickle-lock is held), then signals
    % in_use.
    %
```

Figure 4-8: (continued)

```
del_node_versions = handler (nid: node_id, author: author_info,
                             num_to_del: int,
                             editor: editor_front_end)
                    returns (node_version_info)
                    signals (invalid_priv, not_found, in_use,
                             no_change)
    %
    % Function:
    % If can immediately obtain the tickle-lock for the node
    % specified, then will remove the specified number of
    % versions from the node's version stack.  Returns the new
    % most recent version on the version stack.
    %
    % Exceptions:
    % If author's privilege is not edit access, then signals
    % invalid_priv.  Signals not_found if the node specified is
    % not found in the catalog.  If node is being used by another
    % editor (i.e. its tickle-lock is held), then signals
    % in_use.  If the number of versions to delete exceeds the
    % number of versions on the stack, or if it would cause the
    % oldest version to be deleted, or if it less than 1, then
    % signals no_change and the version stack is not affected.
    %

set_node = handler (did: doc_id, nid: node_id,
                    author: author_info,
                    editor: editor_front_end)
           signals (invalid_priv, not_found, in_use, no_change)
    %
    % Function:
    % If can immediately obtain the tickle-lock for the node
    % specified, then will reset the version stack of the node
    % by setting the most recent version on the stack as the
    % stack's base version.  The version number for the new base
    % version is set to 0.  All other versions are purged.
    %
    % Exceptions:
    % If author's privilege is not edit access, then signals
    % invalid_priv.  Signals not_found if the node specified is
    % not found in the catalog.  If node is being used by another
    % editor (i.e. its tickle-lock is held), then signals
    % in_use.  If the version of node on the version stack is
    % the base version (i.e. version number = 0), then signals
    % no_change and version stack is not affected.
    %
```

Figure 4-8: (continued)

```
lookup_node = handler (nid: node_id) returns (bool)
    %
    % Function:
    % Returns true if node specified is in the catalog, false
    % otherwise.  Does not obtain a tickle lock on the node.
    %

catalogue_nodes = handler () returns (array[node_id])
    %
    % Function:
    % Maintenance handler used to report node ids of all nodes in
    % the catalog; good for determining how large the catalog is,
    % etc.  Tickle-locks are not obtained.
    %
```

  - The execution of a nested pause only releases possession of the nested mutex so that another action is still prevented from seizing the outer mutexes. This is analogous to the "nested monitor" problem [Lister 77].

- Guardian G1, whose use of a tickle-lock has expired, may still have actions running which hold locks on the atomic objects that comprise the node version stack. It is possible for another guardian, G2, to obtain the expired tickle-lock but become blocked by locks still held on the version stack. Eventually these locks might be released, but the tickle-lock interval for G2 may have elapsed. Before G2 can reassert possession of the tickle-lock, guardian G3 "sneaks" in and obtains possession. Eventually, G2 reacquires possession of the tickle-lock, and as was previously the case, some other guardian's actions are still holding node version stack locks. The result is that G2's progress is delayed, or it can become effectively starved.

To address the nested pause problem in the node catalog in general, mutex objects are treated as immutable. By permanently preserving the binding between a mutex object and the object that it encapsulates, nested mutexes can be seized and released in a sequential, rather than nested, fashion. The semantics

of the nested seize are preserved by the fixed order in which the mutexes are seized. For example, in the node catalog, the bindings between each of these objects and their mutex encapsulations are permanent. This is supported by the recycling of node entries and the version stack reset operation. Therefore, in order to read an item on a particular node's version stack, the node catalog is seized to obtain the node entry and then released, the node entry is seized to obtain the node version stack and then released, and then the node version stack is seized to obtain the desired node version.

In order to address the starvation problem, a mechanism is employed that forces a guardian whose possession of a tickle-lock has expired to release any node catalog locks held. This mechanism requires that the editor front-end guardian provide a *force_abort* handler. An editor front-end guardian responds to this handler call by immediately aborting any of its actions that hold node catalog locks. This call is made to a guardian when the guardian's possession of a tickle-lock is lost to another guardian (as opposed to merely expiring). It is for this reason that the name of the guardian which last held the tickle-lock on a node is recorded in the last_editor field of the node_info record. As will be discussed in Section 4.4.2, the editor front-end minimizes the amount of work lost due to such aborts by performing activities as a series of short duration actions.

4.3.1.5 Background and Recovery

A background process periodically increments the tickle-lock interval timer. Both the duration of the tickle-lock interval and the timer increment interval are specified by the creator of the guardian. The background process sleeps in the interval between timer increments.

After a crash, recovery consists of the implicit restoration from stable objects of the tickle-lock timer and node catalog portions of the document library

guardian's volatile state. In addition, the list of indices for "empty" node entries is recreated. The list is compiled by sequentially scanning through every node entry in the catalog and examining its state. Though this process is tedious, it is only performed during recovery before the guardian is on-line, and can therefore be tolerated.

4.3.2 Document Structure Catalog

The document structure catalog data structures are virtually identical to the document node catalog's. Similarly, the structure catalog version stack and tickle-lock semantics are the same as described for the node catalog, and the structure catalog handlers compliment the node catalog handlers. The difference between the two catalogs is the support for the replication of the structure catalog (See Section 4.3). In particular, the structure catalog provides a distributed tickle-lock algorithm and read-one-write-all semantics to coordinate multiple copies of individual document structures.

4.3.2.1 Distributed Tickle-Lock

To coordinate multiple copies of document structures, Gifford's algorithm (See Section 4.2.2.1) was considered, but attempting to integrate tickle-locks with the algorithm created major complications. The tickle-lock relies upon a real-time time base. This means that any quorum must be able to decide whether or not a tickle-lock has expired. The only way to do this is to ensure that the timers of all document library guardians in a CES installation are synchronized. Though this can be done [Lamport 78], the additional complexity is considerable. An algorithm that allows each guardian to independently determine the status of a document structure's tickle-lock is used instead. This scheme is simpler, but less powerful because it necessitates read-one-write-all semantics for structure catalog updates. The structure catalog design specifications of high availability and

responsiveness are consequently compromised.

The distributed tickle-lock algorithm is implemented so that the testing and acquisition of the tickle-lock associated with each copy of a document structure occur in parallel. An attempt to obtain a tickle-lock is targeted at a single document library guardian which in turn broadcasts the request to all of the document library guardians at which a copy of the specified document structure resides. Each of these guardians participates in the distributed tickle-lock algorithm by locally testing and acquiring, as appropriate, the tickle-lock associated with the local copy of the document's structure. The requester of a tickle-lock obtains it only when all of the local copies have been obtained.

A major drawback to this algorithm is that the problem of tickle-lock acquisition time exceeding the tickle-lock timeout interval (See Section 4.3.1.3) is accentuated. This is due to the communication delays involved in executing the algorithm and the asynchrony of timer updates. It should be noted that Gifford's algorithm with synchronized timers would also suffer from this problem because Argus does not guarantee the timeliness of its operations.

4.3.2.2 Coordinating Updates

A direct consequence of the distributed tickle-lock algorithm is read-one-write-all semantics for structure catalog updates. This is because all writers must first obtain the distributed tickle-lock on the target structure, thereby involving in the distributed tickle-lock algorithm all of the library guardians at which copies of the structure reside. The failure of any of the participating guardians prevents the update from completing, and until the failed guardian recovers, only reads of the structure will be possible. Therefore, the high availability and responsiveness of the structure catalog is diminished. With respect to a particular structure entry, the structure catalog is only as responsive as the least responsive guardian

at which a copy of the structure resides. Similarly, when such a guardian fails, the functionality of the structure catalog gracefully degrades.

The mechanics of performing updates is similar to the broadcast scheme described for the implementation of Gifford's algorithm (See Section 4.2.2.4). In the structure catalog, though, the scheme for obtaining the list of guardians to broadcast to is more complicated because the list is potentially different for each document structure. Though it is possible to broadcast the update to every document library guardian in a CES installation and let each guardian decide upon the relevancy of the update, such a scheme involves many messages and can unnecessarily mitigate the availability and responsiveness of the structure catalog. This is because each update would require communication with all of the document library guardians in an installation.

A more efficient approach is to restrict the updates to only those guardians for which the update is relevant. This approach is implemented in CES by keeping a list in the system catalog guardian of all of the document library guardians in an installation and narrowing the list based on the list of sites at which a document's authors reside. The latter list is maintained in the document registry. The document library guardian, which is the root of a structure update, queries the system catalog and document registry to produce the guardians for which the update is relevant. Update messages are then sent to just these guardians.

4.4 Editor Front-End

The editor front-end subsystem is implemented by a pair of guardians. The editor front-end guardian maintains the CES user interface. It controls display updates and parses and interprets user inputs. The I/O guardian emulates an atomic terminal, and serves as the interface to an actual VAXstation 100.

Instances of the editor front-end guard and an I/O guardian exist in a master/slave relationship, with the I/O guardian the subordinate component. This structure is not visible to the user. For each CES user session, a pair of these guardians is created. At the end of the session, the guardians are destroyed, thereby releasing computing resources.

The use of a dedicated I/O guardian is a work-around to an obstacle posed by the current Argus implementation. In this implementation, the Argus run-time system is built on top of the Unix operating system. In Unix, each guardian is instantiated as a Unix process, and concurrent actions (i.e. handler calls, background processes, concurrent sibling actions) are actually pseudo-processes. Though this scheme is intended to be transparent to the programmer, it does create the problem that when any guardian pseudo-process executes a blocking Unix system call, the entire guardian is blocked. For example, it is not possible to wait for user input without blocking an entire guardian. This is not acceptable behavior for the editor front-end guardian. Therefore, the I/O guardian is used as a slave to the editor front-end guardian.

4.4.1 I/O Guardian

The display atomicity provided by the I/O guardian allows the user to view the intermediate updates performed by an unterminated action. This "window" into the activities of an action is required in order to supply the user with the tactile feedback necessary to make terminal activities useful. If the user is considered to be part of the update action, then in fact this "window" is entirely appropriate.

The I/O guardian emulates an atomic I/O device by combining atomic data structures with an action structure that actively assists in undoing the effects of aborted display update actions. The active restoration of displays after update actions abort is necessitated by the fact that I/O hardware (e.g. the VAXstation

100) is not atomic. Even though the guardian can use an atomic text buffer to contain an image of the text being displayed, the display memory is not atomic. When an update action aborts, the display memory must be explicitly restored to reflect the contents of the atomic display image. Input is less complicated to deal with, because for user supplied keyboard or mouse input, it is not clear what "atomic" should mean. As long as the effects of the input are undone when the actions based on the input abort, then the input activity will be, in a sense, atomic. Though the additional property of automatically reentering input is perhaps desirable for "atomic" input, this kind of response is not particularly useful in CES, hence it is not provided.

4.4.1.1 Emulating and Atomic Display

An image of the terminal display is maintained by the guardian in a stable atomic object of type *display_image*. This type can be thought of as an atomic record. The *lines* field of the record is an atomic array of strings. The *row* and *column* fields of the record are integers that specify the coordinates of the keyboard cursor. Other information contained in display_image reflects the display status, such as whether the video is black-on-white or white-on-black.

Three other objects are used by the guardian to emulate atomic display behavior. The first of these objects is called *cr* and is of the type *cregion*, which is an atomic type (arbitrarily chosen to be an atomic record). Cr is used to synchronize I/O guardian actions. The second object is called the *action_count* and is a non-atomic monotonically increasing number with "infinite" precision. The third object is called the *commit_count* and is an atomic monotonically increasing number with "infinite" precision. These last two objects are used to detect aborted actions.

The I/O guardian action structure is such that handlers do not execute nested

actions or topactions. Therefore, the mechanism for providing atomic I/O relies on the active, timely detection of aborted I/O guardian handler calls. Aborted handler calls are detected by the execution of an algorithm that compares the values of action_count and commit_count after the topaction within which a handler call ran terminates. When the I/O guardian is created, both of the counts are initialized to the same value. Each I/O guardian handler call then logs itself in by setting the two counts equal to the larger of the two and then incrementing the commit_count. If the topaction within which the handler call ran commits, then commit_count is greater than action_count. If the handler call aborts, then commit_count equals the action_count. Therefore, the action_count records the fact that a handler call occurred, while the commit_count records the commit/abort fate of a handler call. When an aborted handler call is detected, the display is repainted based on the state of the display_image object.

The timely execution of the algorithm is performed by background processing. Handler calls synchronize against each other and against background actions by acquiring the write lock on cr. The handler sets action_count and commit_count as described above, and then awakens a background process that waits on cr's write lock. The handler then executes its handler specific activities. When the handler's ancestral topaction terminates, the lock that the handler held on cr is released and the background process acquires it. The background process then compares action_count and commit_count to determine whether or not the handler call that awoke it committed or aborted. If the handler call committed to the top, then the background process terminates and goes back to sleep to wait for the next handler call. If the handler call aborted, then the background process repaints the display so that it agrees with the state of the display_image object. By holding cr's write lock, new handler calls are prevented from logging in until the background process completes.

A complication arises if a new handler call acquires cr's write lock before the

awakened background process does. Because of this possibility, the scheme described so far will not guarantee that the display state agrees with the state of the display_image object when each handler call runs. This problem is resolved by letting handler calls execute the same algorithm during login as the background process, if the background process has not already done so. This extension involves the introduction of a boolean flag. Whenever the background process runs, it sets the flag. Whenever a handler call logs in and finds that the previous handler call aborted and the flag is false, then it repaints the display because the background process has not already done so. Otherwise, it sets the flag false.

This algorithm is quite effective, but has several potential drawbacks:

- After the background process is awakened, it must busy-wait until it can obtain cr's write lock. This is an inefficient use of computing resources. In Argus, this kind of busy waiting seems to be unavoidable because it is the only way to provide a timely response to an action termination.

- The entire display is repainted after a handler call aborts. Repainting could be made more efficient by using a scheme that logs display changes rather than handler calls so that individual changes could be undone when a handler call aborts.

- The use of the cr object to synchronize handler calls and background code effectively prevents concurrent execution of handler calls within the I/O guardian.

4.4.2 Editor Front-End Guardian

The editor front-end guardian is the root of all CES application-level activities. These activities can be classified as those that process the inputs of the editor front-end guardian's user, and those that provide updates of the document view. The effects of these activities are presented to the user through the three window display described in Section 2.5.3. This display is implemented in the editor

front-end guardian as a virtual display device. The virtual display is mapped onto the single atomic display provided by the I/O guardian.

4.4.2.1 Basic Data Structures

The basic data structure employed by the editor front-end guardian is of the type *environment*. The environment type is essentially an atomic record with a substantial number of fields, and is used by the guardian as a database of state information. The environment database includes items such as the name of the guardian's user and the name of the document being edited or viewed. The environment is also used for caching information about the document being viewed or edited. This includes copies of the document's structure and the nodes that are displayed in the document node and text image windows. The cache is updated when necessary, as for example, when a new node is selected for editing.

4.4.2.2 Action Structure

The editor front-end action structure is implemented as a loop of six sequentially executed topactions. This sequential structure is primarily designed to prevent deadlock between editor front-end actions, but in some cases the sequential semantics is simply appropriate. The loop is executed as a background process and is designed to cycle rapidly and without blocking. This is realized by using short duration actions or by using subaction level timers to abort actions that have executed for too long. Short duration actions are implemented by trapping at the subaction level on key editor front-end events or states and propagating commits back to the ancestral topaction. In the instances where a subaction is waiting for asynchronous events (such as user input), timers are used to abort the subaction and then propagate commits back to the ancestral topaction. In either case, the consistency of all of the guardians' states is preserved and the amount of work lost due to a timeout is minimized.

Within each pass through the six topactions, some maintenance chores are performed, user commands are processed, and the three windows are updated. The six actions, listed in order of execution, are described below:

- Redisplayer — This topaction is triggered by any of the other topactions when communication with the I/O guardian has failed. When triggered, the redisplayer topaction waits until the I/O guardian recovers, then makes sure that the state displayed by the I/O guardians agrees with the state of the virtual display.

- Catalog-Updater — This topaction is triggered by any of the other topactions whenever communication with a document registry or document library guardian has failed. When triggered, the catalog-updater topaction queries the CES system catalog for the names of the nearest document library and document registry guardians, as the reason for the communication failure might have been the reconfiguration of the CES installation.

- Edit-Status-Updater — This action processes any forced aborts (See Section 4.3.1.3) that occurred since the editor front-end guardian was last using a document library. The topaction does not actually force the abort of any editor front-end actions because when the topaction runs, there are no outstanding actions. The edit-status-updater makes sure that the user is informed that possession of the document structure or node for which the forced abort was requested has been lost by displaying a message on the user's interface display.

- Editor — This topaction executes the document editor and updates the document library if the document's outline representation is edited or parsed, or the node to which the user has exclusive edit access has been altered by the user. Structure and node tickle-locks are explicitly acquired during the

topaction, so the topaction also monitors for forced aborts. If the guardian's forced_abort handler is called, and the handler's arguments indicate that it is for a node or structure whose tickle-lock is held by the guardian, then the topaction will abort outstanding actions so that atomic locks are released.

- Node-Updater

  This topaction implements the timely sharing of changes by updating the document node window display if the node displayed in the window has been altered by another CES user. The topaction is skipped if the tickle-lock on the node displayed in the document node window is held by the editor front-end's user. The node_catalog getr_node operation is used to read the version stack for the node displayed so that the tickle-lock is not acquired.

- Image-Updater

  Like the node-updater topaction, this topaction also implements the timely sharing of changes. It updates the text image window display if a node displayed in the window has been altered by another CES user. If the document structure has been altered, then both the document outline window and the text image window are updated. The topaction is implemented so that it cycles through its update chores with one step of the cycle executed each time the topaction is run. The cycle begins with a check of the struct_cat if the structure for the document being viewed or edited has changed. Each following cycle checks the node_catalog for changes to the nodes in the text image, one node per cycle. Therefore, for a text image display of N nodes, it will take N+1 cycles of this topaction before the entire text image is updated. The topaction uses the struct_cat getr_struct operation and the node_catalog getr_node operation so that the tickle-locks are not acquired.

4.5 CES System Catalog

The CES system catalog guardian is used by all of the CES guardians in a CES installation to locate other CES guardians in the installation. For bootstrapping purposes, access to the CES system catalog is provided by the Argus system catalog. The Argus system catalog is a coarse directory of all of the guardians in an Argus system. The Argus system catalog therefore cuts across applications. In contrast, the CES system catalog is CES application specific.

The CES system catalog gates the availability of an entire CES installation because all of the CES guardians use it to locate one another. Clearly, if the catalog is not functional, then CES guardians cannot locate each other and inter-guardian communication is impossible. Therefore, the catalog guardian is designed to be replicated. The catalog guardian replication techniques are identical to those employed for the document registry guardian. Gifford's algorithm is used to make the catalog a highly available resource. As is the case with the document registry, the quorum sizes are chosen to favor readers.

The CES system catalog is essentially a look-up table that is keyed by guardian type and network address. The catalog accepts queries for an available document registry or document library guardian. A document registry query will return the name of the document registry guardian that has the same network address as specified in the query; otherwise the name of a random document registry is returned. The document library query returns the name of the guardian that has the same network address as specified in the query; otherwise an error is signalled.

The CES system catalog also provides a utility to configure CES installations. Communication with this utility is by an Argus system supplied user interface. Though this interface is less powerful than the CES user interface, it does not require an editor front-end or I/O guardian in order to function. The CES

system catalog provides a menu of commands. These commands select services that range from reviewing the current list document registry or document library guardians to creating new guardians or destroying existing guardians.

The set of commands for the creation and destruction of guardians allows a CES system administrator to expand an installation or to improve the responsiveness and resiliency of the installation. In order to add a new author, a document library guardian can be created at the author's host computer. In order to alter responsiveness and resiliency, both document registry guardians and catalog guardians can be created or destroyed. This gives a system administrator the flexibility to control responsiveness and resiliency by varying the location, number, and weight of document registry guardians and catalog guardians.

A particularly important command provided by the CES system catalog allows a CES system administrator to remove from the list of active guardians any guardians that no longer exist. Though the CES system catalog commands that destroy guardians appropriately update the lists, there are other ways to destroy a guardian. The Argus system provides various utilities to do this, and Argus, application program, or operating system errors can also destroy a guardian.

The cleaning up of the list is automatically handled by the catalog by "pinging" each guardian in a list to see if it still exists. The "ping" is a handler call that simply returns, without results. If the "ping" succeeds, then the guardian still exists. If the "ping" fails due to a communication error, then it is not possible to tell whether the guardian still exists. If the "ping" fails because the guardian no longer exists, then the guardian is removed from the appropriate list.

4.6 Observations about the Role of Argus in the CES Implementation

It became obvious during the early stages of the CES implementation that Argus is an extremely powerful tool for building distributed programs. Argus unquestionably insulates applications programmers from virtually all low level network and operating system issues. For example, only a few hours were spent implementing Gifford's algorithm, and within two months it was possible to install a network of CES guardians on multiple VAXes. The resulting system provided substantial functionality, including the most important of the distributed aspects of CES, such as replication for resiliency, configurability, and highly concurrent data types.

Despite the success of Argus as a prototyping tool, the CES implementation revealed a number of issues that could diminish the utility of the language. Some of these concern the Argus implementation, others relate to the language design. The remainder of this section describes how both of these categories of issues became apparent during the CES implementation.

4.6.1 Argus Linguistic Support

The classic language designer's dilemma is how to trade-off "ease of use" with "expressive power". The Argus designers also faced this dilemma in their attempt to provide solutions to the complex problem of implementing a distributed system within the uniform context of a programming language. For their approach to be successful, it must be easier for a programmer to use Argus to implement a distributed application than would be the case if the application was built from an assortment of tools. The conflict, though, is that languages tend to sacrifice flexibility and restrict the ways in which problems can be solved when they attempt to define succinct, integrated linguistic constructs. Experiences with CES suggest that the Argus designers have not resolved the

language designer's dilemma.

In Argus, the "ease of use"/"expressive power" trade-off is most apparent with regard to the specification of user-defined atomic types. CES provides examples of three different situations that illustrate some of the compromises that had to be made in order to implement the types. These include:

- The garbage collection necessary to reclaim items in an astack (See Section 4.3.1.2).

  Astack garbage collection is performed as a transparent adjunct to some of the astack operations. Unfortunately, the garbage collection is only transparent in terms of its effects on the state of the astack because the activity delays the processing of the operation which invoked it.

- The garbage collection necessary to reclaim entries in a node or structure catalog (See Section 4.3.1.1).

  Node and structure catalog garbage collection occurs after the document library guardian supporting the catalogs crashes, so that only the recovery of the guardian, and not its on-line activities, is delayed. This has two drawbacks, though. First, the prolonging of guardian recovery can be undesirable, particularly in the case of the node catalog which is not replicated and is therefore not a resilient resource. Second, until a guardian does crash, the amount of memory occupied by "empty" entries in the catalogs can grow to substantial proportions. This suggests that it may be necessary to periodically crash the guardian in order to force garbage collection.

- The busy waiting necessary in the I/O guardian in order to implement atomic I/O.

  Each I/O guardian handler call awakens an I/O guardian background process that monitors the commit/abort fate of the handler call and repaints the terminal display if the handler does indeed abort (See Section 4.4.1). If an action I/O does abort, then the background code makes sure that the terminal display agrees with the image of the display as it was before the action ran, maintained in the display_image atomic object. The problem with this technique is that

busy waiting wastes computing resources. This technique is apparently the only way in Argus to provide a timely response to an action termination.

- The inability to easily specify without sacrificing modularity that only portions of an object are stable.

  The last_time and last_ed fields of each catalog entry, used to implement the tickle-lock, do not have be kept in stable storage because the tickle-lock is inherently recoverable. (See Section 4.3.1.3). The cost of making these fields stable is that they occupy stable storage which is an expensive resource. In particular, the size of a last_time object grows each time the tickle-lock with which it is associated is acquired, so that substantial amounts of memory can be occupied by the last_time field of each node and structure catalog entry. The alternative is to maintain a separate volatile structure that maps last_time and last_ed fields to stable catalog entries. This scheme results in a complicated, fragmented structure for the node and structure catalogs, and does not benefit from the modular abstraction mechanisms provided by Argus.

A general solution to the problems itemized above is proposed by Weihl in [Weihl 84]. Weihl explains that the Argus designers opted for an *implicit* approach to the implementation of user defined atomic types. With this approach, Argus programmers must use built-in atomic types in order to provide recoverability for user defined atomic types. The alternative is to design Argus so that an *explicit* approach could be used. With the explicit approach, programmers provide *commit* and *abort* operations, and perhaps other atomic type maintenance operations, as part of the implementation of a user defined atomic type.

An explicit approach, like the one proposed by Weihl, would offer solutions to each of the problems itemized above. Explicitly defined commit and abort operations would obviate the need for garbage collection in types such as the astack and node and structure catalogs [Weihl 84]. Similarly, an explicitly defined display_image abort operation would make it unnecessary to have to busy-wait for the termination of an I/O action; instead, the display_image abort

operation would be implemented so that the portions of a display affected by an aborted action would be repainted when the abort is processed. Explicitly defined copy-to and copy-from stable storage operations would make it possible to isolate portions of a type's representation for stable maintenance without sacrificing the modularity.

Weihl observes that an explicit approach seems to offer more "expressive power", while an implicit approach is "easier to use". For the problems itemized above, such complex programming was required to implement the desired functionality that it is believed that an explicit approach would, in fact, have been "easier to use". This is because an explicit approach facilitates the localization of an abstraction's complexity to the abstraction itself. In CES, the limitations of an implicit approach resulted in a shifting of programming complexity from the implementation of user defined atomic types to the code that uses these types. In the I/O guardian, for example, the code that implements display recoverability could be centralized in an explicitly implemented abstraction rather than being spread throughout the guardian. Consequently, it is believed that further investigation by the Argus designers into the tools provided for the implementation of user defined atomic types is warranted.

4.6.2 Argus Implementation Subtleties

Perhaps the most impressive thing about Argus it that it works. In addition to a compiler, linker, and debugger, Argus programs are supported by a substantial run-time system that tends to the numerous "behind the scenes" chores that make Argus programs work. These chores range from support for guardian multi-processing to the management of the action system dynamics.

The extent of the implementation of the Argus run-time support package far exceeded expectations. In fact, only a few short-comings of the run-time system

were encountered during the CES implementation. These include:

- Intolerance for blocking operating system calls.

  It was originally thought that a separate guardian for I/O would be necessary. This is because operating system calls that wait for input will block an entire guardian, and not just the guardian "pseudo-process" that made the call (See Section 4.4). Only after the editor front-end and I/O guardians were implemented and tested did it become apparent that such a scheme was not required. It turns out that if a guardian is blocked by a call to the operating system while executing a handler call, the Argus run-time system will eventually declare the guardian as "unavailable" and terminate the call. This means that when the editor front-end guardian makes a call to the I/O guardian *get_input* handler, the call might not complete. With the current run-time system, the only way to deal with this shortcoming is to have the I/O guardian busy wait by making non-blocking operating system calls that test for input. This scheme obviates the need to support I/O using a dedicated guardian.

- When guardians crash, they do not release any locks that were held by the guardian's actions at the time of the crash.

  A crashed guardian that holds locks creates a problem with the tickle-lock implementation because an editor front-end guardian can obtain a tickle-lock, only to become blocked waiting for an atomic lock held by a editor front-end guardian that has crashed. Calling the crashed guardian's force_abort handler (See Section 4.3.1.3) is clearly of no use. There is no way for the programmer to work around this shortcoming. The only solution to the problem is that the run-time system must have the ability to detect that a guardian has crashed when queries for a lock held by the guardian are issued.

- Centralized implementation of the Argus system catalog.

  The CES system catalog is a highly available resource that is implemented as a set of replicated guardians. The availability of this resource is the limiting factor for the availability of each CES installation. The availability of the system catalog is in turn limited by the availability of the Argus system catalog. This is because a CES guardian must first query the Argus system catalog in order to get the name of a CES system catalog guardian (See Section 4.5).

The current run-time system implements the Argus system catalog as a truly centralized service, thereby severely compromising the availability of both catalogs. Consequently, until the Argus system catalog is itself replicated, the measures that were taken to make CES highly available are relegated to academic interest.

- Confusing interaction between the implementation of two-phase commit and the lock test primitive.

In an attempt to improve the efficiency of Argus programs, the implementation of the two-phase commit protocol allows sequentially executed topactions to run such that a topaction can start to run while the preceding topaction completes its two-phase commit protocol. Normally, this performance enhancement preserves the sequential relationship between the two topactions so that the second action will always view any committed effects of the first topaction.

Unfortunately, the coupling of the CES astack implementation with the editor front-end action structure created a scenario in which the sequential relationship between sequential topactions may not be preserved. The astack *fasttop* operation, called by an editor front-end topaction, uses an Argus lock testing primitive to examine the astack in order to read an astack item. This editor front-end topaction runs after an editor front-end action adds a new item to the astack. Though the first topaction has committed the addition of the item, the second topaction might not view the item. This is because the lock test primitive only tests for whether a lock can be immediately acquired. In the case of these topactions, two-phase commit messages from the first topaction might not arrive at the astack before the lock test for the second action is executed. Therefore, the second action does not view the effects of the first action.

One solution to this problem is to place constraints on the amount of time that the execution of two-phase commit has to complete. This could make implementations of Argus impossible, or at least extremely inefficient. Another solution is to explicitly accommodate the semantics of the Argus implementation of two-phase commit in the application program. In CES, this would mean providing a variant of the astack fasttop operation that used the Argus lock test-and-wait primitive instead of the lock test primitive. The lock test-and-wait primitive will wait for a lock that cannot be immediately

acquired. Of course, the cost of this approach is that it will reduce astack concurrency.

The fact that crashed guardians do not release locks is the most serious of the run-time system implementation short-comings. This is the only short-coming that does not have a reasonable work-around for the programmer. Busy-waiting is inefficient but easy to implement, the Argus system catalog can be avoided altogether at the expense of complicating guardian configurability, and semantics of sequential topactions can be adjusted to accommodate the lock test-and-wait primitive. In contrast, a crashed guardian that holds a lock can prevent a program from making progress or even cause it to deadlock. Without support from the run-time system to release locks held by crashed guardians, the task of designing resilient programs is substantially complicated.

Chapter Five

Summary and Conclusions

This thesis presented the design and implementation of a distributed program for collaborative editing. The research focused on designing the collaborative editing application and then implementing the design. The design includes specifications for how to confront a variety of distributed system architectural issues as well as the functionality of the application. The implementation, in the programming language Argus, explored the challenge of making the application design a reality.

5.1 Summary and Conclusions

5.1.1 Functionality

The functionality of CES is founded on a structured representation of documents. A CES document's structure, or organization, is explicit and separate from its content. A CES document's content is developed from individual textual components, called nodes. Nodes are linked together by the structural specification to provide a readable image of the document, called a text image.

The CES user interface provides three views of a document, each view in its own interface display window. One view is of the document's structure, represented as an indented outline. Each item in the outline uniquely corresponds to one of the document's nodes. The order in which items appear and the degree of

indentation relative to the surrounding items specify the hierarchical position of the associated node in the document's structure. The second view is of an individual node. The third view is of the document's text image. The text image provides context for node content by "pasting" them together in the order specified by the document's structural specification.

Both the outline and the text of a node can be edited using a real-time screen editor. Modifications to a document's structure are made by editing the outline and invoking the outline parser. The parser scans the outline and reorganizes the document's structure to reflect the structure described by the outline. The text image is also reorganized by the parser so that it agrees with the outline. Modifications to a document's content are performed node-by-node. A node, selected for editing, is displayed in isolation in an interface display window where its text can be edited. The text image is for viewing purposes only and cannot be directly changed by an author, but it does change automatically to reflect modifications to a document's structure or nodes.

In order to coordinate authors' activities, CES requires that an author obtain exclusive edit access to document's structure or node before either the structure or node can be edited. An author can obtain exclusive edit access on the desired document structure or node providing that it is not already held by another author. Once an author obtains exclusive edit access, it can be held for as long as the author actively edits the structure or node. If the author does not actively edit the structure or node for more than twenty seconds, then the author's exclusive edit access expires. Exclusive edit access will be automatically reacquired when the author initiates another editing activity, unless another author intervenes and obtains it.

As an author with exclusive edit access to a document's structure or one of its nodes edits the structure or node, immediate feedback for the changes made are

provided by the real-time editor. The changes are also propagated by CES in a timely fashion so that the other authors will see the changes made to the structure or node. The changes are accompanied by information describing who made the change and where the change was made. The changes circulate about a CES installation so that the interface displays are constantly being updated. An interface display thereby provides a dynamic view of the evolving document.

5.1.2 Architecture and Implementation

CES was implemented in the programming language Argus. Argus is designed to facilitate the implementation of distributed programs that employ a client/server architectural model. Further, the language explicitly supports programs in which data and state information are long-lived. Except possibly for CES time constraints, the CES architectural requirements compliment the Argus application paradigm. Argus linguistic support for guardians, actions, and atomic data types provides the basis for the CES implementation in which active CES resources are implemented as guardians, processing activity is structured using actions, and shared data is defined as atomic types.

The architecture of CES is comprised of three subsystems: the document registry, the document library, and the editor front-end. The subsystems are abstractions of CES resources and are characterized by differing degrees of responsiveness, resiliency, and long-livedness:

- The document registry is the CES installation-wide list of documents and is implemented by a guardian that can be replicated throughout the installation. This subsystem is long-lived and can be configured to be highly available and responsive.

- The document library is the CES installation-wide catalog of document structures and nodes and is implemented by a guardian, an instance of which is installed on each computer in the installation that is the host for a CES author. Each one of these guardians maintains a unique node mini-catalog, but the catalog of document

structures is effectively replicated. The functionality of the subsystem gracefully degrades, it is most responsive to local references, and it is long-lived.

- The editor front-end is the user interface to CES and is implemented by a guardian, a new instance of which is created each time a user initiates a CES session. The guardian is destroyed at the conclusion of the session. This subsystem will maintain its state between crashes, but it is only as available as the computer on which it resides.

A CES installation can be configured using a set of utilities provided by a fourth subsystem, the CES system catalog. The CES system catalog can be used to create, replicate, or delete guardians, thereby allowing the creation or reconfiguration of a CES installation. The subsystem is implemented by a guardian that can be replicated, also using system catalog utilities, throughout the CES installation. This subsystem is long-lived and can be configured to be highly available and responsive.

5.1.3 Conclusions

A general conclusion that can be made about the CES design is that in CES, the structured representation of documents has three substantial rewards. First, CES uses structural information about a document to enhance its author coordination mechanisms. Authors are consequently not burdened by complex coordination protocols. Second, the encapsulation of a document's content into discrete components is the basis for physically distributing a document. This distribution scheme allows documents to be shared while preserving the locality of ownership of content-level contributions. Third, the structured representation provides a framework for the incorporation of authoring tools not typically found in conventional, unstructured document editing systems. These tools include document critiquing and the ability to manipulate perspectives of the document view.

Several conclusions can be made about Argus based on the CES implementation. This implementation demonstrates that Argus is an extremely powerful tool for building distributed programs. The fact that only two months were required in order to implement the bulk of the CES design substantiates the conclusion that Argus successfully insulates programmers from virtually all low level network and operating systems issues. It is unlikely that CES could have been implemented so rapidly were this not the case. As to be expected, Argus is not perfect, and, in fact, the CES implementation revealed several short-comings of the language.

The lack of flexibility provided for the implementation of user defined atomic types caused most of the difficulties. Argus opts for an implicit approach to the definition of user defined atomic types. With this approach, Argus built-in atomic types must be used in order to provide recoverability for user defined atomic types. It is observed that an explicit approach, in which programmers implement commit, abort, and other atomic type maintenance operations, is preferable to the implicit approach, even at the expense of being more complicated. CES provides a number of examples where the set of solutions to a problem are unacceptably restricted by the implicit approach.

A major component of Argus is its run-time system. This package supports the numerous "behind the scenes" chores that make Argus programs work. The run-time system is most impressive because of the tremendous number of activities that it performs. With few exceptions, the run-time system successfully supports Argus programs. Nevertheless, it is observed that several improvements to the run-time system might enhance it. The improvements include altering the action system so that when guardians crash they release any locks held, implementing the Argus system catalog as a highly available resource, and providing non-blocking I/O primitives as alternatives to busy-waiting.

In addition to these conclusions, one of the primary goals of the CES project was

to investigate the utility of the application by testing it on a community of users. Unfortunately, the CES prototype's run-time performance is too slow for use in document writing. The cause of the poor run-time performance is not fully understood yet. The inability to obtain both subjective and objective feedback about CES makes it impossible to reach substantiated conclusions about the system's design. In particular, it has not been possible to answer the following questions:

- How often should interface displays be automatically updated? Alternatively, should users be able to configure the display update rate (within some limits), and possibly even be able to turn it off?
- Is the display of dynamically changing views of a document in the display interface confusing? In particular, is it possible for users to concentrate on editing in one window while the displays in the other windows are changing?
- Can users comfortably accept the graceful degradation of the system and still do work while the system is in a degraded mode, or do users prefer a simpler model of the system in which it either works, or does not work?
- Is the automatic expiration of exclusive edit access a feature or a nuisance? If it is a feature, what is a reasonable interval in which to ensure exclusive edit access (e.g. the CES prototype uses a twenty second interval)?
- In general, is the structured editing style of document editing preferred to more common editing of unstructured documents?

The answers to these questions are relevant not only to CES, but to other cooperative systems. This is because the questions address one of two basic themes. One theme concerns the relationship of time to cooperative systems. Do cooperative systems need to provide real-time synchronization of individuals' activities, or are asynchronous interactions acceptable? The other theme concerns user interfaces. Interfaces are the media of cooperative exchange and

possibly serve the primary role in determining how easy it will be for users to cooperate. The three windows provided by CES may provide just the right compromise between shared and private workspaces. Alternatively, the extremes of completely private or completely shared workspaces might better support cooperative activities. The search for answers to these questions will undoubtedly inspire other questions.

5.2 Extensions

There are several features that could be added to CES in order to enhance its functionality:

- A fourth access privilege, critique-access, could be introduced. Users with this kind of access to a document would be allowed to critique the document, but would not be allowed to modify the document proper.

- The ability to share nodes across documents could be provided, though this would introduce the logistical problem of how to restrict access to nodes. An author may not want a node shared with other documents until the node has matured to a certain point. Similarly, an author may never want a node to be shared. Therefore, the difficult part of implementing this feature is designing an acceptable node-level access restriction scheme.

- The ability to "publish" documents would provide an analogy to the paper publishing of documents. Once published, it would not be possible to edit a document; it would only be possible to view it and perhaps critique it. Publishing a document and the ability to share nodes are related features. For example, perhaps only the nodes of published documents could be shared.

All of these features are straightforward extensions of the current CES implementation. Though they could be added independently of each other, it should be noted that they all involve manipulating access to document structures and nodes. It might consequently be possible to implement a single, cohesive

mechanism that supports all three extensions.

In addition to expanding the functionality of CES, there are several extensions to Argus suggested by experiences with the CES implementation. These are in addition to suggestions that Argus support the explicit implementation of user-defined atomic types. The suggested extensions are:

- Optimization of inter-guardian communication when guardians physically reside on the same computer.

  This would mitigate some of the performance penalty for modularizing a program's functionality by using distinct guardians. For example, a guardian *class* could be defined, where instances of members of the class would reside on the same computer. Such a scheme would allow programmers to specify that certain guardians would only by instantiated on the same computer. The programmer could therefore choose between higher performance (assuming that communication between guardians on the same computer is more efficient than communication between guardians on different computers) at the cost of restricting configurability. In cases like the editor front-end and I/O guardian, pair-wise instances of the two guardians explicitly reside on the same computer, yet communication between the guardians still uses inter-processor, rather than inter-process, communication. A programmer would use guardian classes the same way that integer arithmetic is used as a limited, but more efficient, alternative to floating-point arithmetic.

- The ability for programmers to schedule actions.

  The Argus designers chose not to provide support for scheduling because of their doubt about how frequently programmers will need to prioritize actions [Liskov & Scheifler 83]. Experiences with CES indicate that some reasonably chosen mechanism would be better than no mechanism. A case in point is the I/O guardian, the implementation of which was complicated by the fact that it is not possible to guarantee that the background monitor for the termination of an I/O action will execute before the next I/O action (See Section 4.4.1). Another case is the force_abort handler used to couple the activities of the document library guardian and editor front-end guardian (See Section 4.3.1.3 and 4.4.2). This handler is

explicitly used to convey an urgent request; yet without being able to ensure that this handler call will execute at a higher priority than other editor front-end activities, the urgency of the handler cannot be conveyed by its implementation.

5.3 Future Work

The experiences with CES suggest two areas for further research. First, more experience with actually using cooperative applications like CES needs to be gathered in order to better understand how to design such applications. Second, the CES implementation should be analyzed in order to understand what aspects of it or of Argus are the causes for the implementation's poor performance.

5.3.1 Cooperative Applications

CES and applications like CES that are intended to serve a community of individuals are confronted by a host of issues not found in single-user systems. These issues include characterizing the behaviors and needs of an entire community and modulating these behaviors and needs by the fact that individuals within the community will be interacting with each other. CES is premised on the assumptions that it will be used by a community of cooperating individuals, and that these individuals desire computer support for their cooperative activities. Little is done by CES to prevent an author from maliciously undoing the work of other authors. CES coordinates, but it does not police.

Further research is needed to establish whether the designers of cooperative applications can produce viable systems that are based on implicit assumptions about the behavior and needs of the target community. One place to start is with an activity that requires a great deal of cooperation, such as collaborative editing, and investigate how a community acclimates, or does not acclimate, to

the application. For example, it is conceivable that CES users could be indoctrinated to the socially acceptable ways to use the system. Providing that the CES design assumptions are reasonable to start with, then teaching a community of users about how to respect the application may be an efficient means of obtaining the desired environment for CES.

5.3.2 CES Performance

It has been speculated that certain Argus deficiencies are the primary causes for the poor run-time performance of CES. However, further testing is required to determine the extent to which CES implementation inefficiencies are responsible and to determine whether or not interactive programs like CES are reasonable applications for Argus. It may be the case that given even an extremely efficient implementation of Argus, the processing and communication overhead of a network of guardians may still be too great for the support of interactive applications.

The Argus developers are currently planning to take two approaches to assessing the run-time performance of CES. The first approach is to install the application on a network of VAXes, where the network and each VAX are dedicated to processing just CES. The CES prototype has previously only been tested on multi-user VAXes connected by a heavily loaded network. CES on a dedicated system may be fast enough to satisfy users with its performance. The second approach is to obtain more objective and quantitative measures of the run-time performance of CES by using computer-gathered performance metrics. This will allow a profile of program activity to be constructed, including the frequency of calls to particular procedures and the duration of such calls. The data provided by such a profile can then be used to identify program bottlenecks and facilitate an objective analysis of the CES and Argus implementations.

5.4 Observations

The current trend towards networks of personal workstations and minicomputers as computational vehicles already provides the capability for individuals to share data; the logical extension is for networks to provide the capability for individuals to share the work that produces the data. This thesis explored, through the design and implementation of a collaborative editing system, the problem of creating an application that allows users to work together to write a document. As the understanding of the requirements for systems that support user cooperation increases, and as the tools for building such systems improve, designers will be able to provide powerful and innovative support for shared work.

References

[Akscyn & McCracken 84]
Akscyn, R. M., and McCracken, D. L.
The ZOG Approach to Database Managment.
In *Trends in Applications 1984*, pages 217-225. IEEE, July, 1984.

[Atkinson 84] Atkinson, R.
TED.
1984.
Documentation for TED is unpublished but is available from the Massachusetts Institute of Technology Laboratory for Computer Science.

[Birrell *et al.* 82] Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D.
Grapevine: An Exercise in Distributed Computing.
Communications of the ACM 25(4):260-274, April, 1982.

[Bloom 79] Bloom, T.
Synchronization Mechanisms for Modular Programming Languages.
Master's thesis, Massachusetts Institute of Technology, January, 1979.
Avaliable as Technical Report MIT/LCS/TR-211.

[Englebart 82] Englebart, D. C.
Toward High-Performance Knowledge Workers.
In *Office Automation Conference Digest*, pages 279-290. AFIPS, April, 1982.

[Gifford 79] Gifford, D. K.
Weighted Voting for Replicated Data.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150-162. ACM SIGOPS, Pacific Grove, CA, December, 1979.

[Gifford & Donahue 85]
Gifford, D. K., and Donahue, J. E.
Coordinating Independent Atomic Actions.
In *Proceedings of the IEEE CompCon85*, pages 92-94. IEEE, February, 1985.

[Gifford & Spector 84]
Gifford, D. K., and Spector, A.
The TWA Reservation System.
Communications of the ACM 27(7):650-665, July, 1984.

[Gray 81] Gray, J.
Notes on Database Operating Systems.
In *Lecture Notes in Computer Science: Operating Systems - - An Advanced Course*. Springer-Verlag, Berlin, 1981.

[Gray *et al.* 81] Gray, J. N., et al.
The Recovery Manager of the System R Database Manager.
ACM Computing Surveys 13(2), June, 1981.

[Lamport 78] Lamport, L.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[Liskov 84] Liskov, B.
Overview of the Argus Language and System.
February, 1984.
Programming Methodolgy Group Memo 40, Massachusetts Institute of Technology Laboratory for Computer Science.

[Liskov & Scheifler 83]
Liskov, B. & Scheifler, R.
Guardians and Actions: Linguistic Support of Robust, Distributed Programs.
ACM Transactions on Programming Languages and Systems 5(3):381-404, July, 1983.

[Liskov *et al.* 81] Liskov, B., *et al.*
CLU Reference Manual.
In Goos and Hartmanis (editor), *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[Liskov *et al.* 83] Liskov, B., *et al.*
Preliminary Argus Reference Manual
1983.
Programming Methodology Group Memo 39, MIT Laboratory for Computer Science.

[Liskov *et al.* 85] Liskov, B., Herlihy, M., and Gilbert, L.
Limitations of Remote Procedure Call and Static Process Structure for Distributed Computing.
September, 1985.
Programming Methodology Group Memo 41, Massachusetts Institute of Technology Laboratory for Computer Science.

[Lister 77] Lister, A.
The Problem of Nested Monitor Calls.
ACM Operating Systems Review :5-7, July, 1977.

[Oki, Liskov, & Scheifler 85]
Oki, B. M., Liskov, B., and Scheifler, R.
Reliable Object Storage to Support Atomic Actions.
December, 1985.
To appear in the Proceedings of the Tenth ACM Symposium on Operating System Principles, December, 1985.

[Price 67] Price, D. J.
Networks of Scientific Papers.
In Kochen, M. (editor), *The Growth of Knowledge*, pages 145-155. John Wiley & Sons, Inc., New York, 1967.

[Sarin 84] Sarin, S. K.
Interactive On-Line Conferences.
PhD thesis, Massachusetts Institute of Technology, December, 1984.
Avaliable as Technical Report MIT/LCS/TR-330.

[Stallman 81] Stallman, R.
EMACS: The Extensible, Customizable, Self-Documenting Display Editor.
In *Proceedings of the Symposium on Text Manipulation.* ACM SIGPLAN/SIGOA, June, 1981.

[Trigg 83] Trigg, R. H.
A Network-Based Approach to Text Handling.
PhD thesis, University of Maryland, November, 1983.

[Weihl 84] Weihl, W.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, March, 1984.
Avaliable as Technical Report MIT/LCS/TR-314.

[Weihl & Liskov 82]
Weihl, W. and Liskov, B.
Implementation of Resilient, Atomic Data Types.
December, 1982.
Programming Methodology Group Memo 223-1, Massachusetts Institute of Technology Laboratory for Computer Science (Revised June, 1984).

[Wood 79] Wood, W. G.
Recovery Control of Communicating Processes in a Distributed System.
Technical Report 158, University of Newcastle-upon-Tyne Computing Laboratory, March, 1979.

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 4/29/96

Report # LCS-TR-350

Each of the following should be identified by a checkmark:
Originating Department:

- ☐ Artificial Intellegence Laboratory (AI)
- ☒ Laboratory for Computer Science (LCS)

Document Type:

- ☒ Technical Report (TR)
- ☐ Technical Memo (TM)
- ☐ Other:_____

Document Information

Number of pages: 167 (172 - IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
- ☒ Single-sided or
- ☐ Double-sided

Intended to be printed as :
- ☐ Single-sided or
- ☒ Double-sided

Print type:
- ☐ Typewriter
- ☐ Offset Press
- ☐ Laser Print
- ☐ InkJet Printer
- ☒ Unknown
- ☐ Other:_____

Check each if included with document:
- ☐ DOD Form
- ☐ Funding Agent Form
- ☐ Cover Page
- ☒ Spine
- ☐ Printers Notes
- ☐ Photo negatives
- ☐ Other: _____

Page Data:

Blank Pages (by page number): 2, 12,

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-167) UNNUMBERED TITLE PAGE, 2-167	
(168-172) SCAN CONTROL, SPINE, TRGT'S (2)	

Scanning Agent Signoff:
Date Received: 4/22/96 Date Scanned: 4/22/96 Date Returned: 5/2/96

Scanning Agent Signature: Michael J. Gough

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned
Date: 04/23/1996
M.I.T. Libraries
Document Services

darptrgt.wpw Rev. 9/94