Interactive On-Line Conferences

by

Sunil Kumar Sarin

B.Tech., Indian Institute of Technology (1974)
S.M., Massachusetts Institute of Technology (1977)
E.E., Massachusetts Institute of Technology (1978)

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

June 1984

© Massachusetts Institute of Technology 1984

Signature of Author . . . . Signature Redacted
Department of Electrical Engineering and Computer Science
June 15, 1984

Certified by . . . . Signature Redacted
Dr. Irene Greif, Thesis Supervisor

Accepted by . . . . Signature Redacted
Prof. Arthur C. Smith, Chairman, E.E.C.S. Departmental Committee

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 0 4 1984
LIBRARIES

ARCHIVES

Interactive On-Line Conferences

by

Sunil Kumar Sarin

Submitted to the
Department of Electrical Engineering and Computer Science
on June 15, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

A *real-time conference* allows a group of users, each at his or her own workstation, to conduct a problem-solving meeting by collectively viewing and manipulating a shared space of on-line application information while using a voice communication channel for discussion and negotiation. A real-time conference thus supplements asynchronous communication services, such as electronic mail, by permitting simultaneous manipulation of shared information.

Techniques are presented in this thesis for designing and implementing real-time conferences on distributed computer systems. Useful functions for real-time conferences are proposed: control over who may join a conference, and over concurrent participant commands within a conference; "private" spaces and transfer of information between shared and private spaces; display of "status" information describing who is present in a conference and who is entering commands; eliciting and collecting responses to be used in decision-making; planning a conference and making it known to potential participants; and maintaining and reviewing records of a conference. The tradeoffs involved in selecting a comprehensible set of commands for these functions are illustrated by the detailed design of a real-time conferencing system for joint editing of documents.

Different implementation techniques for real-time conferences are evaluated, with particular emphasis on minimizing the response time of the system to participants' commands. Criteria for choosing among alternative techniques are provided to guide a system designer in tailoring a real-time conferencing system for his particular application, user community, and implementation environment. An architecture is presented that permits dynamic selection of the implementation method based on run-time performance parameters. The feasibility of implementing real-time conferences using the given techniques is demonstrated by two prototype systems.

Thesis supervisor: Dr. Irene Greif
Title: Principal Research Associate

Acknowledgments

The greater the number of years one spends on a project like this (I will leave it to the reader to determine just how many), the greater the number of people one must thank for contributions large and small.

I would like to thank my thesis committee, Irene Greif, Michael Hammer, and J.C.R. Licklider, first for reading this thesis, and second for the feedback that encouraged me to continue an undertaking that they considered worthwhile. As my thesis supervisor, Dr. Irene Greif very patiently read and endured numerous rough and long-winded drafts. Her repeated exhortations to bring out the essential points of the research, what it's useful for and to whom, have considerably enhanced the quality of this document. Irene arranged for financial support even as I extended my stay, and was generous with advice about matters unrelated to the thesis, for both of which I am grateful. Prof. Michael Hammer led me through the early years of my graduate schooling, including my Master's research, and was an able instructor in the art of writing and speaking. He is in many ways the principal instigator of this thesis, having suggested the topic when I was looking for one. Prof. Licklider's obvious curiosity and interest in the subject of my thesis provided a useful bolster. He has never failed to surprise me with fresh insights into the problem, and my only regret is that our meetings were so few and far between.

The erstwhile Office Automation group, and its predecessor the Data Base Systems group, provided a fertile environment for the development of ideas, as well as some wild and crazy times. Brian Berkowitz, Dan Carnese, Arvola Chan, John Cimral, Ed Gilbert, Jay Kunin, Dennis McLeod, Bern Niamir, and Stan Zdonik deserve special mention. Andrea Aparo provided a needed breath of fresh air during the year he visited with us. Members of the Programming Technology group (the "other half" of the second floor), especially Tim Anderson, Dave Lebling, and Chris Reeve, contributed to the atmosphere of friendly anarchy, and provided occasional systems help as well. It was a pleasure to work with the many undergraduates who contributed to "the calendar project", especially Michael How, and to help them through the agony of learning to debug the code. David Brackman implemented a "shared virtual terminal" prototype in one short summer, and I regret not having worked closely enough with him to develop a more complete system.

Many members of the Computer Systems group (and its many incarnations and offshoots) have provided input as well. Prof. Jerry Saltzer conducted a seminar a few years ago that greatly increased my understanding of, and interest in, distributed systems. More recently, Prof. Dave Reed and Dr. Dave Clark conducted a seminar that unlocked some of the mysteries of computer network protocols, a fascinating subject of which I have only skimmed the surface. Larry Allen provided a wealth of background information; he had little choice but to do so, we rode the same bus many times. Michael Greenwald provided help with the "Blink" protocol (designed by David Reed), and Lixia Zhang was kind enough to let me use her simulator program to test out my own extension of the protocol, MBlink. They may no longer remember it, but I also had productive discussions with Bob Baldwin and Karen Sollins.

The CLU language and system, developed by Bob Scheifler and Paul Johnson and others under the leadership of Prof. Barbara Liskov, and Maurice Herlihy's message-passing extension, greatly eased the task of programming the prototypes described in this thesis.

Considerable use was also made of a myriad of text processing, mail handling, programming, and debugging tools developed in the unique creative environment of the Artificial Intelligence and Computer Science Laboratories. The Department of Defense Arpanet and Internet have made it possible for me to access a large amount of network and other documentation without taking up anybody else's time, and to interact professionally with many individuals whom I have never met and perhaps never will.

This document was produced using Unilogic Inc.'s Scribe™ document compiler, and a "Dover" laser printer and related software developed by the Xerox corporation. I would like to thank the research and technical staff members who kept the computers and networks and printers running: Joe Ricchio, Tyrone Sealy, Steve Berlin, Robert Myhill, and many others whose names I will never know.

The important efforts of the support staff at the Laboratory for Computer Science go largely unrecognized. I would like to thank Linda Bragman, Ann Finn, Nancye Mims, Margie Nieuwkirk, Anne Rubin, Janet Schoof, Renata Sorkin, and all the others who have kept things running smoothly over the years. Maria Sensale of the Reading Room has been most helpful, and went to great lengths to acquire a report that I wanted to read.

The late Prof. Bill Martin arranged for financial support during my first year at M.I.T., while I was still finding my way around. The I.B.M. Corporation awarded me a Graduate Fellowship that supported me for one year. Without continued funding from the Defense Advanced Research Projects Agency, this research would never have taken place.

At the Indian Institute of Technology, Kharagpur, Amitava Bagchi got me started in computers through his own obvious enthusiasm with the subject. By demanding to absorb all of my new-found knowledge, Sajan Punwani helped me learn to clearly communicate that knowledge, and to better understand it myself.

My parents, Veena and Dharam Pall Sarin, have always been there for me, for longer than I can remember (naturally). My parents and my elder brother Vinay are responsible for injecting me with a thirst for knowledge and for whatever level of intellectual ability I have been able to attain. My younger brother Sudhir ("Sunny") kept me from lapsing into self-absorption by never giving me a moment's rest whenever I was home. His curiosity and persistent questioning ("But why?") sharpened my wits by forcing me to think. A year after his untimely death, his affection and concern for people and his total immersion in everything he did remain an inspiration to all who knew him.

Leslie Gordon has been the brightest element of my long and arduous graduate student career. She has provided love and encouragement far beyond the call of duty. She demonstrated infinite patience by listening to my endless complaints about real and imagined lack of progress, withheld her own complaints when my thesis kept me busy and irritable, and provided the laughter that was needed when life looked bleaker than it really was. Leslie's contribution to this thesis cannot be measured; if I am able to give her even half the support she gives me, I will consider that an accomplishment.

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

Table of Contents

Chapter One: Introduction 9

1.1 Asynchronous and Real-Time Interactions 9
1.2 Benefits of Real-Time Conferencing 10
1.3 Scope of Thesis 11
1.4 Functional Design of Conferences 13
1.5 Implementing Conferences 15
1.6 Related Work 16
1.6.1 Audiographics Conferencing 17
1.6.2 Electronic Blackboards 17
1.6.3 Virtual Terminal Linking 17
1.6.4 Application-Specific Conferencing 20
1.7 Outline of Thesis 21

Chapter Two: Example: Joint Document Editing 22

2.1 System Environment 23
2.1.1 Communications and Storage 23
2.1.2 Workstation Interface 25
2.2 Editing by Individual Users 26
2.2.1 Document Structure and Operations 27
2.2.2 Document Windows 30
2.2.3 Editor Interface 31
2.2.4 History-Related Commands 38
2.2.5 Access Control 38
2.2.6 Concurrency Control 39
2.3 Real-Time Conference Interface 42
2.3.1 Windows and Reservations 42
2.3.2 Negotiating a Window Size 48
2.3.3 Document Access Control 50
2.3.4 Access to a Conference 51
2.3.5 Leaving and Termination 52

Chapter Three: Design of Real-Time Conferences 53

3.1 Shared and Private Spaces 55
3.2 Access Control 59
3.3 Concurrency Control 61
3.3.1 Validation 62
3.3.2 Reservations 64
3.3.3 Comparison 67
3.4 Meeting Support Functions 68
3.4.1 "Chairpersons" and Participant "Roles" 68
3.4.2 Participant Status Information 69

3.4.3 Interjections 72
3.4.4 Negotiations 73
3.4.5 Access to a Conference 75
3.4.6 Rendezvous 76
3.4.7 Planning and Scheduling 77
3.4.8 Joining a Conference 78
3.4.9 Leaving a Conference 79
3.4.10 Conference Termination 79
3.4.11 Record-Keeping and Review 80
3.5 Two-Person Conferences 83
3.6 Voice Communication 85
3.7 Summary of Design Issues 86

Chapter Four: Implementation of Real-Time Conferences 90

4.1 Distributed System Model 90
4.1.1 Sites and Messages 91
4.1.2 Object Model 92
4.2 Real-Time Conferences 98
4.2.1 Conference Objects 98
4.2.2 Controller and Workstation Sites 104
4.3 Input and Update Processing 106
4.3.1 Workstation Interface 106
4.3.2 Input Processing 109
4.3.3 Update Generation 111
4.3.4 Synchronization Points 114
4.4 Replication Strategies 115
4.4.1 Full Replication 115
4.4.2 Partial Replication 117
4.4.3 Private Spaces 120
4.5 Adding and Removing Participants 124
4.5.1 Immediate Initialization 124
4.5.2 Asynchronous Initialization 125
4.5.3 History-Based Initialization 126
4.5.4 Removing a Participant 128
4.6 Transport Protocols 129
4.6.1 Message Encodings 130
4.6.2 Virtual Circuit Efficiency 131
4.6.3 Reliable Multicast 133
4.6.4 Non-Sequential Datagram Protocols 138
4.7 Improving Interactive Response 143
4.7.1 Many-to-Many Reliable Multicast 144
4.7.2 Adding and Removing Workstations 147
4.7.3 Displaying Unconfirmed Results 148
4.8 Using the Implementation Techniques 151

Chapter Five: Conference System Architecture 152

5.1 Dynamic Conference Control 152

5.1.1 Groups of Sites 153
5.1.2 Conference Descriptions 156
5.1.3 An Example 156
5.1.4 Adding and Removing Sites 158
5.1.5 Negotiating a Change in Protocol 161
5.2 Conference Management 165
5.2.1 Multiple Conference Types and Applications 165
5.2.2 Selecting a Controller Site 166
5.2.3 Backup Controllers 167
5.2.4 Moving the Controller Site 168
5.2.5 Controller Crash Recovery 172
5.2.6 Combining Conferences 174
5.2.7 Server Sites in a Conference 176
5.3 Decentralized Control 180
5.3.1 Locking and Timestamps 181
5.3.2 Token-Passing 182
5.3.3 Two-Person Conferences 183

Chapter Six: Prototype Implementations 187

6.1 Meeting Scheduling in a Real-Time Conference 187
6.1.1 RTCAL Functions 188
6.1.2 Experience with RTCAL 191
6.2 Shared Bitmap System 192
6.2.1 The Blink Protocol 193
6.2.2 Handling Multiple Workstations 196
6.2.3 Remote Mouse-Tracking 198
6.2.4 Experience with MBlink 199
6.3 Extended Bitmap System 200
6.3.1 XMBlink Functions 201
6.3.2 Application Program Interface 205
6.3.3 Transport Protocol 208
6.3.4 Sample Application 210

Chapter Seven: Summary and Conclusion 214

7.1 Future Directions 216
7.2 Relevance to Distributed Computing 217

References 223

Table of Figures

Figure 2-1: A Document Window 30
Figure 2-2: Alternative Document Versions 41
Figure 2-3: Shared and Private Spaces 43
Figure 2-4: Shared Space in JEDI 44
Figure 3-1: Pointer Shapes for Two-Person Conference 84
Figure 4-1: Conference Object Specification for JEDI 99
Figure 4-2: Example Document Structure 103
Figure 5-1: Example Group Graph 154
Figure 5-2: Group Graph for JEDI-CONF 157
Figure 5-3: Ensemble Operations on Groups 159
Figure 6-1: Example RTCAL Display 189
Figure 6-2: Architecture of Shared Bitmap System 193
Figure 6-3: Tracking of Workstation Pointers 199
Figure 6-4: XMBlink Conference Object Specification 202
Figure 6-5: MBlink Signals 205
Figure 6-6: Example Crossword Display 211

Chapter One

Introduction

This thesis addresses the problem of supporting and enhancing remote problem-solving meetings and conferences through access to on-line information. We present both interface design and implementation techniques for "real-time conferences" in which a group of users, each at his or her own workstation, collectively view and manipulate a shared space of problem information while using a voice communication channel for discussion and negotiation.

1.1 Asynchronous and Real-Time Interactions

Interactions among people fall into two main categories: *real-time* interactions such as face-to-face meetings or telephone conversations, and *asynchronous* interactions such as correspondence via the postal service. Each is appropriate for different situations, and neither is likely to completely replace the other. Asynchronous interaction can be extremely useful and efficient in that each communicator can act at a time and pace of his own choosing. However, it is often the case that after a sequence of asynchronous exchanges of information, proposals, and counterproposals it is necessary to negotiate in real time in order to resolve outstanding issues. Rapid group decision-making in a crisis situation also requires real-time interaction.

With ongoing advances in technology and declining costs, computers are being increasingly used to aid problem-solving and decision-making, in business, design and manufacturing, the military, and other areas. General-purpose tools such as electronic mail [52], computer conferencing [56, 76], and form management [115], have been developed for supporting groups of people engaged in such problem-solving, and many such tools are now widely used. These tools, however, support asynchronous communication only. Real-time interaction has been largely ignored in these systems, except for very primitive and cumbersome facilities such as typed text messages or "linking" of terminal input and output.

Our research is concerned with *real-time conferences*, in which groups of users at interconnected workstations can simultaneously view and manipulate on-line problem information in more interesting and useful ways.[1]

1.2 Benefits of Real-Time Conferencing

The benefits of real-time conferencing using on-line information can be viewed in at least two ways: as an enhancement to interactive computing, or as an enhancement to telecommunications. Real-time conferencing increases the usefulness of interactive computing because the same computing tools that are available to a user when working alone at his workstation can also be used in a meeting; this is important because a sizable fraction of people's work time is spent interacting with others.

In contrast, telecommunication facilities, from the telephone through video "teleconferencing", provide little or no access to information of common interest. Real-time conferencing places the full power of the computer, to store, retrieve, edit, and process information, at the disposal of users who are interacting across a long or short distance. Participants can point at or refer to displayed information while conversing, with the assurance that the others know what they are referring to. Information whose need was not anticipated can be accessed, and new alternatives can be explored and analyzed on the fly. (These benefits of on-line information access can also be realized in face-to-face interactions, by placing appropriate equipment in conference rooms, but that is not the focus of our research.)

A somewhat different benefit of real-time conferencing is reducing the need to *travel* to meetings. (Such "travel" does not always involve very long distances; e.g., employees of many large corporations often have to travel between different locations of the corporation within the same city.) We do not claim travel avoidance as a unique benefit of real-time conferencing, because it applies equally to other forms of "teleconferencing" such as video communication and "electronic blackboards". However, when access to on-line information is an important requirement for a given meeting, video and blackboard communication is not

[1] Other terms that have been used to describe a similar concept include "shared-screen teleconferencing", "data conferencing", and "desk-to-desk conferencing".

adequate; a real-time conference is needed if travel is to be avoided. We also note that travel avoidance, for any kind of remote meeting, can be viewed not only in terms of the savings in time, money, and fuel, but also in a more positive way. That is, not having to travel to a conference may allow more people who are interested to attend. And, because relevant information does not have to be collected in advance and "carried" to a conference, more information will be accessible. (We note that Brecht's study of meetings and conferences [11] cites inadequate access to information as a significant problem in many meetings.)

1.3 Scope of Thesis

A real-time conference as we envision it has a small to moderately large group of participants, each seated at a workstation, with the workstations connected by some combination of local and long-haul communication networks or lines. The participants in such a conference interact mainly as peers, in a cooperative atmosphere where the emphasis is on achieving a common goal, e.g., reaching agreement on a design or strategic decision, using on-line information relevant to the problem.

This thesis examines in detail the issues that must be addressed when designing and implementing real-time conferences of the above kinds. The goals of this thesis are twofold:

1. To define a set of useful *functions* that should be made available to users participating in a real-time conference.
2. To develop techniques for *implementing* the above functions efficiently using current and near-future computer and communication technology.

Our description of conferencing functions and implementation techniques is directed at the *designer* of a real-time conferencing system. We illustrate the design process by developing an example real-time conferencing system in detail. The functions and techniques are then discussed in general terms for a wide range of applications, allowing the designer to select those functions and techniques that are best suited to his particular application, user community, and implementation environment. To aid the designer in this undertaking, we present different options and compare them in terms of their effects on user interface complexity and on efficiency of implementation.

We emphasize that real-time conferences are not intended to be the sole means of communication in a system. Most interaction using shared information is likely to take the

form of asynchronous individual actions over a period of time, with an occasional real-time conference being held when necessary, e.g., because outstanding disagreements remain or because of an impending deadline.

Asynchronous communication can and has been used to record and structure users' discussion of a problem in the form of commentary, arguments, proposals, and responses; this is done using typed text as the medium in several "computer conferencing" systems. In a real-time conference, the use of typed text for discussion of the problem at hand is usually too slow and cumbersome to be practical. We instead expect that *voice* communication will be necessary for discussion and negotiation in a real-time conference.[2] This thesis does not address how voice communication is to be provided in a real-time conference. We will assume the availibility of voice communication, and concentrate on how shared on-line information is managed in a conference. An ideal "integrated" system would include microphones and speakers in every workstation, and would automatically set up the voice connections for a conference at the same time as the on-line "data" connections. Until such integrated hardware and software is universally available, it may be necessary for participants to establish voice communications separately, e.g., using a "conference call" on the telephone network. Similar arguments apply to various forms of *video* communication that may sometimes be useful in transmitting non-verbal sources of feedback such as gestures and facial expressions. Video communication requires more expensive equipment and greater communication bandwidth than voice, and studies by Chapanis [16] suggest that the usefulness of video in helping a group solve a problem is not significantly greater than voice alone. It is also hard to concentrate on both displayed on-line information and participants' faces at the same time. We therefore consider video communication optional for real-time conferences, and again assume that it will be set up separately if needed.

We do not propose real-time conferences as a replacement for all simultaneous interactions among people. Interactions whose primary purpose is social or political will benefit little from on-line information access; a face-to-face meeting will usually be necessary, with video teleconferencing as a second choice when travel is a problem. However, for meetings where problem-solving using on-line information is the primary objective, and the

---

[2] Typed text communication should not be ruled out completely, especially since voice communication is useless for users with hearing disabilities. In other situations, voice communication equipment may not be available, or conference calls may not be supported, or a user's only telephone line may be taken up by his terminal.

social component of the interaction is secondary (e.g., the participants already have an ongoing work relationship), a real-time conference should be the ideal mode of interaction.

We do assume that a real-time conference is conducted in an atmosphere that is mostly cooperative. Participants may modify or overwrite each others' contributions and updates if they believe that will help in solving their problem. Disagreements among participants will no doubt arise, but we assume that the participants resolve their differences by social and organizational processes outside the system. They may use the voice channel during the conference, or deal with conflicts and disagreements after the conference. It will be useful for the system to keep histories and records to support this process, so that participants' actions can be reviewed and "undone".

While the functions and implementation techniques we describe are applicable to large conferences involving perhaps tens of participants, most real-time conferences in practice will involve only a few participants. We also expect that many "conferences" will involve only two participants, just as many face-to-face interactions and telephone calls do. This thesis therefore includes special kinds of functions, and special optimizations at the implementation level, that can be used for the particular case of two-person conferences.

The remainder of this chapter provides an overview of our design and implementation techniques for real-time conferences, and compares our work with other work in the same and related fields.

1.4 Functional Design of Conferences

The primary function of a real-time conference is to provide a *shared space* of information that the conference participants can display on their workstation screens and that they can manipulate (e.g., edit or process) in some specified way. The precise nature of how information is displayed and edited will vary from application to application, e.g., document editing or circuit design, and a given conference will use known techniques from its particular domain. Our concern is with the additional functions that are needed to support and manage the simultaneous manipulation of a given shared space by multiple participants. Such functions include the display of "status" information showing who is present and who is entering commands, support for negotiation and decision-making by eliciting and collecting

responses, and control over who can join a conference and when. We describe these functions in terms of the abstract *objects* and *operations* needed in order to realize them.

We also consider the problem of multiple participants operating *concurrently* on the shared space. This may often result in confusion, when a given participant's command has a different result from what he intended because of an intervening concurrent operation by some other participant. An extreme solution to this problem is to allow only one participant at a time to perform operations on the shared space. This is analogous to holding the "floor" in a face-to-face meeting or conference, and a set of commands for requesting and passing the floor can be defined. More concurrent activity can be permitted in a controlled way by participants setting *reservations* on parts of the shared space, e.g., on different displayed "windows" and on different application objects such as regions of documents.

A *private space* allows a conference participant to retrieve and edit and compose information outside of the shared space, invisible to the other participants. A private space is thus analogous to the private notes that one might bring to a face-to-face meeting. Commands for "crossing the boundary" between shared and private spaces should be provided, to allow a participant to "submit" selected information from his private space to the shared space for all participants to see and perhaps even edit, and to allow a participant to copy information from the shared space and privately "browse" or edit and explore alternatives. A related consideration is whether feedback from the system (command "echo", or error messages) in response to a given participant's actions should be shown privately to just that participant, or should be shown to all participants in the shared space. The latter is useful in giving other participants an indication that a participant is doing something and not sitting idle (just as one can be observed to be drawing on a blackboard even though the drawing is not complete and ready for discussion), but requires that a participant be able to distinguish the feedback that was caused by his own commands from feedback generated by the other participants.

Selecting a set of functions to provide in a real-time conference can be considerably more difficult than the same task for a single-user interactive system, because of the wider range of options available. Different choices may be appropriate for different applications and user communities, or even for individual users with different preferences. The implementation environment (processors and communication links) will in many cases constrain the

functionality that can be achieved with acceptable response.[3] However, even within a given set of implementation constraints the range of design choices may be considerable, and the main task of a conference system designer is making a selection of useful functions that will not overwhelm the user with their complexity. We illustrate this design process in detail by taking an example application, namely joint document editing, and explaining the reasoning behind a set of choices that achieves a balance between user interface simplicity and full functionality.

1.5 Implementing Conferences

The implementation techniques we present for real-time conferences use many well-known methods from the areas of computer networking and distributed databases. Our main contributions in these areas are an evaluation of which techniques are suitable for real-time conferencing and when, and special adaptations and extensions of existing techniques for the particular needs of real-time conferencing. In particular, many existing techniques for interactive distributed computing deal with a single user's workstation and a remote host; these techniques require modification for the case of multiple workstations in a conference.

We present an architecture in which a real-time conference is implemented by a *controller* site, which manages the shared space of the conference, interacting with participants' *workstation* sites to realize the desired functionality. We describe techniques for replicating parts of the shared space at the workstations, for propagating updates to replicated information, and for supplying newly-joining workstations with replicated information. The techniques are evaluated on the basis of their effect on response time under various conditions. Response time can often be improved by easing requirements for "consistency" of the information seen by the participants. We describe a method whereby temporarily inconsistent data may be shown to the participants, but is quickly corrected so as to achieve eventual consistency among all participants' views. This can be tolerated by participants if it occurs infrequently; the use of "reservations" to control concurrent activity is shown to considerably reduce the probability of seeing inconsistent data.

[3] To be truly "interactive", the system should respond to a participant's command within a small fraction of a second [84]. Commands that invoke lengthy computations or disk reads and writes may take longer to process overall, but some immediate feedback should be given that the system is working on the command.

Performance for a given real-time conference can be optimized by dynamically selecting the protocol used for replicating objects based on workstation and network capabilities. In an architecture that we have named *Ensemble*, any protocol parameters that are dynamically determined are included in the shared space of the conference as "meta-information" that may be replicated at workstations. Then, a workstation wishing to join an existing group of workstations in a conference can examine the protocol meta-information in order to determine whether or not it is capable of supporting the current protocol. When the controller site wishes to select an initial protocol for a group of workstations, it initiates a *negotiation* procedure in which each workstation specifies what range of protocols it is capable of supporting. Based on the responses received, the controller can then make a choice that trades off better performance against including as many workstations as possible. A similar negotiation allows the controller to dynamically adjust the protocol as workstations join and leave the conference or as network conditions change.

The integration of real-time conferences into the overall distributed system environment is examined. This includes techniques for publicizing a conference and inviting users to join, and for making a description of the implementation protocol available to participants' workstations. The interaction between real-time conferences and other system services such as permanent storage is also discussed.

1.6 Related Work

This section compares our real-time conferencing functions with other systems that support simultaneous interaction among a group of users. We do not discuss related implementation techniques in this section, but postpone such discussion until our own implementation techniques are presented in more detail.

We also do not discuss voice and video teleconferencing systems [93] below, because these address a different problem. Our research is concerned with the presentation and manipulation of information in remote meetings, under the assumption that voice communication, and video if needed, is already available.

1.6.1 Audiographics Conferencing

Several informational aids have been developed for so-called "audiographics" conferencing, that combines voice conferencing with transmission of on-line or facsimile information. Most of these can be characterized as *message*-oriented, in that some participant "sends" an item of information, say a document or drawing, that is displayed at all locations. This is the case, for example, with the "data bridge" of AT&T's Network Services Complex [80]. (It also applies to the so-called "synchronous" mode in computer conferencing systems, where participants' contributions consist of text messages.) Some systems (e.g., "telewriting" using a stylus and tablet [65]) may allow modification of the information transmitted, but each image can only be modified from its source location, not by the others. What all of these lack is the ability of multiple users to interactively edit and modify a common shared object, such as a blackboard in a face-to-face meeting or an on-line "shared space" in the kind of real-time conference that we are concerned with. One-way submission and editing of documents, messages, or pictures may still be useful in a real-time conference, but only as a special case.

1.6.2 Electronic Blackboards

Closer to our concept of real-time conferencing is AT&T's "Electronic Blackboard", which does present the illusion of a shared object that can be modified from both (or all) locations. Electronic blackboards are, however, only as useful as real blackboards; the only shared "object" that can be manipulated is the blackboard image itself. This is usually good enough for drawing and annotating, but for little else. In a conference held from on-line workstations, the participants can manipulate not just displayed images but a collection of objects with rich structure and application semantics that can be viewed and edited and processed in more interesting and useful ways. Again, an electronic blackboard can be implemented in an on-line conference by defining an appropriate type of object and set of operations, but this is only a special case that does not fully exploit the power of the computer.

1.6.3 Virtual Terminal Linking

Terminal linking, a facility popularized by NLS [29, 27] and now available on many operating systems, can be used to provide a limited form of "real-time conferencing". When two or more terminals are linked, output directed to any one of them is sent to all of them. It is also possible to allow a terminal to "advise" another terminal, in that input typed at the first

terminal is directed to and processed by the program running at the second terminal. Thus, a given application program, say a text editor or spreadsheet, can be "shared" in a real-time conference by linking terminals and allowing the terminals to enter input to the given program. This kind of "conference" is in fact quite common when a user encounters a problem with a program and notifies a programmer who links in his own terminal in order to debug the problem; such remote debugging is often accompanied by a telephone conversation as well. (Terminal linking is also commonly used for typed text communication, but this is effective only for very short messages and does not involve joint interaction on a shared object.)

Terminal linking as currently available on most systems is very primitive and inconvenient; for example, linking of terminals at the level of their hardware input/output buffers does not work well when different terminal types, which expect different character sequences for screen control, are linked. These problems can be remedied by defining a device-independent *virtual terminal* protocol and implementing translations for different physical terminal types; virtual rather than physical terminals can then be linked for a real-time conference. Tymshare's Augment™ system [28] (which evolved from NLS, above) supports virtual terminal linking, together with "floor-passing" commands that allow control over which participant's terminal can enter input to a given virtual terminal. (Such control is not available on most systems with terminal linking, leading to confusion when participants type simultaneously.) Augment currently supports the display of one user's entire screen image on one or more other users' screens. It would not be difficult to add support for multiple screen "windows" into different virtual terminals, some of which may be "shared" and others "private", and transfer of information between shared and private virtual terminals by copying output from one virtual terminal to the input of another virtual terminal. (These facilities are available to individual users on many systems, such as Rochester's VTMS [72], and are likely to be just as useful in real-time conferences.)

The main virtue of linking virtual terminals is that it is the only way of accessing existing application programs from a conference without modification to the software; this is important because of the considerable effort and money already invested in such software, and because conference participants can use programs with which they are already familiar. However, the adaptability of virtual terminals to arbitrary existing programs is their main limitation as well, because this adaptability comes from assuming the lowest common interface to all programs,

namely their input and output character streams. When information transfer is implemented, between the output of one virtual terminal and the input of another, it can only be done at the character stream level. High-level application objects cannot be easily transferred between applications, only their printed representations can. This requires that the program receiving copied output as input must be able to "parse" it correctly; while this might be possible with echoed commands or sometimes with text, it is difficult if not impossible with more complex information such as a component of a graph. Even more difficult is *sharing* of information between different virtual terminals, e.g., a participant in a private virtual terminal browses over a document that is also being edited via a shared virtual terminal. While this might be arranged by having both programs access the same file, it is not possible to edit the file within the address space of one program and have these changes reflected in the other program in "real time".

Because all information is transmitted at a very low level, virtual terminal linking also does not permit the capabilities of modern high-performance workstations to be exploited. When workstations are capable of maintaining application data and executing application software, transmitting low-level display operations instead of more compact high-level application operations is an unnecessary waste of communication bandwidth. High-level application-specific communication also allows workstations to transmit application objects stored on their local disks; this is not possible with virtual terminal linking.

Another problem with virtual (or physical) terminal linking is that a program being "shared" among a set of linked terminals still believes that it is interacting with a single user at a single terminal, and runs with a single user's access privileges and naming environment. If permission to enter input is given to a different user, his commands may have a different effect than what he would normally expect. Thus, unspecified parts of file names may be defaulted differently, or command customizations may be different. Worse, the user may be able to read or modify a file that he is normally not allowed to read or modify, or not be able to read or modify a file that he is normally allowed to. These problems can sometimes be tolerated if the "chairperson" (the user under whose name the program is running) controls who can enter input and intervenes by revoking input permission if he is not pleased with what a participant is doing. Other ad hoc solutions abound. In Bell Laboratories TOPES system [95], the operating system code that implements terminal linking is specially modified to indicate which of the linked terminals input was received from. Then, the application program

checks this information on every file read or write command and rejects the command unless it was typed by the chairperson. (Non-permanent display and editing commands, which do not touch files, are accepted from any participant.) Another approach is to program the system not to accept a "carriage return" character, and the system "attention" or "interrupt" character, from any terminal except the chairperson's. This only works for older teletype-based programs that accept commands a line at a time, and not for character-at-a-time programs such as screen-based text editors. Even for line-oriented programs, this can be rather restrictive because *every* command now requires the chairperson's explicit approval.

Since input to a conventional interactive program appears as a single character stream, virtual terminal linking does not allow input from different participants to be treated differently. Thus, concurrent participant activity via multiple cursors cannot be implemented within a given application program. If concurrent input is permitted from more than one participant, it is simply merged into a single stream as it arrives; this is a frequent source of irritation with terminal linking. Concurrent participant activity can be supported with multiple virtual terminals, but each virtual terminal must run a separate program instance which does not permit concurrent activity on common data.

1.6.4 Application-Specific Conferencing

The above limitations of virtual terminal linking can be overcome only by implementing new distributed application programs that explicitly recognize and interact with more than one user simultaneously, and that communicate with participants' workstations in high-level application terms. A few such application-specific conferencing systems have in fact been implemented. Structural Programming Inc.'s Palette™ drafting system supports a conferencing mode in which two users can jointly compose and edit a common drawing using independent cursors [85]. The two cursors have different appearances, one of which is identical to the cursor that a user sees when working alone. The cursors are displayed on the two workstation screens with their appearances exchanged; i.e., each participant sees his own cursor the way it normally appears when working alone, while the "other" participant's cursor has a different appearance. A participant's commands, e.g., to mark a corner of a rectangle, are interpreted based on the position of his own cursor.

Other undocumented conference system prototypes have been developed at the Xerox

Palo Alto Research Center, e.g., for joint document editing and circuit design.[4] These systems, and systems such as Palette, above, do in fact illustrate some of the functions we propose for real-time conferences, although they are somewhat inflexible. Many such systems, for example, are restricted to two participants only, or make no allowance for private "spaces". (Such restrictions may have been dictated by performance considerations.) Our research is concerned with exposing the design principles underlying such conferencing systems; as such, these systems (as well as terminal linking, synchronous message communication, and electronic blackboards) can be considered to be particular realizations of the ideas presented in this thesis.

1.7 Outline of Thesis

This chapter introduced the concept of a real-time conference, and provided a brief overview of our work and comparison with other work. Chapter 2 develops an example application area, namely joint document editing, in order to explore the design choices and tradeoffs that must be made. Chapter 3 generalizes from this exercise by listing the design issues that must be addressed for any real-time conferencing application, together with criteria for selecting among the different design choices available. Chapter 4 presents implementation techniques, considering both the overall distributed software organization for the sites (e.g., workstations) participating in a conference, and protocols for communicating commands and results. Chapter 5 describes dynamic selection of the implementation method for a conference based on run-time parameters and constraints, and issues relating to starting and managing conferences in the overall system environment. Chapter 6 describes two prototype systems: an early prototype that led to many of the ideas presented in this thesis, and a current prototype for experimenting with some of our protocol and architectural ideas. We conclude in Chapter 7 by summarizing the thesis and indicating directions for further research.

---

[4] Several distributed *games* have also been developed on the Ethernet environment at Xerox PARC and elsewhere, such as Amaze [6]. While there are some similarities, these differ somewhat from our notion of real-time conference in that games tend to be competitive rather than cooperative.

Chapter Two

Example: Joint Document Editing

This chapter illustrates the design of real-time conferences by developing the specification of an example real-time conferencing system. We have chosen one of the most common uses of interactive computing, namely text editing [3], as the application for this example system. This system, which we shall call *JEDI* (for *Joint EDItor*), is meant to support groups of people working together to produce and maintain documents of any of several different kinds, e.g., a technical article or research proposal with two or more co-authors, an annual progress report of a project team, or product documentation written by the designers of the product in cooperation with a professional technical writer.

While holding a conference using JEDI, each participant will be presented with a *shared space* and a *private space* on his workstation screen; the shared space shows information (from one or more documents) that is identical for all participants, while each participant's private space shows information visible only to himself. (We will describe the appearance of the shared and private spaces in more detail later in this chapter.) While editing one or more documents in a conference, the participants will conduct a voice conversation to discuss the documents being worked on.

Most document editing by users of the system will be conducted in individual editing sessions; a real-time conference will be held only occasionally when the need arises. For example, after working asynchronously over a period of time to produce drafts of different sections of a large document, a group of co-authors may wish to hold a real-time conference in order to put the sections together, filling in transitional paragraphs and making other changes so that the sections flow smoothly. Or, after a draft of a document has been read by its co-authors and editors and reviewers, each of whom has entered some comments and possibly proposed alternative sentences or paragraphs, a real-time conference may be held to clarify the comments and to resolve outstanding differences. A conference will often be held only after some planning and scheduling by the users involved, but may sometimes be held in a more impromptu manner, e.g., a user calls up one or more other users on the phone, or

using the system's communication facilities, and suggests holding a real-time conference right away.

Because real-time conferences are not held in isolation, but in the larger context of many asynchronous editing sessions by individual users, it is important that identical or similar command interfaces be presented to a user when working alone and when in a conference. Before describing the real-time joint document editing interface, therefore, we first describe in detail how documents are edited by individual users. This will provide a basis for describing the real-time conference interface, and for showing how certain commands need to be modified, and new ones included, for a real-time conference. First, Section 2.1 briefly describes the overall system environment as it is perceived by the users. Section 2.2 then describes document editing by individual users, and Section 2.3 describes joint document editing in a real-time conference.

2.1 System Environment

This section describes the overall system as it appears to the users, and the style of user interface that we are assuming.

2.1.1 Communications and Storage

We assume a distributed system consisting of users' *workstation* machines and various *server* machines, connected together by some kind of communication network. The network is largely transparent to the users, i.e., except for differences in response time (which may be unbounded, e.g., when a remote machine crashes), a user need not be aware of the physical separation and location of different machines. We list here some important features that are found in many contemporary distributed computing environments; not every system will actually have every feature, and some systems may have certain features in a more advanced form than described here.

The system maintains a collection of long-lived *objects*, such as documents, circuit designs, and calendars, which reside on one or more server machines. *Directories* of objects are used to provide access to objects based on their "filenames" or in response to "queries" on various other properties, such as:

- the type of an object;
- the user who created or updated an object;
- the date and time of creation or update;
- scheduled date and time (e.g., for appointments and project deadlines).

Each object has a *history* consisting of the sequence of *versions* that it goes through as it is updated. Particular versions may or may not be explicitly stored in the system, depending on record-keeping requirements and the cost and availability of storage. Versions of an object may be *volatile* or *permanent*. For example, a common way of updating an object, such as a document, is for a program (e.g., an "editor") to first make a volatile copy of the current version of the object. The user then makes changes to this volatile copy, using an interface provided by the editor program, occasionally requesting that the volatile copy be saved permanently. We shall call each such saved version a *checkpoint*. Checkpoints may sometimes be made automatically by the editor program, without the user explicitly requesting them, e.g., every few minutes or every N update commands. (This is often referred to as "auto-saving" [111].)

In addition to checkpointed versions, the sequence of individual changes between two checkpoints, or between the last checkpoint and the current volatile version, may also be saved in an object called a *log*. Logs allow finer reviewing of the history of the object, and allow any arbitrary sequence of updates to be "undone", with a corresponding cost in storage for the log. In addition, if a machine crashes before an object being edited has been checkpointed, much of the lost work can be recovered if a log of updates has been saved.

Each object has associated *access control* information describing who can perform what operations, such as read and update, on the object. Various forms of protection against *concurrent update* of an object may also be provided. We will describe these features in more detail as we develop the document editing example.

User-to-user communication is supported in the form of *messages*, typically but not necessarily consisting of text, that are sent to users' *mailboxes*. Various forms of *implicit* communication may also be supported, in which a user is informed when an object of special interest to him has been updated by some other user. A user checks his mailbox and reads new messages at his own leisure, but he may wish to be interrupted, by a real-time *notification* appearing on his screen, on the occurrence of events deemed important, e.g., the arrival of

new mail, or a particular object updated, or a real-time conference called, or an "alarm" set to go off at a designated time. Each user controls what kinds of events should interrupt him and when.

2.1.2 Workstation Interface

The physical display *screen* of a user's workstation is assumed to be multiplexed among one or more *viewports*. Each viewport presents an *image* that is derived from an object (such as a document) or a collection of objects in some specified way. The image in a viewport is generated and updated by a program, such as a text editor, that is associated with the viewport.

We shall assume that the workstation has the following user input devices:

- A *keyboard*, which has keys for all the characters in a given character set (e.g., ASCII or EBCDIC) and some additional specially-labeled *function keys*. (We will make many references to function keys in this discussion; for a keyboard that does not have function keys, or only has a few, we will assume that some "control" character from the given character set is used instead.)
- A *mouse* with some small number of buttons, say two or three. A *pointer*, which is a distinctive small pattern, is superimposed on the display screen at some location, and the location of the pointer on the screen moves in proportion when the user moves the mouse over the surface, e.g., the user's desk, on which the mouse is resting.

Each user action on an input device, i.e., a keystroke, or a mouse movement and/or button push, updates a corresponding object in the workstation's memory: an input event *stream* for keystrokes and button pushes, and a mouse position register for movements of the mouse. Updates to objects, such as the above, can cause actions to be invoked that may update other objects, and so on. Thus, a typical action sequence may be: the user updates an input object; some application object, e.g., a document, is updated; a viewport that displays the object in question is updated in turn.

At any given time, one of the viewports on the screen is the *current* viewport, and user input actions are interpreted as *commands* to the program associated with this viewport, in any of several ways:

- *Character* commands, where a single keystroke invokes a command.
- *Button* commands invoked by depressing a mouse button. The arguments to the command, and sometimes even which command gets invoked, will in general

depend on the position of the mouse at the time the button is clicked.

- A command may be *selected* from a *menu*, by pointing at the command name in the menu and clicking a mouse button. The menu, of currently meaningful commands, is displayed only when the user requests it by typing the MENU function key. The menu appears in a *pop-up* viewport that temporarily obscures some of the information displayed on the screen; the menu disappears and the obscured information is redisplayed after the user selects a command or types CANCEL.

- After typing the COMMAND function key, the user can type the *name* of a command, in full or using a prefix that is sufficiently long to distinguish it from all other command names, followed by any necessary arguments to the command. Characters typed are collected in a command line that is "echoed" at some appropriate position within the current viewport. The ENTER key completes the command, causing the command line to be parsed and executed; the CANCEL key can be used instead to discard the command line.

Which of the above is used for a given command in a viewport is determined by the associated program; it will often be possible to invoke a given command in more than one way. Once a command has been entered and processed, its effect will be displayed in the viewport in the form of an update to a displayed object, such as a document, and/or some feedback such as highlighting or an error message.

Some small set of user input actions is reserved for the following commands to the workstation *executive* which has overall control over the screen and viewports: designating a given viewport to be the current one, creating a new viewport with some associated program, destroying a viewport, and moving or changing the size of a viewport.

2.2 Editing by Individual Users

To establish a proper context for joint document editing in a real-time conference, we first describe a document structure and single-user editing interface that has many of the features of contemporary text editors. For simplicity, this example editor does not include some of the more advanced features that are becoming available in interactive document editing systems, such as multiple fonts, multiple component types with different formatting parameters, pagination, floating figures and footnotes, embedded graphics, and so on. This will allow us to concentrate, in Section 2.3, on the novel issues that are raised by real-time conferencing rather than on the details of particular document editing facilities.

2.2.1 Document Structure and Operations

We describe here the underlying data structure and system-level operations on documents. (User commands, which are translated in to these underlying operations by the text editor program, are described in later subsections.) A two-dimensional "quarter-plane" model of the structure of a document is assumed, similar to that of Meyrowitz and van Dam [83]. That is, a *document* consists of a sequence of one or more *lines*, where each line is a variable- and unlimited-length sequence of *characters*. A line with no characters in it will be called a *blank* line. An *empty* document has a single line which is blank.

For many of the editing operations, it is also convenient to view the contents of a document as a one-dimensional *string* consisting of the lines of the document concatenated together with special *end-of-line*, or *EOL*, characters separating consecutive lines. There is no EOL before the first line or after the last line in this model. (A user may sometimes find it convenient to end a document with a blank line, in which case the last character in the document content string will be the EOL that follows the next-to-last line.) An empty document, with a single blank line, has an empty content string.

Every document has a unique *name*, which is a user-supplied character string. The document name may in many systems have some internal structure, e.g., may include a "path" through a directory hierarchy, and/or a suffix identifying the document's "type"; the details of these naming conventions vary from system to system and are not important here. The operation **Create-Document(name)** creates, and returns, an empty document (one blank line) with the given name. The name must be unique, i.e., not already used.

Operations on documents, such as insertion and deletion described below, are performed using *cursors*. A cursor is a kind of "pointer" that is logically positioned between two characters of a document content string, or before the first character or after the last character. A document may have any number of associated cursors, which may be moved explicitly or updated as a side-effect of operations on the document as described below. A cursor is considered to be *in* a given line of the document if it appears after the EOL (or the beginning of the document) that precedes that line and before the EOL (or the end of the document) that follows the line. The operation **Create-Cursor(document)** creates and returns a cursor that points to the beginning of the given document, i.e., before the first character if there is one. A cursor can be moved using the following operations (additional cursor movement operations will be described presently):

`Move-Cursor-Absolute(document,cursor,line,offset)`:
: Moves the given cursor to the specified position of the specified line. The offset must be between zero, in which case the cursor is positioned before the first character of the line, and the length of the given line.

`Move-Cursor-Relative(document,cursor,number)`:
: Moves the cursor forward by the specified number of characters. The number may be negative, in which case the cursor moves backward (or zero, in which case the cursor does not move.)

`Move-To-Cursor(document,cursor,another-cursor)`:
: Moves the first cursor to be coincident with the second cursor. (The two cursors will subsequently move independently of each other.)

A *search pattern* is a regular expression that can match a sequence of characters in the contents of a document.[5] The following patterns will match single characters as specified:

- An alphabetic character will match the same character if it appears in the document in upper or lower case. Case-insensitive search is the default; if a given character must match exactly including case, it can be "quoted" using **** below.
- **$** matches the end-of-line character.
- A \\ preceding any character will match exactly that character; this is used to match characters that otherwise have special meaning.
- **?** matches any character.
- `[c1-c2]` matches any character in the range *c1* through *c2*, e.g., `[0-9]` or `[A-Z]`.
- Any character other than the special characters defined here will match exactly that character only.

Patterns can be combined in the following ways:

- *Concatenation*: *p1p2...pN* will match anything that matches *p1* immediately followed by anything that matches *p2* and so on through *pN*. A simple example of this is a string of individual characters (e.g., `the`), although more complex patterns can be concatenated.
- *Repetition*: *p** will match as many consecutive occurrences as possible, zero or more, of the pattern *p*. The pattern *p+* will match one or more occurrences of the pattern *p*.
- *Alternatives*: **{***p1p2...pN***}** will match anything that matches *p1* or *p2* or ... or *pN*. For example **{ +$}** will match either several consecutive spaces or an EOL.

---

[5] The syntax we present for search patterns is derived from the text processing facilities available on UNIX™ [67].

Patterns can be used for simple string and word searches, case-sensitive or case-independent. In addition, patterns can be used to delimit the boundaries of logical units of text, such as:

- The end of a "paragraph" is delimited by one or more blank lines, i.e., two or more consecutive EOLs: **$$+**.
- The pattern **{.!\?} ***, i.e., a sentence-terminator followed by any number of spaces, delimits the end of a "sentence".
- The end of a "word" is delimited by any punctuation character or space followed by zero or more spaces: **{ ,.!\?} ***.

The above patterns will be used for supporting user operations on logical units of text, as described later.

Patterns can be used to move a cursor within a document using two basic operations:

Move-End-Pattern(document,cursor,origin-cursor,pattern,number):
: Moves the first argument **cursor** to the end of the **number**-th occurrence of the given **pattern** following the current position of **origin-cursor**. **Cursor** and **origin-cursor** may or may not be the same cursor. A cursor can thus be moved by searching relative to its own current position, or can be positioned relative to some other cursor which does not move. If there are fewer than **number** occurrences of **pattern** following **origin-cursor**, **cursor** moves to the end of the document. Zero and negative values of **number** are used to move to occurrences of **pattern** that precede **origin-cursor**: zero moves to the end of the immediately-preceding occurrence, -1 moves to the end of the one preceding that, and so on until the beginning of the document is reached.

Move-Start-Pattern(document,cursor,origin-cursor,pattern,number):
: This is similar to Move-End-Pattern, except that **cursor** is moved to the beginning of the given occurrence of **pattern**.

The contents of a document may be modified, using a cursor, with the following operations:

Insert-String(document,cursor,string):
: Inserts the given text string at the position specified by the cursor. The cursor is advanced to point *after* the inserted text. (Other cursors that may have been coincident with this cursor do not move.) The inserted string may contain EOLs, in which case the current line (in which the cursor is located) is split and additional lines added to the document.

Delete-Region(document,cursor,another-cursor) returns(string):
: The text between the two cursors is deleted from the document. Both cursors, as well as any other cursors that may have been located in the

deleted region, now point between the character just preceding the deletion and the character just after it. The deleted text is returned by this operation; it may later be used in an Insert-String operation on the same or different document, for example.

Delete-Characters(cursor,number) returns(string):
Deletes, and returns, **number** characters forward from the given cursor position, no farther than the end of the document. Negative values of **number** delete characters backward from the given cursor.

Format-Region(document,cursor,another-cursor,line-width):
Causes the region of text between the two cursors to be "formatted" according to the given line-width. The formatting algorithm is fairly straightforward: lines are "filled" to contain as many words as possible without exceeding **line-width**; a word that is longer than **line-width** appears all by itself on an oversize line; lines are not filled across paragraphs (which are delimited by blank lines).

The operation **Copy-Region(document,cursor,another-cursor)** does not modify the document, but simply returns the contents of the region between the two cursors. Again, the returned string may be used in a later Insert-String operation.

2.2.2 Document Windows

```
+----------------------------------------+
|  Thesis.draft v#17     MODIFIED        |
+-------------------------------------+--+
|  It was the best of times and it was | |
|the worst of times.                  | |
|                                     | |
|  A paragraph with a long line.      | |
|The quick brown fox jumps over the laz|*|
|                                     | |
|***END***                            | |
|                                     | |
+-------------------------------------+--+
```

Figure 2-1:A Document Window

A *document window* is a rectangular subset of the quarter-plane of the document, defined by a *starting-line*, a *height* (expressed as the number of lines), a *starting-column*, and a *width*. Windows are displayed on the workstation with some additional information, as shown in Figure 2-1. An extra line at the top of the window is used for displaying the document name and version (described later) and whether the user has modified the document. An extra column or "margin" on the right indicates whether any of the lines displayed in the window

have additional characters past the last displayed column. If the starting-column of the window is not column one, then all lines displayed have additional text to the left; when this is the case, it is indicated in the top line. (The top and right margins are also used for horizontal and vertical "scrolling" as described below.) In addition, if the window extends past the last (or first) line of the document, that is indicated within the window as shown. Some of the text displayed in a window may be specially highlighted, e.g., to indicate the position of the editing cursor and the selected region, described presently.

2.2.3 Editor Interface

The text editor program uses its screen viewport, assigned to it by the workstation "executive" when the editor is invoked, to display a *super-window* that is a collection of non-overlapping windows. For simplicity, we shall assume that the windows are stacked vertically and all have the same width. (More complex arrangements are possible, including side-by-side placement of windows, requiring additional commands for window manipulation.) Initially the editor super-window is occupied entirely by a single *blank* window that has no document assigned to it; documents may be assigned to windows, and windows created and removed, using window manipulation commands described presently.

The text editor maintains the following invisible *state variables* that are used and updated by user commands in a manner that will be described presently:

- A *Paste-Buffer*, that contains the text string, possibly including EOL characters, returned by the last delete or copy command.
- The *Search-Pattern* last used in a SEARCH or SELECT command (described below).
- A *Line-Width*, used for automatic formatting.
- A *Working-Set* of documents being edited. The Working-Set includes every document currently displayed in a window, and may also include documents not currently displayed; a given document may be displayed in more than one window.

Each non-blank window has two associated cursors into the document that it displays. The *editing cursor* is the position at which insertions by the user take place; we will for brevity call this "the cursor". The second cursor is called the *mark*. The mark and the cursor together delineate the region to which deletion and other commands, described below, apply when the user is working in the given window.

At any given time, one of the document windows in the editor viewport is the *current* window. The region between the mark and cursor of the current window is called the *currently selected region*; this region, or as much of it as is visible in the active window, is specially highlighted, e.g., using reverse-video or underlining. We first describe how scrolling and editing operations are invoked on the current window, and then describe how windows are manipulated and the current window selected.

2.2.3.1 Scrolling a Window

Scrolling of a window is the means by which a different region of the given document is displayed in the window; the document contents are not changed. *Vertical* scrolling is accomplished by moving the mouse to some location in the window's right margin and depressing one of two mouse buttons that will do the following:

- Downward scrolling: the window is scrolled such that the document line currently alongside the mouse position appears at the top of the window.[6]
- Upward scrolling: the window is scrolled such that the line currently at the top of the window appears alongside the mouse position.

The vertical distance of the mouse from the top of the window thus determines the number of lines by which the window is scrolled in either direction. If the mouse is at the bottom of the window margin, the window is scrolled by the number of lines in the window less one. If the user presses the wrong button and scrolls in the direction opposite to the one he intends, simply pressing the other button without moving the mouse restores the window to its original position.

Horizontal scrolling, which is only needed if some lines in the document are longer than the width of the window, is accomplished in an identical manner, except that the mouse must be positioned in the top margin of the window.

Note that scrolling of a window does not move the window's editing cursor or mark; it is possible that either or both might not be visible when the user scrolls. If the user performs an insertion or deletion operation, below, while the cursor is not visible, the window is automatically scrolled so as to make the cursor, and the effects of the operation, visible. If the

---

[6]Our interpretation of "upward" and "downward" scrolling is based on a model of the window moving over the document. The opposite model, of the document moving under the window, is sometimes used and is equally valid [12].

cursor is not currently visible and the user wishes to see it without performing an editing operation, he can click a mouse button to invoke the *Mouse-to-Cursor* command; this scrolls the window as necessary to display the cursor at approximately the same location on the screen as the mouse. For this command to be invoked, the mouse must be "inside" the window, not in the margins which are used for scrolling. (If the user instead wants the cursor to move to a location that he has reached by scrolling, he points at the desired location with the mouse and clicks another mouse button to invoke *Cursor-to-Mouse*; this is described in more detail below under cursor movement.)

2.2.3.2 Editing Within a Window

The editing operations below cause the contents of the document displayed in the current window to change as specified at the location of the cursor. The current window, as well as any other windows into the same document that show some or all of the affected text, is updated to reflect the change.

The simplest editing operation is the insertion of a single character: Simply typing a printable character causes Insert-String to be invoked with the editing cursor and a single-character string as arguments. The cursor is advanced to point after the inserted character. (No other cursor into the given document is moved.) The RETURN key is used to insert an EOL character, causing the current line of the cursor to be split into two lines at the location of the cursor. (Either of the resulting lines may be empty, if the cursor is located at the beginning or end of a line.) Since the SPACE key is used for formatting a line, as described below, a separate control or function key, which we shall refer to as Quoted-Space, is used to insert a space character without causing any formatting.

Two function keys, RUBOUT and ERASE-FORWARD, are used to delete the character just before and just after the cursor, respectively. The deleted character, which may be an EOL, is discarded by the editor.

The DELETE function key deletes the currently selected region, between the mark and the cursor, and saves the deleted text in the Paste-Buffer. The previous contents of the Paste-Buffer are lost. (They may still be retrievable, albeit in a roundabout way, using "history" commands described presently.) The mark and cursor are now coincident, i.e., the selected region is now empty.

The COPY function key copies the text in the currently selected region into the Paste-Buffer without deleting it from the document.

The PASTE function key inserts the contents of the paste buffer at the current cursor position. The cursor now points at the end of the inserted text, and the mark at the beginning; the inserted text is thus automatically "selected" (and highlighted in the window). Note that the contents of the paste buffer may have been obtained from a document other than the one currently being edited, if the last delete or copy command was executed in a different window; text can thus be transferred between documents in a "cut and paste" fashion.

The FORMAT function key reformats the currently selected region by invoking the Format-Region operation with the current region and the current value of the Line-Width state variable as arguments. (The Line-Width, whose initial default value is the same as the width of the editor viewport excluding margins, can be changed by a separate command.) Typing the SPACE bar inserts a space character and then causes the current line of the cursor to be reformatted. The user can thus type in an entire paragraph without having to use the RETURN key to break lines; the editor automatically formats the text as it is typed.

2.2.3.3 Cursor Movement and Selection

The user can change the position of his editing cursor using the following commands. All of these commands scroll the window as necessary to keep the new cursor position visible.

"Arrow" function keys are used to move the cursor one position forward, backward, up, or down. Forward and backward movement wraps around EOL characters. Upward and downward movements keep the cursor at the same column position in the previous or next line, unless the line in question is shorter in which case the cursor moves to the end of that line.

When the mouse is in the main text region of the window (as opposed to the margins), a button invokes Cursor-to-Mouse, which moves the cursor to the location corresponding to the mouse position. (This requires a reverse transformation from display coordinates to logical positions in a document, and may involve approximation to find the "closest" logical location. For example, if the mouse is in column 30 of a 20-character line, the cursor will move only to the end of that line, i.e., column 20.)

The EXCHANGE function key interchanges the locations of the mark and cursor; the same

region as before remains selected.

The SEARCH function key initiates a pattern-search as follows. The user is prompted by the word "SEARCH" being echoed at the top of the window; the rest of the command is also echoed as typed, and the user can at any time CANCEL the command. The user may optionally type START-OF or END-OF, with END-OF assumed as default if neither is typed, followed by an optional positive or negative number, which defaults to 1 if not typed. The command is then completed by the user specifying a pattern (using the syntax presented in Section 2.2.1), at which point the Move-Start-Pattern or Move-End-Pattern operation is invoked to move the cursor over the specified number of occurrences of the pattern. The pattern itself is specified in either of two ways:

1. A *built-in* pattern is associated with each of the function keys WORD, SENTENCE, PARAGRAPH, DOCUMENT, LINE, and CHARACTER.
2. Any other pattern can be specified by typing the characters of the pattern, followed by the ENTER key. If ENTER is typed without specifying a pattern, the Search-Pattern state variable, set by the previous SEARCH command (or SELECT, below) is used.

The mark stays fixed when the cursor is moved by the above operations; the selected region implicitly changes as the cursor moves. (The only exception to this is EXCHANGE, which moves both the mark and cursor but keeps the region fixed.) The commands presented next are used to change the selected region by moving the mark and keeping the cursor fixed.

MARK-CURSOR moves the mark to the current location of the cursor. The selected region is now empty, but will grow if the cursor is moved and the mark stays fixed, e.g., if characters are typed.

The SELECT function key moves the mark over a given number of occurrences of a specified pattern, relative to the current position of the cursor (not of the mark). The syntax of this command is identical to that for SEARCH, above.

The use of the above commands can be illustrated by means of an example. If the user types SELECT PARAGRAPH, the mark will be positioned after the end of the blank line(s) terminating the paragraph in which the cursor is currently located, i.e., just before the beginning of the next paragraph (or at the end of the document). The text between the cursor and the end of the paragraph is thus selected, and can be deleted or copied, or formatted, if

desired. If instead the user wishes to delete or copy or format the entire paragraph, he can type SELECT PARAGRAPH as above to position the mark, and then SEARCH 0 PARAGRAPH to position the cursor at the beginning of the paragraph.[7] The user in either case may instead use the mouse and Cursor-to-Mouse if he wishes.

2.2.3.4 Window Manipulation

The commands described above operate on the user's *current window*. The editor viewport may contain other windows as well, with the same or different documents, and the user can designate a different window to be current in either of the following ways:

- Using the *Next-Window* and *Previous-Window* commands, which select the next window up or down, respectively, from the current window. These commands "wrap" around from the topmost to the bottommost window, or vice versa.
- *Implicitly* by positioning the mouse in a window other than the current one and issuing a mouse button command to scroll the window or to move the window's cursor to the mouse position or vice versa.

The current window can be made larger, at the expense of adjacent windows but no farther than the boundaries of the editor viewport, using the *Grow-Top* and *Grow-Bottom* commands. (The editor viewport itself may be grown or shrunk, or repositioned on the physical screen, by an interaction with the workstation executive; this is not discussed here.) These commands may cause an adjacent window, above or below respectively, to disappear completely. If the document in such a window is no longer visible in any other window, the cursor and mark are saved in an invisible window of zero size; this allows the cursor and mark positions to be restored when the same document is later displayed, using Edit-Document below. (Note that there is no need for a command to remove a window, or to shrink a window, because these can be accomplished by selecting an adjacent window and growing it as desired.)

The current window can at any time be split into two approximately equal-sized windows using the *Split-Window* command. After the split, the upper window displays the same document as the window before splitting, while the lower window is *blank*, i.e., has no

---

[7] We have arbitrarily chosen an "object-verb" command style, of selection followed by an operation such as delete or copy. A valid alternative is a "verb-object" syntax, such as Etude's DELETE 3 PARAGRAPH [48]. The verb-object style allows a "selection" and an operation to be performed all at once with fewer user keystrokes. The object-verb style on the other hand provides feedback about what the user is about to delete, by always highlighting the currently selected region, before he performs the deletion.

associated document. The new blank window becomes the current window, but no editing can be performed in it until an Edit-Document command, described below, is performed. A user's text editing session starts with a single *blank* window occupying the entire viewport assigned to the editor; the user can then select documents for editing and create windows within the viewport as desired.

The Edit-Document command causes the current window to display a different document; the document is specified in either of two ways:

1. By selecting from a menu that displays the *Working-Set* of documents being edited.
2. By typing the *name* of the document. If the named document is not already in Working-Set, it is added to it. In addition, if a document with the given name does not already exist, an empty document is created.

The Working-Set is initially empty when the editor is invoked, and grows as Edit-Document commands are invoked. The editor maintains its own volatile copy of every document in Working-Set. Each copy is at any given time either *saved*, indicating that the current contents have been saved in a permanent version or *checkpoint* of the document, or *dirty*, indicating that the copy has been updated since the last checkpoint. Dirty document copies are checkpointed periodically by the editor, and whenever the user invokes the Save-Document command which checkpoints the document displayed in the current window. An additional command, Save-All-Documents, checkpoints all dirty document copies; this is also invoked automatically when terminating the editor program.

The Quit-Edit command dissociates the current window from the document it displays; the window becomes blank. If the document is not displayed in any other window, the editor's volatile copy of the document is discarded and the document is removed from Working-Set. (If the document copy is "dirty" when this happens, the user is asked whether he first wishes to checkpoint it.) A list of documents currently in Working-Set, together with flags indicating which are dirty, can be called up by the user in a pop-up viewport; the user can invoke Edit-Document, Save-Document and Quit-Edit by pointing at this list and using the mouse buttons.

2.2.4 History-Related Commands

A window may be used to review one or more past versions of the document it displays using the HISTORY command. The title line of the window shows the word "History" together with an identification (e.g., version number or "timestamp") of which version of the document is currently being viewed. The window is placed in a "read-only" mode that does not allow the document contents to be modified; it does not make sense to change past history. Scrolling and cursor movement and selection, and the COPY command, are permitted in read-only mode. This allows text from a previous document version to be pasted into another window that displays the current version of the same or different document; even the entire previous contents of a document may be copied in this way.

In addition, the following commands are provided for operating on the history:

- Next-Version and Previous-Version, which show the next and previous checkpoint, respectively, from the document's history.
- QUIT viewing the document history, and restore the window to allow editing of the current version of the document.
- UNDO, which makes the currently displayed version the new version of the document.
- MERGE the contents of the currently displayed checkpoint with the current version of the document to produce a new version; this uses the procedure described below for merging "alternatives".

If the user wishes to see both the current version and the history of the document at the same time, he can do so in different windows.

2.2.5 Access Control

Every document has an *owner* who is the user who originally created it, i.e., wrote the first version associated with the given document's name. The document is initially accessible only to its owner, but the owner can at any time change the specification of which users can *update* the document, i.e., write a new version, and which users can *read* the document, i.e., view its contents. Each of these is specified as an *access control list* of user names, or as the special value *public* which gives all users the given privilege, to read or update. Users with update access are implicitly given read access as well. The owner retains the sole ability to erase past versions of the document, or to erase all versions and delete the document altogether; he may also instruct the system to automatically "garbage-collect" old versions

based on some criteria such as age or an upper limit on the number of recent versions to retain.

2.2.6 Concurrency Control

Giving multiple users update access to a document introduces the familiar problem of concurrent update. In the absence of any controls, described below, the following may happen. Users A and B both execute Edit-Document commands on the same document X, say. Their editors now have separate volatile copies of X which are updated by A and B independently of each other. Each user then checkpoints his updated copy of the document. If A, say, is the first to write out a new version, then the new version that B checkpoints will include none of A's changes; A may further update his copy and write out a new version that does not have any of B's changes, and so on. Even though neither users changes are lost when multiple versions are retained, any other user who reads the latest version of the document will see only one user's changes, A's or B's, and will not be aware of the others'.

Two levels of protection are provided against the above kind of problem. The first, *validation*, works as follows. A user's editor, on executing an Edit-Document command, is supplied with the version number of the document version that it reads into its volatile copy. When the editor wishes to checkpoint its updated copy as a new version of the document, on a user command or after an "auto-save" interval, the attempt to write fails if the version that was read is no longer the current version of the document, i.e., some other user wrote a new version in the meantime. If the attempt to write succeeds, the editor remembers the version number of the new document version that it wrote, so as to validate this the next time it tries to write a new version.

This validation using version numbers will avoid the problem described above. If A and B read the same current version of a document, and A is the first to write out a new version, then B will be unable to write a new version because the version he read is no longer current. B's work is not necessarily lost; this is discussed below under "alternatives".

A second level of protection is the setting of *reservations* in an attempt to forestall the possibility of failing to write a new version because of an intervening concurrent update. The user's editor, on executing an Edit-Document operation, can set a reservation that prevents

other users from writing new versions for some specified period of time, say a minute or two.[8] If the user does not perform a Save-Document command within the given time limit, his editor will automatically do it for him. When a new version of the document is to be written, validation of the version number is still performed; the write succeeds only if the document version read is still current *and* no other user currently holds a reservation on the document. This allows the write to succeed even if it was attempted after the reservation time limit expired, provided no other user has updated or reserved the document in the meantime. On successfully writing a new version, the editor tries to reacquire the reservation for further updates; this may fail if another user has been granted the reservation.

If a user A, say, tries to acquire a reservation on a document that some other user, say B, already has reserved, the following happens. A is informed that B is currently updating the document, and for the moment is shown the current document version (i.e., the last checkpoint, not B's dirty version which is visible only to B), in *read-only* mode. B in turn is informed that A has requested the reservation, and is warned that he will lose his reservation when its time limit expires. This gives B a chance to record his changes in a new version, at which point A receives the reservation and the new document version, which he may proceed to edit. Since only one user at a time can hold the reservation on a document, and the holder of the reservation always sees the most recent changes made by the previous holder of the reservation, the above problem of lost concurrent updates cannot arise. A may of course choose to undo changes made by B, but will only do so with full knowledge of what he is doing, rather than accidentally overwriting changes that he has not seen.

While A is editing the document above, B now sees the document in read-only mode. B may in turn request the reservation, at which point A is informed, and so on. This process may go on indefinitely, although in practice one user or the other might choose to bow out and let the other proceed. Alternatively, users who find each other trying to update the same document concurrently might choose to work together in a real-time conference, described presently, and view each others' changes in real time.

If a user does not save his changes by the time he loses the document reservation to

---

[8] A reservation is in some ways similar to a "lock" in database systems, but differs in that it is more volatile and may be revoked by the system, e.g., when the time period expires or if the document owner revokes this user's permission to update. A reservation is also held for longer than a lock; in our system, we assume that true "locks" on the document will be set only briefly, invisible to the user, while actually writing a new version.

another user, his works is not lost. Instead, his copy of the document, if dirty, is saved as an *alternative* version. An alternative document version has an associated *origin* version number, which is the version number of the original document that was read by the user creating the alternative. An alternative can be further updated by its creator, and a sequence of alternatives saved to form an alternative "branch" of the document history. There may be more than one such alternative branch, generated by different users, as shown in Figure 2-2.

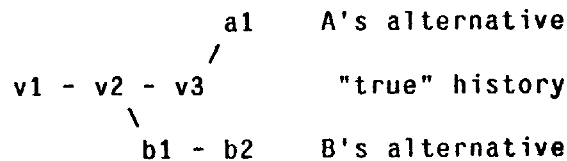

Figure 2-2: Alternative Document Versions

The user generating an alternative branch may discard individual versions in that branch or the entire branch. More likely, the user at some point may wish to "install" his changes into a new version in the document's true history. If the origin version number on which the alternative was based is still current (true for **a1** in Figure 2-2, but not for **b2**) and no other user has the document reserved, the alternative is successfully written as a new document version. If not, the user's editor will read the latest true version of the document, acquiring the reservation if possible, and display this document version to the user. The user may then edit this version, copying over some or all of the changes he made in his alternative. Or, the user may instruct his editor to automatically *merge* the latest document version and his alternative. The editor does this as best as it can using a document comparison procedure, asking the user to make a decision or perform some editing whenever incompatible changes (e.g., to the same line or paragraph) appear in the two versions. (Certain "minor" changes are ignored by the merge procedure: additional or missing blank lines; words that differ only in case; and paragraphs that differ only in the position of the line-breaks between words.) The user then saves a new version, as already described, when satisfied with the combined changes.

The above support for "alternatives" allows users to continue working in the presence of communication failures or "partitions" that may make it impossible for a user's workstation to acquire a reservation or write a new version. Or, a user wishing to explore an alternative development path without interfering with other users can do so by deliberately not acquiring

the reservation, with the intention of later merging (or discarding) his alternative.

2.3 Real-Time Conference Interface

We are now in a position to describe JEDI, our real-time joint document editing system. We first describe the joint document editing interface as it appears in a conference that has already been set up with some set of participants; the process of starting a conference and adding participants is described later.

2.3.1 Windows and Reservations

Each participant in a real-time conference using JEDI sees two collections of document editing windows, or "superwindows", on his screen: a *shared space* and a *private space*. A participant may place the private and shared spaces on his screen in any juxtaposition he chooses, either one above the other or side-by-side. This is illustrated in Figure 2-3; a more detailed picture of the contents of the shared space will be shown below. A participant may in addition have other viewports unrelated to the conference on his screen, although it is unlikely that there will often be enough screen space left over. A participant with a small screen may have to overlay one space with the other, with commands provided for switching between the two, i.e., making the obscured space, shared or private, fully visible and obscuring the other one. Or, a participant with two physical screens at his workstation may place the shared space in one screen and the private space in the other.

Each participant has a *pointer* that he can move over the shared space using his mouse. The participants' pointer positions are visible to all participants in the shared space, with each pointer having a different "shape" (i.e., the pixel-pattern that is superimposed on the displayed information) so that different participants' pointers can be distinguished. In addition, each participant sees the position of his pointer on his screen with a special shape that he can easily distinguish as his "own". This is also illustrated in Figure 2-3. A participant can move his pointer off the shared space, making it no longer visible to the others. (A participant's mouse position will still be visible to the participant himself, e.g., he may use it for commands in his private space.) Note that the participants' pointer positions are defined with respect to the image displayed in the shared space, and are not affected by scrolling, editing, or window manipulation operations that change what is displayed in the shared space.

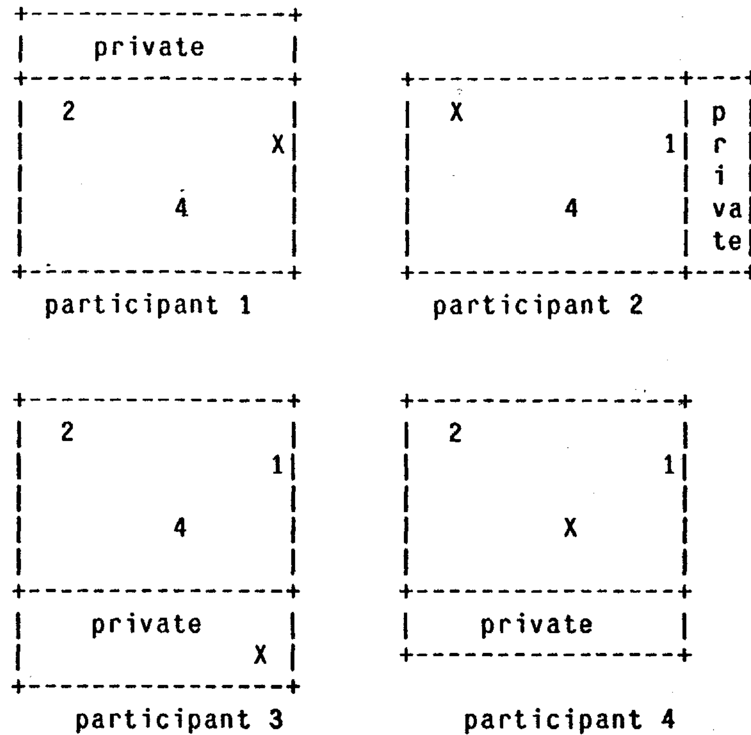

Figure 2-3:Shared and Private Spaces

In addition to document windows, the shared superwindow has a small number of lines at the top occupied by a conference *status window* that shows who is in the conference and other useful information about participants that will be described presently. This is shown in Figure 2-4.

Each participant's private space has a Working-Set of documents being edited privately, a Paste-Buffer, and a Line-Width for formatting; the participant can work in this private space in the same way as he would when not in a conference, using the single-user editing interface described earlier. The shared space has its own Working-Set of documents, and a Line-Width, but no paste buffer for reasons we describe presently. How the shared Working-Set is updated (from its initial empty state), and what happens when it overlaps one or more participants' private Working-Sets, is described presently.

Participant activity in the shared space is controlled using the following kinds of *reservations*:

- A *window* reservation on each window in the shared space, which allows a

```
+-------------------------------------------------------+
|conference "Annual Report"   2-Jan-84 11:37            |
|MERRILL  LYNCH  PIERCE  FENNER  SMITH  JONES           |
| Req#1   Chair            Left          Waiting        |
+-------------------------------------------------------+
|Stocks.mss v#13                        Merrill         |
|  Blue chips rose 10.3% this year, reflecting a trend  |
|that began last year when the Dow Jones hit 1000.      |
+-------------------------------------------------------+
|Bonds.mss v#9                          FREE            |
|  Returns on T-bills were not as promising as          |
|anticipated; we are therefore taking a tax writeoff.   |
|Municipal bonds on the other hand                      |
+-------------------------------------------------------+
|Speculation.mss v#1                    Smith           |
|***TOP***                                              |
|  A good year for pork bellies, soybean futures, and   |
+-------------------------------------------------------+
```

Figure 2-4: Shared Space in JEDI

participant to perform scrolling and editing operations in that window.

- *Region* reservations which isolate participants who may be editing the same document via different windows.
- A *super* reservation on the entire shared space, which allows a participant to perform more global operations involving multiple windows.

(In addition, *document* reservations, described earlier in Section 2.2.6, isolate the conference as a whole against other users who wish to concurrently edit the same documents.) We describe each type of reservation in more detail below.

Each window in the shared space has an associated *window-reservation* which at any given time is either *free* or is assigned to one participant; the name of the participant, if any, holding the reservation, is displayed in the "title" line of the window (along with the document name, as before). Possession of a window-reservation allows a participant to perform the following operations in the window, using the same command interface as a single user:

- Scrolling, cursor movement, and selection using the mark.
- Typing in, erasing, and formatting text.
- Deletion and copying of the currently selected region; these operations cause the selected text to be saved in the participant's *private* paste buffer.
- "Pasting" of the contents of the participant's private paste buffer into the document.

If the participant holding a window reservation does not have update access to the document in that window, he may not modify the document; he may only scroll, select by moving the cursor and mark, and copy into his paste buffer.

The use of private paste buffers for the delete, copy, and paste commands permits transfer of text between shared and private spaces. Thus, a participant may copy or delete text from a shared window, and then select a private window into which he pastes the copied text, or vice versa. The absence of a shared paste buffer also prevents different participants' delete or copy commands from interfering with each other. The only problem with not having a shared paste buffer is that if one participant deletes text from a shared document then another participant cannot get back the text from the first participant's private paste buffer; however, deleted text can still be retrieved from the document history.

In addition to the above, the participant holding a window reservation may perform a limited set of operations on windows and on the shared Working-Set:

- He may *grow* the top or bottom of the window only if the adjacent window, above or below, is not currently reserved by any participant.

- He may *split* the window, acquiring the reservation on the lower, blank, window resulting from the split.

- He may perform an Edit-Document command, specifying a document already in the shared Working-Set or a new document. The latter allows participants to "submit" documents to a conference.

- He may perform a Quit-Edit command if he has update access to the given document. The reserved window becomes blank, and the document is removed from the shared Working-Set if it is not displayed in any other shared window.

Window reservations are acquired and released by participants as follows:

- Any participant may acquire the reservation on a window that is free by using one of the commands normally used for designating the "current" window: moving his pointer into the window and executing Cursor-to-Mouse or Mouse-to-Cursor or scrolling using the margins, or executing Next-Window or Previous-Window from an adjacent window.

- A participant can release his reservation on a shared window at any time.

- A participant can hold a reservation on only one shared window at a time, except when holding the "super-reservation" described below. Thus, a participant holding a reservation on one shared window loses that reservation when he acquires a reservation on a different window.

- A participant automatically loses the reservation on a shared window if he does not perform an operation in that window for some specified period of time, say half a minute. This protects against a participant who goes off into his private space and works there without releasing his shared window reservation, or a participant who simply sits and does nothing or who walks away from his workstation or whose workstation crashes.

- If participant A, say, attempts to acquire a window-reservation currently held by participant B, say, A's command fails and is discarded but B is notified and loses his reservation after a brief time interval. Once the reservation is lost, any participant, A or B or any other, may then acquire it.

Window reservations permit only one participant at a time to operate in each shared window. It is possible, however, for a given document to be displayed in more than one window, and the document may be modified concurrently by the participants holding those window-reservations. Interference among these participants is prevented by providing *region-reservations*: The currently selected region in each window, between the cursor position and the mark, is reserved and no two region reservations are allowed to overlap. Since a participant can only insert at his cursor position and delete between the cursor and mark, concurrent changes by different participants to a given document cannot overlap. (It is not possible, for example, for one participant to inadvertently erase text that another participant inserts into the region that the first participant is about to delete.) An attempt by a participant to move his cursor or mark into another participant's region will generate an error message, and the cursor or mark will not move, i.e., the participant has the same region selected and reserved as before.

The current region in a given window is reserved only if some participant is actually working in that window, i.e., has the window reserved. Thus, a participant working in one window is not kept out of a region in another window that nobody is using. When a participant acquires the reservation on a window that is currently free, an attempt is made to reserve the current region of that window. If this fails, because some other participant has an

overlapping region reserved, the mark in this window is moved to the cursor position, giving an "empty" region reservation. Even an empty region reservation is not allowed to overlap other region reservations, because the participant may insert text at the cursor position and thus modify the other participant's region; this may require moving the cursor to the beginning of the document, outside any regions reserved by other participants. No time limit is set on region reservations; because free windows do not have their regions reserved, the timeout mechanism on window-reservations is sufficient.

The *super-reservation* is a reservation on the entire shared superwindow. No other participants can hold window reservations when a participant holds the super-reservation. The holder of the super-reservation, if any, can perform any editing command or window manipulation operation in any window. He may also change the size of the superwindow, which is described below in Section 2.3.2.

The super-reservation is useful, for part of a conference or for an entire conference, when it is desired to have only one participant at a time working in the shared space, as well as for global actions such as changing the size of the shared space. This is similar to holding the "floor" in a face-to-face meeting or conference. The super-reservation is controlled by the conference *chairperson*, who is designated by the creator of the conference. A participant can *request* the super-reservation, which notifies all participants of the request and also updates the shared status window to show that the request has been *queued*. The status window, as shown in Figure 2-4, shows the current chairperson, the current holder of the super-reservation if any, and any requests that are queued. Operations involving the super-reservation and request queue (requesting, granting, and releasing the super-reservation, and withdrawing a request) are performed by pointing at the appropriate region of the status window and clicking a mouse button. Keyboard alternatives to these operations are also available, e.g., allowing the chairperson to give the super-reservation to the "next" participant in the queue without having to find and point at that specific participant.

The chairperson can at any time grant the super-reservation to any participant, including himself, even if the participant's request is not first in the queue and even if the participant has not requested the super-reservation (for example, the participant may have used the voice channel to ask for the super-reservation). The current holder of the super-reservation loses it, or, if nobody currently holds the super-reservation, every participant loses the window-reservation that he may be holding. The chairperson can also "request" the super-

reservation, which serves as a polite warning to participants that they may soon lose their reservations; the chairperson's request is not queued, since he himself can at any time give or take away the super-reservation.

The participant currently holding the super-reservation may *release* it at any time, at which point the super-reservation is free. Participants may then acquire window-reservations, or the chairperson may choose to acquire the super-reservation or give it to some other participant.

2.3.2 Negotiating a Window Size

The holder of the super-reservation can change the size of the shared superwindow, e.g., he may wish to grow the superwindow to display more information, or shrink it to accommodate a participant with a small screen. Since changing the size will require every participant to adjust his screen accordingly, the participant wishing to make such a change must first go through the following *negotiation* procedure.

Let us assume that the participant wishes to make the super-window larger, for which he invokes the Grow-Superwindow command; the procedure for making the superwindow smaller is similar and is discussed below. The participant, who we shall call the *initiator* of the negotiation, specifies the desired new size, either as a pair of numbers (height and width), or by indicating the desired-size rectangle on the screen using his mouse. Each participant is notified of the desired change in size, and in turn specifies the maximum new size, no larger than the proposed size and no smaller than the current superwindow size, that he is willing to accommodate on his screen. This again may be specified as a pair of numbers or indirectly by specifying a rectangle on the screen.

As participants respond, their offered sizes are processed to generate a list of plausible alternative superwindow sizes, together with which sizes are acceptable to which participants; this list is displayed to all the participants. Suppose, for example, that participant A is the initiator, currently holding the super-reservation, and wishes to grow the superwindow from its current size of 36x48 characters to 50x54. When the negotiation starts, i.e., no replies have yet been received, only two alternatives are presented: the current superwindow size, and the desired new size. By implication, the desired new size is acceptable to the initiator A, who does not have to generate an explicit response.

Size	Participants
36x48	all
50x54	A

As replies come in, the list of alternatives is expanded and updated. Suppose, for example, the other participants respond as follows: B will accept only 44x54, C agrees to 50x54 (the proposed new size), and D 50x48. The list of alternatives will then look like:

Size	Participants
44x48	all
44x54	all but D
50x48	all but B
50x54	A,C

(The set of alternatives may in addition be presented visually by displaying rectangles of the corresponding sizes around the shared superwindow.)

When all participants have responded, the initiator must make a selection from the alternatives presented, by deciding whether it is more important to accommodate as many participants as possible, with a smaller superwindow size, or to present as much information as possible, with a larger superwindow that fewer participants can accommodate. Unless the current superwindow size is retained, the shared superwindow is grown as specified, and each participant adjusts his screen to make room for the larger shared superwindow. If a particular participant was not willing to accommodate the size finally chosen, he can still change his mind and make room on his screen, or he can display only part of the shared superwindow in a smaller viewport that can be scrolled by commands to his workstation. In the latter case, the participant will not always see all of what everybody else can see, and may miss some of the action; he may instead choose to leave the conference.

The initiator of the negotiation may interrupt the procedure at any time and either retain the current superwindow size or select a new one based on only those replies received so far. Thus, he does not have to wait for participants who are slow in responding, or never respond. So long as one or more participants have not responded, the system will prompt the initiator at periodic intervals urging him to make a selection and terminate the negotiation. As a last resort, e.g., the initiator does not make a selection after a long time, the chairperson can intervene at any time by taking away the super-reservation. The negotiation continues if the super-reservation is thus transferred, until some participant terminates the negotiation while holding the super-reservation.

A similar procedure is followed when a participant, holding the super-reservation, uses the Shrink-Superwindow command to make the shared superwindow smaller. Participants now specify the *smallest* size superwindow they are willing to accept, and alternatives are ranked in the opposite order, i.e., larger sizes, beginning with the current size, first. The initiator then selects an alternative as described above.

Negotiation of the shared superwindow size is performed at least once in a conference, at the beginning. In this case, the system performs the above negotiation automatically without bothering the participants by asking for their preferences. Then, additional explicit negotiations are needed only if participants are not satisfied with the default selection made by the system.

If a participant joins the conference after the shared superwindow size has been selected, and his offered size is smaller than the current size, the other participants are informed thereof. The chairperson can take this opportunity, if he wishes, to shrink the superwindow to accommodate the new participant, or even to some in-between size as a compromise. This is a special case that does not involve any negotiation, and the chairperson can do this without acquiring the super-reservation. Whatever the decision, the new participant either adjusts his screen or uses a smaller viewport, as described above.

2.3.3 Document Access Control

Documents are added to the shared Working-Set by users executing Edit-Document commands from a shared window. Any user with read or update access to a document can do this, even if not all participants in the conference have read access; all participants are in effect given "temporary" read access to documents in the shared Working-Set. Update access controls on a document are strictly obeyed: a participant is not allowed to modify a document to which he does not have update access.

A document can be *removed* from the shared Working-Set only by a Quit-Edit command, when the document is not visible in any other shared window. Since this may erase changes made by other participants, Quit-Edit can be invoked only by a participant with update access to the document. (An exception to this is a document that is in "read-only" mode, i.e., not being edited in the conference; any participant can remove such a document from the shared Working-Set.) It is expected that such "retractions" of documents will rarely be needed; in

any case, retraction of a document does not stop other participants from resubmitting the document.

We next consider the possibility of interference between a conference and other users who may be attempting to edit the same document(s), privately or even in another conference. The same method used for protecting individual users is used: The conference, as an active entity in its own right, acquires a reservation on the document, that prevents other users from modifying the document. An individual user can still attempt to acquire the reservation, and will receive it after the time limit on the reservation held by the conference expires. However, the time limit on a reservation held by a conference is set to be much longer than the time limit on a reservation held by an individual user. This has the effect of discouraging users from editing a document that is being used in a conference; the user must either wait a long time, or, more sensibly, try to join the conference if he can. A conference may view one or more documents in "read-only" mode, either on purpose or because some other user has temporarily reserved the document.

A participant in a conference may attempt to privately edit a document that is shared in the conference. Unless he is able to acquire the reservation for himself, which will involve a long wait and will last for only a short time, he will only be able to view the document in read-only mode in his private space. He may freely select and copy text from the document, but his selected region will not be protected against concurrent changes to the document that take place in the shared space.

2.3.4 Access to a Conference

A real-time conference introduces a second level of access control, namely access to the conference itself. Again, the creator or *owner* of the object in question, in this case the conference, specifies the necessary access controls, which consists of:

- A list of the names of those users who will be allowed to join the conference unconditionally.
- A specification of whether the *public*, i.e., all users other than the above, is allowed to join unconditionally, not allowed to join at all, or allowed to submit a *request* to join.

Requests to join are granted or refused by the current *chairperson*. The chairperson is initially the conference owner, but the owner or the current chairperson can designate a new

chairperson at any time. The chairperson also has the power to *remove* a participant from the conference, e.g., one whose behavior is disruptive.

2.3.5 Leaving and Termination

A participant can leave a conference at any time; the other participants are notified and their status windows updated. The leaving participant's copy of the shared space now becomes part of his private editing space. The documents in the shared Working-Set are included in his private Working-Set, allowing him to browse over the information that was discussed and manipulated in the conference. (The participant will have to do this in read-only mode, since the conference presumably still has the documents reserved.) The participant may instead discard his copy of the shared space, making room on his screen for other activities.

A participant leaving a conference may join again, unless the conference has terminated (below) or his permission to join revoked in the meantime. He may leave temporarily, for example, in order to respond to an urgent message or other interruption outside the conference. Or, a participant may "leave" temporarily for the sole purpose of browsing over one or more shared documents in the region of his screen occupied by the shared space.

When the last participant having update access to a given document leaves a conference, the document is automatically checkpointed if "dirty" and the reservation released; the document reverts to read-only mode for the rest of the conference (or until a participant removes it with a Quit-Edit operation). The document may be later updated in the conference if a participant having update access to the document joins and acquires the document reservation for the conference.

When there is only one participant left in a conference and that participant leaves, the conference terminates: No further participants can join or rejoin, and no further activity can take place. A record of the conference is retained so that participants who are trying to join are properly informed that the conference has terminated.

Chapter Three

Design of Real-Time Conferences

Having developed a particular real-time conferencing system in detail in Chapter 2, we now attempt to generalize from that exercise. In the course of designing the joint document editor JEDI, we addressed several issues that can be expected to arise, in some form or another, when designing a real-time conferencing system for any other application, e.g., computer-aided circuit design or financial planning using spreadsheets. We discuss these issues in detail in this chapter, presenting the different options available for resolving each issue. We believe that different choices will be appropriate in different contexts, and therefore do not prescribe a single solution for all users and all applications. Instead, our discussion of the issues and options is oriented towards a hypothetical *designer* who is faced with the task of designing a conferencing system for his particular application and user community. To aid the designer in his task, we describe the factors and tradeoffs that must be taken into consideration when selecting among the available options. (The designer may in some cases allow for more than one option in his system, with the choice for a particular conference being made at "run-time" by the users involved.) We will in this chapter refer back to the joint document editing example, indicating why particular design choices were made the way they were, what alternatives were available, and how the selection among choices might be made in a different application or different setting.

Our discussion focuses primarily on the underlying *functions* presented to the user, rather than on cosmetic details about how the functions are presented such as different kinds of command syntax or pointing and selection methods. While these details may be important in making a system "user-friendly" and "easy-to-use", it is the user's *conceptual model* of what the system can do that ultimately determines whether the system will be used. Details of the "physical" interface can usually be selected after an appropriate conceptual model has been defined, and considerable literature [1, 2] is available on this subject.

For each kind of different function that can be provided in a real-time conference, the following issues will be addressed:

1. What *objects*, of what type and structure, are used to support the given function.
2. What *operations* are performed on the given objects in order to implement the function.
3. Which *users* are permitted to perform the operations.
4. How *concurrent* operations by different users are handled.

We will first discuss these questions in the context of the "main" application function of a conference, be it document editing or circuit design. We will then describe several other functions that may be useful in conducting and managing a conference. These functions, which include participants' leaving and joining a conference and support for participant negotiations, we shall collectively refer to as *meeting support*. In selecting a set of meeting support functions for a given real-time conference system, it is important to bear in mind the effect on the user interface of providing these functions. Each new function introduces another set of commands that the user must know about, and the additional power is gained only at the expense of a more complex interface. The potential for user confusion is considerably higher in a real-time conference than in a solitary user interaction with the system, and it may in many cases be best to err on the side of simplicity, i.e., leave out more advanced functions or at least make it optional for users to know about and use them.

We also consider in this chapter what special support can be provided for "conferences" involving only two participants. The need for two-person simultaneous interactions is likely to arise very frequently in practice, and such interactions tend to have a somewhat different nature than interactions among three or more people.

For most of this chapter, we assume that voice communication is not included in the real-time conferencing system. Rather, the participants in a conference must themselves establish a voice channel for discussion by some unspecified external means, e.g., a telephone conference call (or a simple two-party call if there are only two participants). At the end of this chapter, we briefly discuss some issues that arise if voice communication were to be included as an integral part of the system. Integration of voice communication is desirable, and may be feasible with current and near-future voice interface hardware and packet voice communication protocols.

It is rarely the case that conceptual design can be carried out in a vacuum without consideration of the feasibility of implementation, with acceptable performance, in the given

implementation environment. We will therefore briefly mention implementation-related factors that do have a bearing on the conceptual design, as and when they arise. Implementation issues are treated more fully in Chapters 4 and 5.

3.1 Shared and Private Spaces

The primary purpose of a real-time conference is to share some collection of *objects* that are relevant to the problem or task that the participants are trying to solve or accomplish. We shall call this collection of objects the *shared space* of the conference. In addition, it is useful to support the notion of *private spaces* in which participants can consult or compose information without involving the other participants, in the same way that they can do with their private notes in a face-to-face meeting. (In the absence of explicit support for private spaces, the best that a participant can do is to "leave" the conference, work on some private information, and then return to the conference. This is cumbersome and is best avoided if possible; it should only be necessary when the private interaction is completely unrelated to the conference.)

Private spaces are only useful to the extent that it is possible to establish some relationship between the information in the shared and private spaces. Thus, a participant may wish to privately view some of the information from the shared space, perhaps looking at a different region of a shared document or viewing an object in a different display format, or may wish to copy information from the shared space for private editing and exploration of alternatives. Conversely, a participant may wish to "submit" information from his private space to the conference. In JEDI, for example, a participant can introduce an entire document to the conference, and can insert text selected from his private space into a shared document. Similar functions are likely to be useful in other applications as well.

Once a decision has been made to support private spaces in a conference, the question then arises as to which objects should be shared among all participants and which should be private to individual participants. This issue can be quite complex because an interactive computer system typically implements many kinds of "objects" at many different levels of abstraction: not just abstract application objects, such as documents and circuit drawings, but also a variety of objects that support interaction with the user. Such objects include *images* of application objects (such as a document window), cursors, mouse pointers, "paste

buffers", command feedback in the form of echoes and error messages, status and summary information, menus, and other on-line assistance. We examine here the factors involved in determining whether and how such objects should be shared in a real-time conference.

Sharing an "image" of the application objects being manipulated, e.g., the "superwindow" in JEDI, is usually important in providing the illusion of a common "blackboard". Knowing that all participants can see an identical image on their workstation screens provides a common basis for discussion and negotiation. Thus, a participant can say "this part of the document" over the voice channel with the assurance that he will be understood; this can be further refined using a mouse to "point" at specific parts of the shared image. However, sharing only the image, in the form of a character array or bitmap or graphic "display file", without providing direct access to the contents of the underlying application objects, e.g., documents, is inflexible because not all participants' workstations may have the same display capabilities or available space on their screens. To accommodate such differences, it is important to allow tailoring of the presentation of the shared space by local transformations at individual workstations. If only low-level information, in the form of an already-formatted image, is accessible then the kinds of transformations that can be performed are extremely limited. Thus, while it may be easy to "clip" an image down to the size of a viewport available for display, this only allows part of the image to be displayed. A participant might instead decide to "scale" the entire image down to the smaller size, but if scaling is performed on an already-formatted bitmap, the result is likely to be illegible. On the other hand, if high-level application information, i.e., the document contents, were accessible, it is somewhat easier to construct a smaller image that approximates what everybody else can see, e.g., using a smaller font.

While sharing high-level application information does allow for individual participants to tailor their views of the data, it may result in the loss of a common view that forms the basis for discussion, as with a blackboard. A compromise solution is the one taken in JEDI: both the document contents and an image, the superwindow, are shared, with the understanding that participants are encouraged to display the shared image unless their workstations are incapable of doing so or if they wish to asynchronously view different parts of the document(s). Thus, participants do have a common view when they want it, and also have the flexibility to deviate from that when they wish. (If it is desirable to find out *which* participants are looking at the shared image and which are not, this can be done using "status"

information as described below.)

We next consider other kinds of "objects" that will be needed to support user interaction in the shared space, and whether these objects should be visible only to individual participants or to all. (For user interaction in private spaces, the issue of whether or not other participants should see feedback does not arise.) Feedback from the system, in the form of an echo or error message or highlighting of selected information or a menu, occurs in response to an action by a specific participant. That participant should obviously see the feedback, but the question is should other participants be shown the same feedback as well? How this question is resolved will depend on several factors:

- How useful is it for a participant to see feedback generated in response to another participant's actions? For commands that involve several physical steps, e.g., SELECT 2 PARAGRAPH or other "line-oriented" command interfaces, sharing the command "echo" is useful in giving the other participant's a sense that one is actively doing something and not sitting idle. It may sometimes also be useful in demonstrating to another participant how to enter certain commands, allowing a real-time conference to be used for "training" purposes. These benefits may be counteracted by the possibility that participants may sometimes *not* be interested in seeing feedback generated by other participants and might find that it distracts them from their own interaction. (The additional communication bandwidth needed to display feedback to all participants might also be an issue.)

- How easy is it to distinguish feedback generated in response to different participants' actions? For example, in JEDI, the participants' mouse pointers are assigned different "shapes"; different colors could be used instead if participants' workstations had color screens. Command feedback from actions within a shared window are displayed within that window; since each window can have only one participant working in it, and that participant's name is shown at the top of the window, feedback from different participants' editing actions is easily distinguished. (This would not be so easy if we allowed two or more participants to concurrently edit within the same shared window.) On the other hand, certain kinds of feedback are difficult to separate so easily, and should be displayed only to the participant performing the action that caused the feedback. For example, if a participant moves his mouse pointer into a window reserved by another participant and attempts to acquire the window reservation, only the first participant sees the resulting error message. (And only the second participant sees the message informing him that his

reservation will soon be lost.) A requested menu or help display will often be larger than the shared window in which a participant is working; it is not desirable that this participant's request for a menu obscure information on other participants' screens that these other participants may be working on. This kind of feedback is again shown only to the participant to whom it applies, and does not disturb other participants. Note that some of these decisions could be made differently in a conferencing system that only allows one participant at a time to work in the shared space as a whole.

- How does a user distinguish feedback generated in response to his own actions from feedback generated by other participants' actions? This issue does not arise when feedback is shown only to the participant to which it applies. If, however, it is considered useful to share feedback, i.e., display it to all participants, then this question becomes particularly important. The purpose of the feedback is to aid the user in deciding his subsequent actions, and it is therefore not desirable to let a user mistake some other participant's feedback for his own. The main technique that can be used here is for feedback to be specially highlighted on the screen of the participant to which it applies, in a way that other participants will not see. Thus, echoing and error messages in a document window will flash or appear in reverse-video on the screen of the participant working in that window; other participants will see the feedback without any highlighting. (This is also less obtrusive to the other participants should they be busy working on something else.) Similarly, even though every participant's pointer into the shared space is assigned a different shape, each participant sees his own pointer with a fixed shape that is identical to what he sees when in a editing session by himself. Thus, a participant can easily distinguish his own pointer from all others and can correctly determine which way to move his mouse in order to reach a desired pointer position.

In summary, sharing feedback by displaying it to all participants can be useful, but only if individual participants can easily determine which feedback applies to them, if it does not intrude on other participants' activities, and if it does not take up so much extra bandwidth as to seriously degrade performance.

3.2 Access Control

The types of objects shared in a conference and the operations on these objects will vary from application to application. Unless there are important security reasons, permission to perform the operations should in general be given to all participants in the conference. (This does not necessarily mean that all participants may perform operations at the same time; that problem is discussed below in Section 3.3.) In certain special situations, the information being shared might be particularly sensitive, and some users might be allowed in the conference only as "observers" with no update privileges. Similar considerations might apply in determining whether or not all participants can see a given object or image, e.g., in a game where participants are meant to have different views of the shared space.

We first discuss some problems that arise when the objects being manipulated in a conference have independent existence outside the conference, objects for which access controls have already been set. In such a case, giving all conference participants unlimited read and update access to an object may violate the access control specifications, i.e., a participant might update objects that he is not normally allowed to update, or see information that he is not normally allowed to see. This problem arose in the joint document editing example, and our solution was as follows. A participant is only allowed to update documents to which he has update access, obeying existing access controls on documents. An alternative might have been to allow any participant to update the conference's volatile copy of a document, but only allow participants with update access to permanently save, or discard, the document copy. This is also a reasonable approach, except for the complications that arise when we consider that the conference might "auto-save" a document without any participant having explicitly requested it; it is not clear whose permission must be obtained in order to auto-save, and this alternative was rejected for JEDI.

As far as read access is concerned, we decided to allow all participants to see all documents submitted to the conference. While this might appear to violate access control restrictions, it is in fact not unreasonable, based on the following considerations. (These considerations may not apply in all other situations, e.g., classified military information.)

1. If only those participants having read access are allowed to see a given document (or other object), then the system and the participants must keep track of who can see what. Without some assurance that all participants can see more or less the same information, the common basis for discussion is lost.

2. Access control requirements for an object are usually not well anticipated, and often need to be updated as needs change. If a given object is not accessible to some of the participants in a conference, the others will often want to given these participants read access in order to have a common basis for discussion.
3. Read access controls can easily be circumvented on most systems by a user copying the contents of an object into another object and releasing the second object to anybody he please. It therefore seems pointless to be overly cautious about participants not having read access to an object. (We do not want to be too casual about read access, however; in JEDI, we do require that at least the participant submitting the document have read access to it.)

All of the above assumes that the existing hardware and software base allows the kinds of manipulation of access control information described; this may be true on personal computers that allow any form of access control to be implemented, or in an advanced "object management" system such as ENCORE [119]. If the conferencing system is to be implemented on a traditional time-shared operating system, however, this may not be the case. On most file systems, for example, it may not be possible, short of the user actually attempting an operation, to determine whether a given user has read or write access to a given file. In addition, a program must usually be "logged in" under one particular user, and runs with the privileges of that user. Then, a participant other than the one under whom the conferencing program is logged in might be able to update a file that he is not permitted to update, or, conversely, not be able to update a file that he normally does have permission to update.

In cases such as the above, it is not clear that any scheme can be devised that is both "correct" in not violating access control specifications yet not unduly restrictive to the participants in a conference. Some compromise will probably be necessary, such as the following: Give the user running the conferencing program the power of *chairperson*, with the ability to give or take away permission to operate on the shared space. Then, even though operations on files are executed by the system based on the chairperson's privileges, the chairperson can at least watch the other participants and intervene if he does not like what some participant is doing. It is still not possible for a participant to update, from the shared space, a file for which the chairperson does not have update access. The participant could, however, copy the necessary information into his private space, which presumably is implemented by a program running with his own privileges, and update the file from there. Note also that most operating systems do not allow a running program to change its user-name. It will therefore not be possible to "change" the chairperson in mid-conference,

except by copying the entire state of the conference to a different program running under the "new" chairperson's privileges; this may be too expensive or even impossible, and a given conference on such a system will usually have a single fixed chairperson.

The situation is further complicated if we consider a *new* object or file that is created from within the shared space of a conference. In this case, who is the "owner" of the object or file, with the power to specify which users have what access? The owner should perhaps be the participant who performed the operation that created the new object, but again this may not be implementable on a system that automatically sets the owner to be the user running the conferencing program, i.e., the chairperson. Furthermore, which users should be given read or update access, from their own programs outside the conference, to the new file? It seems reasonable to give all current participants read access, but again the access controls on a new file are often set automatically based on a default that is system-wide or obtained from the user's "profile". Even though the owner of an file can always change the access controls if he is not satisfied with the default, he is usually not aware of what the default is and will not notice if the default differs from what he expected or desired; he will only notice the problem much later, when some user tries to access the file and is unable to.

3.3 Concurrency Control

While certain shared objects in a conference might be updatable only by a single fixed participant, e.g., the position of a participant's mouse "pointer", most objects will be updatable by more than one participant since that is how a conference supports interaction among users. Consider the following scenario that may occur when participants can update a shared object concurrently: Participant A inserts some text into a document, and then participant B deletes a region of the document that includes A's insertion. If we assume that B saw A's insertion and knows what he is doing, then his deletion of what A inserted is not a problem as far as the system is concerned; the ability to edit other participants' contributions is an essential part of the give and take of a real-time conference. (If B is acting maliciously when he deletes A's inserted text, that will presumably be dealt with by means that do not properly belong in the computer system.) If on the other hand B did not know that the text he is deleting includes some text that A is inserting, then this is a problem because B's deletion command causes an unintended effect that he is not aware of. This can in fact happen if

participants are allowed to enter commands concurrently: even though A's insertion "happened first", from the point of view of the system, its effects were not yet displayed on B's screen when B entered his deletion command, because of communication and processing delays.

One approach to such concurrent conflict is for the system to do nothing about it. The system leaves it to the participants to notice when interference occurs (such as in the above example), and to take corrective action (such as restoring the deleted text) if they find it necessary. While it may be reasonable to leave the responsibility to the participants, given that each can see what everybody is doing, the problem might simply compound itself: On noticing interference among their actions, two or more participants might concurrently attempt corrective action thereby causing further interference and confusion. This is a well-known problem with "terminal-linking" systems; the problem is bad enough when users are simply typing text messages to each other, and is even more serious when users' input affects application data.

A more reasonable approach to dealing with concurrent conflicting operations is to use either or a combination of the following techniques:

1. *Validation*, which invokes protective measures only after a problem of concurrent conflict is detected.
2. *Reservations*, by which a user protects an object or region of an object against modification by others.

3.3.1 Validation

Using validation, the system "detects" a conflict when it determines that a given operation will probably have a different effect from what was intended by the participant when he entered the operation. A simple detection mechanism is the following. Each operation entered by a participant on an object is tagged by the system with a *version number* that identifies the version of the object that the participant saw when he entered the operation. Then, when the operation is ready to be processed by the system, typically after some communication delay, the version number is compared with the current version number of the object. If the version identifiers are different, then a conflict has occurred because there was an intervening concurrent update that the participant did not see. This can be refined further by checking how the intervening update actually changed the object, and pronouncing a

conflict only if this operation would in fact have a different effect. For example, if some other participant updated the document concurrently but the latter's update does not overlap this participant's update, the system will not treat this as a conflict. How this is determined is likely to be very application-specific, and requires that additional history information about updates to the object be maintained. Such information might often be needed anyway, to support various forms of command "undoing".

Once the system decides that a conflict probably did occur, it can do either of two things:

1. Reject the participant's operation on the assumption that the participant will not want it to have a different effect than what was intended. The participant who entered the operation is sent an error message, and he may or may not wish to attempt the operation again after seeing the new state of the object.
2. Perform the operation anyway but inform the participant that the operation may not have had the effect he intended. In this case, it is still up to the participant to take corrective action if he is not satisfied. Unlike the case where the system provides no control at all, the system now aids the participants by determining when a conflict has occurred. The problem of more than one participant concurrently attempting corrective action can still occur, but the system, having detected the problem, is now in a better position to forestall that. For example, the system may introduce a short delay during which only one participant is allowed to proceed.

Which of the above should be used depends on the relative likelihoods that, once the system detects a conflict, the participant will want his operation to be rejected or will want it to be performed anyway. While it is impossible for the system to always correctly guess the participant's desires, we note that the relative likelihoods will be influenced by how intelligent the system is in detecting "conflicts". For example, if the system only checks a version number on the whole document, it will often "detect" a conflict when concurrent operations affect disjoint parts of the document. A participant is much less likely to want to abort his operation in this case than if the system detects a conflict only when concurrent updates do overlap.

It is important to note that the system, using any form of validation, cannot catch all cases of concurrent interference. For example, an operation by a participant may carry an up-to-date version identifier of the affected object, but even if the participant's screen already shows the latest version of that object the participant may not have noticed it; there will always be some delay between when an update appears on the screen and when the participant sees it and comprehends its meaning. Or, the participant may have been entering

a command and did not react quickly enough on seeing a concurrent update to stop his command. Thus, while the system can often be helpful in detecting concurrent interference, it cannot cover all cases because of user reaction delays outside the system.

3.3.2 Reservations

A *reservation* on an object (or on a collection of objects, or on a region of a large object) prevents all users except the holder of the reservation from updating the object. In its simplest form, a single "super" reservation can be set on the entire shared space of a conference, allowing only one participant at a time to enter commands on the shared space. There will of course be little interaction if the same participant holds the reservation for the duration of the conference; mechanisms for passing the reservation are needed, which are described presently.

If more concurrency is desired in a conference, reservations can be set on smaller disjoint subsets of the shared space. An example is provided by the joint document editor, where participants hold reservations on different windows and on non-overlapping regions of a document (or on regions of different documents). Thus, it is not possible for participant A in the scenario above to insert text into the region about to be deleted by participant B because B has that region reserved. (If B reserves the region, and then deletes it, after A inserts some text, we do not consider this a problem because B saw what A inserted.)

Assuming reservations are to be used, the first question to be addressed is: How many reservations on how many different kinds of objects are needed? A single reservation on the entire shared space may be used, or a collection of reservations at a finer granularity, or both as in JEDI which allows multiple reservations so long as nobody holds the "super-reservation".

If multiple reservations are permitted, allowing multiple participants to work concurrently on different parts of the shared space, we must carefully consider the burden on the user of understanding what a given reservation means, and how to acquire and release it. "Window" reservations, as in JEDI, are easy to understand: the windows themselves are easily distinguishable, and only one participant at a time can work within a given window. And no special commands are needed to acquire a window reservation; the same commands used to "select" a window implicitly try to acquire the reservation. It might be plausible, in this or

another application, to allow more than one participant to work within the same shared window. This allows each participant to see where he is in relation to other participants, and may be particularly useful when different participants' cursors are close together. But we rejected this choice in JEDI because it introduces other problems such as participants concurrently trying to scroll in opposite directions, and where to display feedback for the different participants' commands. The choice made in JEDI is for participants wishing to concurrently edit the same document to use different windows; this avoids interference at the window level, at the cost of being able to display less information in each window than if the participants worked concurrently in one larger window.

Window reservations do not prevent interference among participants editing the same document through different windows; "region" reservations were therefore introduced. The way region reservations were defined in JEDI has the following advantages:

- The maximal region of the document that the participant can modify with his next command, e.g., DELETE, is the "selected" region, between his mark and cursor. It therefore makes intuitive sense to reserve precisely this region and protect it against changes by other participants.
- Again, no additional commands are needed to acquire a region reservation; "selecting" a region, by moving the cursor or mark, implicitly requests a reservation on that region. This would not be the case if users could reserve arbitrary regions that are unrelated to the region selected for deletion or copying; special reservation commands would need to be defined.
- Showing the user what region he has reserved involves no extra work, because the "selected" region must be highlighted anyway. Again, the problem of properly displaying the information would be worse if the reserved and selected regions were allowed to be different.

Thus, by keeping the reserved region identical to the selected region, the burden on the user is minimized.

Once the number and variety of reservations to be used is determined, the next question is: how do participants attempt to acquire, or *request*, reservations, and how and when are such requests granted? In some cases, such as the window and region reservations of JEDI, a reservation request is *implicit* in certain related commands, i.e., a command to select a window or region. In other cases, there may be no such conveniently related command, and special commands must be introduced to *explicitly* request a reservation; this is the case with the *super-reservation* in JEDI.

Requests for a reservation may be granted *manually* by some participant designated as "chairperson". If so, requests should be saved on a queue so that the chairperson knows who has requested a reservation and when. As with the super-reservation in JEDI, the chairperson can at any time grant the reservation to any participant, taking it away from the participant who currently holds it. This may be supplemented by polite warnings from the chairperson that a participant should finish what he is doing before he loses his reservation.

Having the chairperson manage reservation requests is reasonable when there is only one or a few reservations to manage. With a possibly large number of reservations, the burden on the chairperson (or other participants designated to manage reservations) will be prohibitive. Reservation requests may alternatively be managed *automatically* by the system according to some scheduling policy. (It is also possible to have a mixture of strategies, e.g., the system automatically manages reservations except when the chairperson manually intervenes.) The policy involves deciding what happens when one participant requests a reservation that is currently unavailable because some other participant holds it:

- The reservation request is *queued* until such time as the reservation becomes available.
- An unsuccessful reservation request is *discarded*; the participant himself must try again at a more opportune moment.

Queuing a reservation request removes the burden of a participant having to resubmit the request, but makes the user interface more complex: that there is a queued reservation request needs to be displayed somewhere, and if a participant changes his mind about the reservation request and decides to do something else instead, he needs a command for *withdrawing* his queued request. Even with such a command, he may forget to withdraw his request, in which case he will later be interrupted on receiving a reservation he no longer wants. All this can get very complicated when there are many reservations; thus, window and region reservation requests are not queued in JEDI. On the other hand, there is only one "super-reservation" in JEDI, and it is not difficult to display and manipulate the state of a single request queue.

The scheduling policy may be refined to provide some degree of *fairness*, by not allowing a participant to hold a given reservation for too long. (This also protects against crashes of a participant's workstation.) This can be done by setting some kind of *time limit* on a reservation, such as:

- An *idle* time limit: a participant loses a reservation if he does not use it, i.e., perform a command for which the reservation is needed, for some specified period of time.
- An *overall* time limit on how long a participant can hold a given reservation once granted. In this case, it is useful to warn a participant a short time before he loses the reservation so that he can finish up what he is doing.
- A *conflict* time limit: the participant keeps the reservation indefinitely so long as no other participant wants the same or a conflicting reservation. Once a conflicting reservation request is made, this participant is notified and loses his reservation after the given conflict time limit. (This method was used in JEDI for window reservations.)

A combination of the above might also be used, e.g., with a moderately short idle time limit and a longer overall time limit.

3.3.3 Comparison

Reservations are useful in avoiding concurrent conflict, but they incur an additional overhead on the user. This is especially so when we consider that the shared space of a conference may have complex structure; in JEDI, for example, reservations on windows do not prevent interference at the document level, and reservations on document regions were therefore included as well. The user overhead of acquiring and releasing reservations can be minimized by making some reservation operations implicit in existing commands, as in JEDI.

It is also possible to use a concurrency control method that combines reservations and validation. For example, reservations could be set on windows but validation used to detect concurrent conflicting operations on the same document from different windows. Or, reservations could be made optional in that each participant decides whether or not he wishes to protect himself against concurrent updates by others. For a participant who chooses not to set a reservation on an object he is updating, either validation or no control at all may be used.

Validation does not require additional user commands. However, validation will be practical only when the probability of interference is very low. This may be the case when communication delays are short (e.g., all workstations located on a single local area network) and few participants are acting concurrently. As the number of participants and the communication delay increases, however, the probability of concurrent operations conflicting increases, and the need to frequently resubmit or recover from conflicting operations will be a

source of irritation.

It is possible for the participants in a real-time conference to use the *voice* channel to negotiate the taking of turns and thereby avoid concurrent conflict. If the participants can successfully do this, no special protection in the form of reservations or validation is needed from the conferencing system. However, this will not work very well with a large number of participants. If there are more than two or three participants, the voice channel itself becomes an object of contention. Participants may have difficulty agreeing on whose turn is next, and concurrent conflicting updates may occur quite frequently. Therefore, reservations seem particularly necessary in large conferences, whereas in smaller conferences validation or no control at all may be sufficient.

3.4 Meeting Support Functions

We next look at some "generic" functions that may be used to facilitate the management of a real-time conference. While the specific nature of a given function, and whether the function is provided at all, may vary from system to system, the issues involved in designing objects and operations for a given function are common. These are discussed below, together with the problems of access control and concurrency control for each kind of function.

3.4.1 "Chairpersons" and Participant "Roles"

In most face-to-face meetings, participants can be observed playing different "roles" such as moderator or chairperson, note-taker, and so on. It is possible to build such roles into a real-time conferencing system if desired by defining exactly what a given role can and cannot do, in terms of access to and ability to update different objects, and who can assume a given role and when. However, participants' roles in a meeting, and even the meaning of different roles, frequently change and the system should not get in the way of this dynamic behavior.

One useful role is that of a conference *chairperson* with overall control over the conference, e.g., the ability to allow participants into the conference and to grant reservations. (The concept of a chairperson has appeared in other real-time conferencing systems, as well as in network voice conferencing.) Having a chairperson is particularly useful for resolving contention in large groups, say more than three participants. For smaller

conferences, participants can usually coordinate their activity without resorting to a central mediator. (The joint document editing system has a chairperson regardless of conference size, but the powers of the chairperson do not have to be exercised if not found necessary, e.g., participants may reserve individual windows without the chairperson's intervention.)

If a chairperson role is defined, it should be flexible enough to allow different participants to play that role at different times, e.g., by allowing the current chairperson to designate a new one. More complex participant roles could be defined, but it is not clear how useful they will be; users will have to learn new commands, and invoke them every time they wish to change roles. Role definitions can be very useful for longer-term coordination, as in [66, 18]. The burden of defining and assuming different roles is not unreasonable when users are working individually over a period of time; in a real-time situation, however, these can easily get in the way. Instead of defining highly stylized and restrictive roles, it might be better to give all participants free access to the shared objects involved and let them implicitly switch "roles" as they please. For example, a participant can implicitly play the role of "note-taker" by acquiring a reservation on the object (e.g., a document) being used to record the "minutes" of the conference; it is not necessary for the conferencing system to provide special support for the concepts of "minutes" or "note-taker".

3.4.2 Participant Status Information

It is important for a participant in a real-time conference to know which other participants are present and watching what he is doing. (Or to know who is listening to him speak on the voice channel - this is discussed separately in this chapter, but the same principles presented here apply.) In a face-to-face meeting, this information is readily available because all participants are physically present and each can see the others. Since this is not the case in a real-time conference, some means of presenting "status" information to the participants is needed. One might take the approach used in telephone conference calls, where each participant announces his presence and every participant is expected to rely on his own memory of who is present; this hardly seems necessary in a conferencing system that has the power of a computer at its disposal.

The simplest kind of status information is illustrated by the "status window" in JEDI: a list

of names of those participants who are present in the conference.[9] In addition to listing just those participants who are currently present, it may sometimes be useful to present status information about other users, e.g., participants who were "invited" (below) but who have yet to join, or participants who were present in the conference but have since left.

Not only does a participant wish to know who else is present in a conference, he will typically wish to know who is *doing* what: working on a given window or object, or pointing at information with a mouse. This information can be presented by *labeling* the object in question appropriately, e.g., each shared window in JEDI carries the name of the participant, if any, who is working on it. In some cases, e.g., participants' mouse pointer positions, labeling with a participant's name is too unwieldy; some kind of color or shape coding can be used instead. If so, it will be important to indicate which colors or shapes have been assigned to which participants; these might be displayed, for example, next to the participants' names in the "main" status window. An alternative way of presenting such information is to highlight or flag a participant's entry in the main status window in some distinctive way. This can be useful if there are not too many different ways of highlighting a participant's entry, making them hard to distinguish and understand. In JEDI, the current chairperson, the holder of the "super-reservation", and the participants requesting the super-reservation are flagged in the main status window. More detailed information, such as which participant is working in each of several windows, is hard to present in this manner without making it too confusing; other approaches, such as labeling relevant objects, should be used instead.

Finally, a participant may wish to know what each other participant is *looking* at, so as to be sure whether they all understand what he says when he refers to the displayed information. In a face-to-face meeting, it is usually possible to discern that some participant is looking at his notes, or at a different part of the blackboard, and therefore may not be paying full attention; it is then possible to interrupt him and call his attention to a particular item on the blackboard. A useful approach is to define one shared window, or collection of windows, as the "main focus" of the conference with the assumption that participants will most of the time be looking at this main focus. Then, it is only necessary to indicate which participants are "distracted", i.e., not currently paying attention to the main focus; this could also be done by

---

[9] Other identifying information may be used instead of names, if it were available and considered useful. For example, it may well be possible in with future technology to display a small digital image of each participant's face, retrieved from an image database or transmitted in real time from a camera.

flagging the participants' entries in the main status window. (A participant should not be flagged as distracted if his attention wanders only for a very short time, e.g., he moves into his private space and then comes back quickly; notifying everybody of such momentary distractions will only be source of irritation.)

If more detailed information is desired as to exactly what a "distracted" participant is looking at, an "outline" of the document(s) being shared could be displayed that indicates the position of the main shared window(s) as well as which other regions specific participants may be separately looking at. This idea could be used in other applications as well, e.g., in a graphics-based system a "map of the world" could be displayed with rectangles outlining the main shared window and participants' private windows, if any.

A different kind of "status" information relates to the *performance* of the conferencing system. If the system is responding sluggishly due to communication delays or errors (which are usually "converted" into delays by later retransmission), it is useful to know whether the problem lies in the communication path to a particular participant's workstation; it may then be desirable to remove that participant from the conference in order to let the conference continue properly. Such action might be taken automatically by the system, by setting a "timeout" and removing a participant's workstation if it does not respond, and the participants need not be bothered with the information. However, the system might not always make a choice that the participants like; whatever the timeout interval chosen, participants might sometimes be willing to tolerate a longer delay, and sometimes might be more impatient and only accept a shorter delay. It therefore seems reasonable to let the participants themselves, or a specially designated participant such as the chairperson, decide when and whether to remove a "slow" participant because of delay problems. (If the chairperson himself is the one who is slow, the system could automatically designate a new chairperson.) The performance-related information that is displayed, on request or on a timeout, should be in a form that is meaningful to the participants, e.g., an indication of which participants' workstations are experiencing delays over a given threshold, rather than a collection of numbers with only low-level significance.

It should be apparent from the above that the amount of "status" information that can be presented in a conference is considerable. Not all of it may fit on a participant's screen at the same time, and even if it did it would leave little room for viewing application information, which is after all what the conference is for. Fortunately, participants will typically not be

interested in all of this information all of the time. It will usually be sufficient to allocate a small amount of screen space for the main summary of the conference status, like the shared status window in JEDI, with a participant calling up more detailed information in a "pop-up" viewport only when he needs it. (Even the main status window might be displayed only on request, if screen space is scarce; participants can still rely on their memories to know who is present.)

In addition, it is useful to provide special notification of significant *changes* to the conference status, e.g., a participant leaving the conference or a new participant joining; such notification should be accompanied by some attention-getting mechanism such as ringing a "bell" or flashing the part of the screen where the notification appears. Notification of significant changes should be provided even if the status information in question is visible elsewhere on the screen, because the participant might not be looking at it and may not notice an update.

3.4.3 Interjections

A large part of the usefulness of simultaneous interaction in a face-to-face meeting comes from the ability to interrupt a participant in order to inject a brief remark or otherwise elicit attention. Similar mechanisms are needed in a real-time conference, that can be quickly and easily invoked and do not incur the overhead of formally acquiring a "reservation". Voice is usually the quickest way of making an interjection, but it may not always work well depending on how the voice channel in a given conference is implemented. For example, many voice conference "bridges" cut off all speakers except the current one, and therefore allow an interjection only when the current speaker pauses for long enough. Even when this cutoff problem is not present, a separately-implemented voice channel will not identify which participant is speaking or interrupting; either all participants' voices are clearly distinguishable, or a participant must identify himself verbally when speaking.

Within the conference system, some operations are by their very nature intended to attract attention, e.g., pointing with a mouse, or requesting the "super-reservation". In addition to these, however, some "interjection" facility whose sole purpose is to attract attention is likely to be useful, in particular:

- The ability to compose and send a brief text *message*, e.g., a "one-liner" that flashes on the all participants' screens. This may be particularly useful for *private* communication with a selected participant, which is not possible over a shared voice channel.

- The ability to highlight a specific object or window, e.g., causing it to blink, perhaps by pointing and clicking a mouse button. (A participant who does not currently have the desired object or window on his screen will instead receive a message informing him that his attention is being elicited.) This is useful when other participants may be looking elsewhere, and a given participant wants to call their attention to some data that he wishes to discuss or update. No semantics is associated with this mechanism by the system, it is simply a means for participants to quickly signal to each other. Participants are free to ignore such requests for attention, and may even instruct their workstations to suppress certain notifications, e.g., from a participant who is overusing this facility and causing too many interruptions.

3.4.4 Negotiations

In JEDI we saw an example of *negotiation* where a participant wishing to grow (or shrink) the shared superwindow first elicits reactions from all participants, in the form of maximum acceptable superwindow sizes, to the proposed change. There are many other situations where a similar pattern of eliciting and collecting responses can be useful, such as:

- Collecting "votes" on some resolution, or remarks about some object (e.g., a new document version), or rankings from a set of alternatives.
- Collecting participant's reactions to some proposed operation on the shared space, e.g., changing the superwindow size as in JEDI, or terminating the conference.
- "Inviting" a set of users to join the conference and then waiting to see which ones join and which ones don't.

In designing a negotiation protocol for any of the above functions, the following questions must be considered:

- *Who* can initiate a particular kind of negotiation? This privilege might be restricted to the chairperson, or given to all participants. The latter approach was used in JEDI, allowing any participant to initiate a window size negotiation, but subject to the restriction of holding the super-reservation.

- How are participants *notified* that a negotiation is taking place and that a response is expected? This will typically require some attention-getting mechanism, e.g., flashing a message on the screen or ringing a bell.

- What *other* activity can take place concurrently in a conference while a negotiation is under way? It is possible to envision participants working in one or more windows while a

negotiation is taking place, but this can easily get confusing. In any case, a participant must (but cannot be forced to) usually interrupt what he is doing anyway in order to respond to the negotiation. Thus in JEDI it was decided not to permit editing while a super-window size is being initiated, and this was implemented by requiring the initiator of the negotiation to hold the "super-reservation". Holding more than one concurrent negotiation about the same topic can similarly be a source of confusion. Not all activity in the conference should be prohibited during a negotiation, however; participants should at least be permitted to leave the conference or work in their private spaces. New participants should be allowed to join, in which case responses should be elicited from them as well.

- When does the negotiation *terminate*? There is certainly no need to wait any longer once a response has been received from every participant in the conference. However, some participants might take a long time to reply, or never reply, and these participants should not be permitted to hold up the conference for too long. It is therefore necessary to provide some kind of *timeout*, and/or allow the participant conducting the negotiation to *interrupt* the negotiation and stop waiting for responses.

- How are the responses *displayed*? This would depend on the type of information carried by a response, e.g., "votes" may be tabulated alongside the participants' names, or some problem-specific algorithm may be used such as the computation of alternative super-window sizes in JEDI.

- *When* are the responses displayed? They must of course be displayed when the negotiation terminates, whether by virtue of all responses having been received or on a timeout. It will in most cases be useful, however, to display responses as they arrive, both to give feedback that something is happening and also to allow the participant to prematurely terminate the negotiation should a particular early response seem to warrant it.

- To *whom* are the responses displayed? Unless there is some special reason to be secretive, all participants should see the responses; showing them only to the initiator of the negotiation will make the others feel left out. If a participant continues to work, e.g., in his private space, he may not wish to have his screen cluttered and be interrupted by responses to the negotiation; he may instruct his workstation accordingly, asking to be

notified only when the negotiation is terminated.

- If the negotiation involves a proposed operation on the system, in what way do the responses *constrain* the possible execution of this operation? (This question does not arise in cases such as collecting participants' remarks about some application data, which is simply information-gathering and does not involve any proposed operation.) It is conceivable that a voting rule could be implemented by the system, with a proposed action automatically executed if a unanimous vote or some specified majority is achieved, and automatically rejected when enough negative votes are received to make such consensus impossible. Rules could similarly be devised for other negotiations, e.g., automatically selecting the minimum of the offered sizes when trying to grow a window. However, we feel that it will not often be desirable to build such rules into the system, because it will not always be possible to capture a participant's intentions and preferences in an algorithmic rule and because it limits participants' flexibility to make decisions unconstrained by the rules. Therefore, it will usually be best to make responses *non-binding* and to let an authorized participant (e.g., the initiator of the negotiation, or the conference chairperson) decide how to proceed. The decision made may go against one or more participants' responses, but such differences are best resolved by the participants outside the system. While leaving the choice to the participants, it is still possible for the system to aid in the selection process, e.g., by presenting a set of plausible alternatives based on the responses, as in JEDI.

3.4.5 Access to a Conference

When a conference comes into existence, the system must be told which users will be allowed to participate. The simplest form of specification is a list of user-names (i.e., an "access control list"), although other forms might often be useful:

- A conference might be designated as *public*, allowing anybody to join.
- If the system supports the notion of "user groups" or "projects", a conference might be made open to all members of a given group or project.
- A *password* might be selected for the conference, and distributed to selected users by some unspecified outside means (e.g., word of mouth); only users who can supply the correct password are permitted to join the conference.
- The set of allowed users may be *implicitly* defined by the object(s) being manipulated in the conference, e.g., all users having access to a given document.

A combination of the above might sometimes be used. In addition, a conference might or might not allow users who have not explicitly been given permission (by one of the above means) to *request* permission to join. Then, some participant designated as "chairperson" would consider such requests and grant or deny them, or this power could be given to all participants, allowing each to "bring your friends".

As with access controls for any other kind of object, the question arises as to *who* specifies, and can update, the access controls for a conference. A typical approach, taken in JEDI, is that the user who first creates an object, in this case the conference, is its *owner* and has the sole authority to set and update access controls. An extension might be desired allowing users other than the owner to grant access rights to other users; if so, the question again arises as to who designates these other users. The authority to grant authority can be taken through as many levels of indirection as desired, and the same question arises at each level; the recursion must be terminated somewhere with an object's creator being the one who has to specify the first level of access rights. Few systems actually permit arbitrary levels of indirection in granting access rights; one or two levels is usually sufficient for practical purposes.

3.4.6 Rendezvous

Once a conference has been created, some means is needed for authorized users to find out about it so that they may join the conference if they wish. There are two kinds of mechanisms for doing this:

1. *Inviting* specific users by sending them messages describing the conference.
2. *Announcing* a conference by including a description in some shared database(s).

Inviting users is the more direct and immediate method of informing them about a conference, but it is limited to users who can be explicitly named and enumerated; it is not possible, for example, to "invite" the entire public in this way. (At best, it may be possible to send a message to all users logged onto one or more given machines.) Inviting a user of course requires knowing where to send the invitation; this will usually require looking up some kind of *name server* [9, 92, 109]to determine a user's workstation's network address. (If name servers are for some reason inaccessible, the inviting participant should be allowed to manually enter the invitee's network address if he knows it, e.g., if the invitee supplies his address over the phone.)

A conference can be "announced" by placing a description of it in any of several kinds of shared database, e.g., a special "conference server" or a project database or a calendar; which one(s) to use will depend on the expected use of these databases and on how far out into the network the word is to be spread. A user will only find out about a conference if he looks at one of the databases where it has been announced; he may also program his workstation to periodically "poll" selected databases for him, looking for new conferences.

Conference invitation messages and announcements should carry information that will help a participant decide whether or not he wishes to join, e.g., a brief text "statement of purpose", and/or a description of which objects will be discussed and manipulated, and/or a scheduled time and duration. (In Chapter 5, we shall see that additional information, not seen by the user, will need to be included in invitations and announcements for implementation purposes.)

3.4.7 Planning and Scheduling

Both of the above mechanisms, inviting users and announcing conferences, require some prior agreement in order for them to work: a user will only receive conference invitations if he "registers" his workstation address at one or more name servers, and will only find an announced conference if he (or his workstation) looks in the right places. Such agreement will often take place outside the system, e.g., participants agree in a prior face-to-face interaction or over the phone to hold a real-time conference at a particular time.

If the right tools are available, however, some of the work of planning and scheduling a conference could be done using the system itself. A conference can be created, or "called", well in advance of its actually being held, and announced in a calendar system or a project database or wherever. This gives potential participants time to indicate their availability and to negotiate a suitable time for the conference, and to make other preparations that they may find necessary, e.g., locate relevant private notes or information that they would like to "submit" to the conference. (As we describe in Chapter 5, advance notice might also allow participants to transfer copies of large shared objects such as documents to their workstations, avoiding this transmission delay when the conference actually commences.) Conference schedule information can be used in reminders or alarms that are provided by many calendar and "tickler" systems, e.g., a participant could ask that he be interrupted by a message when the scheduled time of a conference approaches. This same idea can be used

during a conference as well, e.g., participants could be warned, once or periodically, when a conference runs for longer than the scheduled duration.

Even planning and scheduling a conference well in advance will only succeed when there is an understanding that potential participants will do the right thing — check the calendar system, or look at the databases for the projects they are involved in — at some time before the conference is due to begin. This is no different from electronic mail, or even the postal service — the system works only to the extent that users check their mailboxes frequently enough.

3.4.8 Joining a Conference

Regardless of how a user finds out about a conference, by invitation or by looking in a database, some interface must be provided for him to indicate his desire to join some conference. For example, he might select a conference from a list, consisting of invitations received plus conference descriptions obtained from one or more databases, by pointing. If explicitly invited to a conference, a user should have a way of "declining" to join. Then, if the other participants are waiting for him to join (e.g., to complete a "negotiation"), they will know that there is no need to wait any further. The user should of course have the option of changing his mind and joining a conference even after having declined.

An authorized participant may join a conference at any time, whether or not any activity has taken place. (Similarly, "requests" to join from users not explicitly allowed should be entertained at any time, unless the conference is completely closed to the public.) A newly-joining participant should be provided with an up-to-date version of the objects and windows in the shared space, in order to follow and participate in the subsequent action. (The participant may instead wish to see a more detailed history of what went on; that is described later.) In certain cases, though, it may be necessary to *delay* the admission of a joining participant for a short while. For example, if the participant can only "request" to join, his admission to the conference will have to await the approval of some authorized participant, e.g., the chairperson. Or, even with permission to join the conference, the participant might not automatically receive access to a given object. (This situation should admittedly be rare.) Again, some authorized user must give the new participant such access, perhaps after updating the object(s) in question to a state considered appropriate for release. In some cases, the system itself may briefly delay providing the new participants with shared objects

until a more opportune moment. At the start of the conference, for example, it may be convenient to wait for additional participants to join so that they can all be sent the data at the same time (perhaps using multicast communication, as described in Section 4.6.3). Or, a window size negotiation may be taking place, in which case it is best to wait and not show the participant the current window size because he will have to readjust his screen soon when a new window size is chosen. Or, if a current participant is entering a command when a new participant joins, the system may wait a few seconds while the first participant pauses or finishes his command. In cases where a joining participant does have to wait, he should be given some feedback informing him that his case is being considered; in the absence of such feedback, the participant might think there was a communication failure or crash of the conference and simply give up.

3.4.9 Leaving a Conference

A participant should be free to leave a conference at any time, with all other participants being informed. The leaving participant will lose any reservations that he holds, and gives up the "chair" if he is currently the chairperson; a new chairperson can be designated by the system if the chairperson leaves without giving the chair to somebody else. A participant should be allowed to join a conference again after having left, unless his permission to join has been revoked (which might have happened if he was removed from the conference for misbehavior). On leaving, a participant should give the other participants an indication of whether or not he plans to join again and after how long. This is only informative and not binding, because regardless of what he says there is nothing to stop the participant from returning earlier than indicated or after much longer than indicated or never. On returning to a conference, a participant must be supplied with up-to-date versions of shared objects and windows. This is no different than for a participant joining for the first time, except that it might involve less transmission delay if only changes made since the participant left are sent. (This and other optimizations are discussed in the next chapter.)

3.4.10 Conference Termination

"Termination" of a conference causes all activity in the conference to cease. Permission to terminate a conference might be granted to the conference owner or chairperson, or perhaps to any participant when holding a "super-reservation" on the shared space. Since terminating the conference is such a drastic action, it is best to precede it with a

"negotiation" that warns all participants that the conference may be about to terminate. An alternative approach, used in JEDI, is not to give any one participant the power to terminate the conference, but instead terminate the conference only *implicitly* when all participants leave. (Again, a participant who wants to have the conference terminated can use a negotiation protocol to urge all participants to leave. Not all participants may choose to leave, however, and the initiating participant can do no better than leave and let the others continue.) This method requires some timeout mechanism for removing participants whose workstations have crashed, so that the conference does not remain stuck indefinitely with participants who will never issue a command to leave.

Which of the above methods should be used will depend on security requirements. Thus in JEDI we decided that no participant should be allowed to terminate the conference against the wishes of the others, i.e., all participants must leave. On the other hand, on a system where the conferencing program runs with a particular participant's privileges, it seems reasonable to let that participant and no other one terminate the conference.

Once a conference is terminated, some record of it should be maintained. Users who received invitations or announcements generated before the conference was terminated may still be trying to join; if all evidence of the existence of the conference is erased, it will not be possible to inform these users that the conference has terminated. The actual deletion of the conference record, and reclamation of the associated storage, should be a separate action from terminating the conference, that is taken only when it is decided that a record of the conference is no longer needed or that it is not useful enough to justify the associated storage cost. This decision to "garbage-collect" a terminated conference may be made automatically by the system or manually by the conference owner, and may take place any time from a few minutes after the conference terminates to several days later, depending on record-keeping needs and available storage.

3.4.11 Record-Keeping and Review

Independent of real-time conferences, it is often useful to record the evolution of an application object by keeping a sequence of versions or "checkpoints" of the object. Many systems do provide such a facility, and we have included one in our example joint document editor. Functions for using this record are illustrated in JEDI: selecting a particular version for reviewing in "read-only" mode, and extracting information from past versions for use in

operations on the current version of the same or different object. ("Undoing" updates to an object by discarding its current version and reverting to some previous one can be viewed as an extreme example of the latter.) More sophisticated ways of designating the version of interest may be designed, e.g., by date and time or "the version that user X wrote" or "the version I last saw when I looked at this object", or by pointing and selecting from a "directory" or "time line" of available versions. (For a detailed discussion of version reference mechanisms, including branching "alternatives", see [119].) In addition, if a *log* is maintained of incremental updates applied between two checkpoints, or between the last checkpoint and the current "dirty" version, it should be possible to continuously *replay* the updates at some reasonable rate and view the sequence of intermediate states. A replay facility should allow the user to interrupt the replay and pause to view the displayed state before continuing or skipping ahead. The usefulness of such history and review facilities must be weighed against the cost of storage and the complexity of the user commands and access controls (e.g., who can delete which past versions) that must be defined. Such functions have little to do with conferences in particular, but where they are available they should be accessible from a real-time conference as well.

The concept of object history may be applied to a real-time conference itself. (And to individual editing sessions, or private spaces in a conference, although the need to show such history to other users seems less compelling.) Thus, it is possible to design a conferencing system that records not only updates to the main application objects such as documents but also window updates such as scrolling, when participants joined and left, and so on. It is important to define precisely what should and should not be recorded, e.g., recording and retrieving past cursor positions does not seem very useful because one can no longer edit the past document version using the cursor.

The history of a conference might be useful, for example, to a participant who joins the conference late and wishes to see what happened before. The participant might in this case continue to fall behind the conference while he is reviewing its history, a risk he must evaluate for himself. (The ability to quickly stop the review and enter the conference at its current state will of course be important.) Alternatively, a conference participant, say the chairperson or holder of the "super-reservation", could use the shared space itself to review the conference for the benefit of a newly-joined participant. This might be useful to the other participants as a summary and recapitulation; a participant who is not interested in this review might instead

work in his private space and receive a notification when the conference review is complete. This style of review has some important benefits. The participant who was already present in the conference may be able to choose what is important enough in the history to show the new participant, saving the new participant some trouble and time. And, the current participant or participants may use the voice channel to explain what is being shown, perhaps repeating or summarizing some of the past discussion that took place on the voice channel. If a user were to review the record of a conference by himself without the help of some knowledgeable participant explaining it and pointing out important events, that would be about as useful as seeing how the blackboard evolved in a meeting without hearing any of the accompanying discussion. (We assume that the voice discussion itself is not being recorded; even if it were, it has other problems that are discussed below.) Similarly, the history of a given conference can be reviewed in a future conference, with a user who was present at the first conference explaining it to a group of interested users who were not present. (A valid question arises as to whether the act of reviewing a conference should itself be recorded in a conference's history; while this is technically feasible, its usefulness is uncertain.)

An alternative to a complete record of a conference is to allow the state of the shared space to be "checkpointed" in the same way that document versions are checkpointed. (The system might also automatically save the state at points that appear interesting, such as a new participant joining or a long pause in activity on the shared space.) This would make it easier for a user to find the important points of interest for review. A log of updates might still be recorded and made available to those who wish to see more detail; again, the possible usefulness must be weighed against the cost of storage. A useful feature to provide is the ability to *edit* the history at the end of the conference, retaining just those parts that seem worth saving; this might occasionally be used in mid-conference as well.

A slightly different form of record-keeping function is maintaining statistical and summary information about past activity, e.g., which participants held a given reservation and for how long. Assuming the participants do not object to such monitoring, this information can be useful to a chairperson who is trying to be "fair" in granting requests. Such statistical information may also be useful for sociological studies of how real-time conferences are used. (In the latter case, the participants' identities should probably be hidden.)

Record-keeping facilities introduce some new access control questions. Once something has been recorded, who in the future should be permitted to review it? This applies not only

to users currently in the conference, but to users who might later join; depending on the situation, it might or might not be desirable to give a newly-joining participant access to everything that happened in his absence. Editing the history after-the-fact is also important, allowing users to change their minds, but in the case of a conference, where the history is a collective effort not attributable to any one user, the question arises as to who should be given permission to edit what parts of the history.

3.5 Two-Person Conferences

A large proportion of simultaneous interactions between people involve only two participants. While a strict interpretation of the word "conference" might imply a group of three or more, we include two-person conferences in our research because most computer systems offer little support for any group of users, two or more, to interact in real time using shared information.

Two-person interactions in real life tend to be somewhat different from interactions involving three or more people; a participant is only concerned with one other person's perception of what one is doing or saying rather than with several people's perceptions. Similar differences can be expected in an on-line "conference" involving only two users; we describe here a few special kinds of support that the system can provide.

First, operations that require naming and identification of participants are simplified when a participant need only think about, and be informed about, one person other than himself. Thus, when a participant is holding a reservation or the chair and wishes to pass it to the other participant, it is not necessary to explicitly name the other participant; a simpler command with no user-name as argument, the equivalent of saying "Over", can be provided by the system. Similarly, "status" information can be presented in terms of one's own "self" and the "other" participant. For example, if each participant can move a "pointer" over the shared space using his mouse or cursor movement keys, only two pointer "shapes" are needed as shown in Figure 3-1. (This is how the two participants' cursors are displayed in the Palette™ drafting system [85].) When the number of participants N is greater than two, N + 1 different pointer shapes are needed in order to allow each participant to clearly identify his own pointer and at the same time ensure that a given participant's pointer has the same appearance on every participant's screen except his own. (This was illustrated in Figure 2-3.)

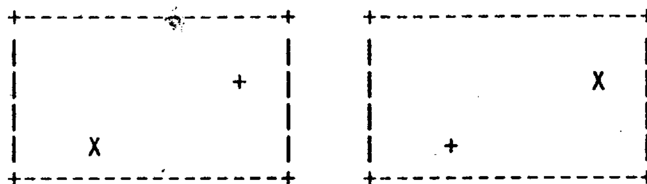

Figure 3-1:Pointer Shapes for Two-Person Conference

Negotiations between two people are also simpler than among a group of three or more. A participant wishing to make a change, e.g., of the shared window size, can propose a change or a range of choices and simply let the other participant make the decision. One round of communication is saved, relative to the negotiation procedure of Section 3.4.4, because once the second participant receives the proposal from the first one, all the information needed to make the decision is available; it is not necessary for the second participant to send his preferences back to the first one before a decision can be made. (This assumes that the two participants have equal power to make the decision; this method cannot be used if only one participant has this power.) This simplification does not work so well when there are three or more people. At best, participants' preferences or votes could be accumulated one by one in some specified order, with the last participant making the decision based on the information received from all the others. This however involves too many rounds of communication compared with the method of collecting participants' preferences in parallel, as in Section 3.4.4.

If the designer of a conferencing system wishes to provide a special two-person "mode" along the above lines, that is different from the interface provided in a larger conference, he must consider the possibility of a two-person conference growing into a larger one when a new participant is added. (It is also possible to design a restricted system that supports only two-person interactions, in which case this question does not arise.) Since increasing the conference size from two to three (or more) represents a significant change in the nature of the interaction, the original two participants may be quite willing to tolerate a change in the user interface, and possibly a brief delay as well if the system needs to reorganize its implementation of the conference (Section 5.3.3). The reverse problem must be considered when participants leave a conference, i.e., should a larger conference revert to a special two-person mode when participants leave and only two are left? Changing modes seems

useful, but may be irritating if a participant who left returns shortly afterward and causes the mode to change back again. In general, a special two-person mode should only be used when there is a reasonable expectation that additional participants (or participants who have left) will not soon be added or try to join the conference. Unless additional participants have been invited and the system is waiting for them to join, the system usually cannot tell whether or not this is the case. It is best to let the two participants decide whether to use a special two-person mode, either by a simple negotiation or by a unilateral decision from one of them.

3.6 Voice Communication

So far, we have implicitly assumed that voice communication in a real-time conference is handled "outside" the conferencing system by the participants themselves making the telephone connections. With voice input and output hardware becoming increasingly available, it may be possible to integrate voice communication with the conferencing system. Such integration will have many advantages, such as:

- A single mechanism rather than two for looking up and connecting with participants.
- A system-enforced correlation between the set of users one is talking with on the phone and the set of users one is interacting with in the shared space.
- Automatic identification by the system of which participant is speaking. (For example, the name of the participant(s) speaking could be made to flash in the "status" window, or a simulation of lip movement could be shown on a displayed facial image.)

Conceptually, a voice connection can be viewed as just another shared abstract "object" that can be "updated" by a participant speaking into his microphone.[10] Concurrency control of this shared object can be implemented, and has been in network voice conferencing systems, by associating a "reservation" and defining some policy for granting the reservation. Permitting only one participant at a time to speak, however, does not allow for interjections such as "hmmm", "I see", or "Wait a minute", which are important sources of feedback in conversation; many people find it difficult to continue speaking with assurance in the absence of any such feedback. This problem can be alleviated by defining limited "side channels" for

---

[10] This conceptual view of course hides the special implementation protocols that are needed for voice encoding, transmission, and decoding; see [20] for details.

such feedback [101], or by not reserving the voice channel at all and letting the participants do their own "concurrency control" (e.g., backing off and repeating their utterances whenever there is interference). Again, the latter is likely to be practical only for small groups.

Recording of the discussion on the voice channel is a potentially useful idea, e.g., for other interested parties to determine what went on, or for later transcription of "minutes". It must be used with extreme caution if at all, because knowing that the discussion is being recorded discourages "off-the-record" comments. Users become particularly cautious about what they say; the discussion, and even the outcome of the conference, suffers as a result. A possible approach to this is to define two different "logical voice channels", one recorded and one not. Then, a participant could choose whether or not he wishes to speak on the record or off, perhaps using a button on his microphone. Thus, a participant could address a comment to the record by explicit command, just as he would when using a voice "mail" system or when performing voice "annotation" of documents or other objects. "Status" information about whether or not recording is on should of course be presented, e.g., in the form of a red light or even periodic beeps. Facilities for excising from the record after-the-fact are also likely to be useful, both on-the-spot (when a participant makes a comment that he would like to erase), and at the end of the conference when it is decided to remove irrelevant discussion and make the record more compact and easier to review. In the absence of facilities for selective recording and for editing the record, replaying the record of the discussion will take too long to be of much use.

3.7 Summary of Design Issues

It is useful to support both *shared* and *private* "spaces" in a conference. Careful attention needs to be paid to which objects, abstract or displayed, are shared among all participants and which are seen only by a single one; a participant's display should be set up so that he can clearly distinguish which is which. The usefulness of private spaces in a conference depends largely on the extent to which the system allows information to be transferred across the boundary between a participant's private space and the shared space of the conference.

Real-time conferencing raises new *access control* issues that have not appeared in traditional computer systems where each action is performed on behalf of, and seen by, some

one user. We have only begun to address these questions, and ours is by no means the last word. But it does appear that reasonable solutions, e.g., to the problem of who can see and update an object in the shared space, are possible. If the conferencing system is constrained in its implementation by an existing conventional operating system, some compromises will have to be made, e.g., having a "chairperson", under whose name the conference system is running, decide when and to whom to give read and update permission.

The basic problem of *concurrency control* in a conference is that a given operation by a participant may have an effect different than intended because of an intervening concurrent operation by another participant, the effects of which the first participant has not yet seen. This problem can be avoided by setting *reservations* on the shared space or parts of it, with an automatic or manual method for managing reservations and requests for reservations. It is possible to allow concurrent activity in a conference by setting reservations on disjoint parts of the shared space. The number of such reservations actually used will in practice be small, because users will have difficulty keeping track of multiple ongoing activities on their screens, and because of the user and system overhead of requesting and granting and releasing reservations. An alternative approach to concurrency control is not to use reservations to prevent interference, but to perform *validation* (e.g., using version numbers) to detect interference when it does happen. This avoids the need for participants to set reservations, but is restricted to situations where it is possible to be "optimistic" that the probability of interference actually occurring is low. We will also see, in the next chapter, that using reservations permits certain improvements in response time that are not feasible unless the probability of concurrent conflict is low.

It may be useful to define participant *roles* in a conference; an example is that of conference "chairperson", whose powers might be defined differently for different conferencing systems. Defining too many different roles is likely to be counterproductive, because it is hard to anticipate all the different ways in which users may wish to interact, and the need to formally switch roles will be a burden on the participants.

It is important to present meaningful *status* information about who is present in the conference, who if anyone is the chairperson, who is working on and looking at what data. Detailed information of this nature will not be needed all of the time; it is best to present a small amount of summary information in a "status window" that a participant can refer to, allowing him to call up more detailed information on request. It is also important to provide

special notification, in a way that attracts attention, of important changes to the conference status.

Interjection facilities, such as sending short messages or causing a designated object to be highlighted, are useful as a means for participants to attract each others' attention quickly without having to go through the process of acquiring a reservation.

Negotiations are useful in selecting a window size for the shared space, or for collecting remarks or votes relating to some application object. All of these negotiations have a common pattern: some participant initiates the negotiation, all participants are notified and their responses collected, and the responses are tabulated for display or processed to generate a set of plausible choices. A negotiation terminates when all participants respond, or on a "timeout" when the participant in charge no longer wishes to wait for a participant who has not responded.

Once a conference has been created, two complementary mechanisms are available for participants to join the conference: direct *invitations* to those participants who are currently on-line and registered with a "name server"; and *announcements* of the conference in well-known databases and directories where interested users are likely to look. A participant should be allowed to leave a conference at any time and join again later, receiving an up-to-date display of the shared space. Permission to *terminate* a conference may be given to just one participant at any given time, or a conference may be terminated only implicitly when all participants leave; the choice depends on security requirements.

"Conferences" involving only two people can make use of special features that are not possible or useful in larger groups, e.g., passing a reservation to the "other" participant without having to explicitly name him, and faster negotiations.

It is desirable to include voice communication in the conferencing system if the hardware and software is available, reducing the user overhead of making telephone calls and providing more complete and useful status information. Similar "concurrency control" methods, using reservations or no control at all, will then be applicable to the voice channel as well. The voice conversation can also be *recorded* automatically, but this has undesirable social consequences. Continuous recording also requires a large amount of storage and makes it harder for a user to later separate the interesting discussion from the irrelevant chatter. Selective recording by explicit command, together with the ability to edit the record, is much

more likely to be useful.

In the next chapter, we will address the problem of implementing a real-time conferencing system that has some or all of the features described in this chapter.

Chapter Four

Implementation of Real-Time Conferences

In this chapter we address the problem of implementing the desired functionality of a real-time conferencing system, such as the joint document editor of Chapter 2, on a distributed collection of workstations and other machines. The main implementation question to be resolved by the designer is the following: How are data and processing distributed or replicated among the workstations of the participants in a conference, and possibly other agents, and how are the data and processing coordinated in order to realize the desired functionality? This chapter addresses this question from the point of view of a single real-time conference that is already running; different implementation organizations and techniques are presented, and their tradeoffs discussed. The overall system architecture for starting and managing conferences, for finding conferences and participants, and for dynamic selection of the implementation method, is discussed in Chapter 5.

Both this chapter and the next use many existing techniques from the areas of computer networks and distributed databases; we will mention these, and point out differences and extensions, as they are presented. Our main contributions are: a review and evaluation of which techniques are suitable for real-time conferences and when, and special adaptations of existing techniques for the particular needs of real-time conferences. The latter include the coordination of multiple participants' workstations rather than just one (which is the usual scenario treated in the literature on interactive distributed computing), and specific ways of trading consistency for improved response time.

4.1 Distributed System Model

In addressing conference implementation issues, we assume a somewhat idealized model of the system implementation environment that will insulate us from considerations relating to the particular technology, operating system, and programming language being used for the implementation. This model is general enough to apply to a wide range of implementation

environments. We do not expect this model to be implemented in exactly the manner described; rather, we view actual system implementations as particular instances of the model.

4.1.1 Sites and Messages

The system is assumed to be a distributed collection of *sites*, where each site consists of a virtual processor and a virtual address space. A site may or may not have its own physical processor, e.g., a given physical processor may be multiplexed among some collection of sites. Wherever it is necessary to maintain the distinction, we shall call a physical machine a *host*; more than one site might reside on a given host. Every site has a unique global *site-address*, which other sites can use in order to send messages (below) to it.

The execution of a site's virtual processor consists of a sequence of *steps*, where each step consists of the following:

1. The site *receives* and reads a *message*.
2. The site performs some *computation* in response to the message, typically modifying the contents of its address space.
3. The site *sends* zero or more messages.

Messages in the above may be sent to or received from the following kinds of agents:

- A *device* that interfaces with the external world, e.g., a workstation screen and keyboard.
- A *site*, including the given site itself. It is assumed that every site can communicate with any site whose address it knows.
- A site may cause itself to receive a *timer* message at some specified later time. Timer and background messages (below) allow a site to perform housekeeping chores, to break up lengthy computations into shorter steps, or to delay acting on a particular message until a more appropriate time.
- Certain messages are spontaneously generated: an *initialize* message is received by a site when it is created (e.g., when the machine's "boot" button is hit or when the host kernel starts up a process in order to create the "site"); and a *background* message is received whenever the site has completed a step and there are no other messages waiting to be processed.

With the exception of "datagram" communication, which discussed in Section 4.6.4, messages sent by a given site (or device) to a given site (or device) are never lost, and are received by the destination site in the order sent. The delay between the sending and the receipt of a given message may be indefinitely long. Messages from different sites to a given

site are received in some arbitrary "arrival" order; this order may be different at different destination sites.

Sites do not share memory; their address spaces are disjoint, and messages are the only means of communication between sites. For programming languages and operating systems that do allow processes on a host to share memory, a given collection of processes can be treated as one or more "sites" depending on how the shared memory is being used:

- *Loosely-coupled* processes, where the amount of shared memory is limited. Shared memory in such a system is typically used in a stylized way that simulates message-passing, with additional efficiency gained by passing pointers to shared memory rather than copying the contents of "messages". Each process in such a system will therefore be viewed as a distinct "site" in our model. Note that this allows multiple user processes on a conventional time-shared system to be included in our model of a "distributed" system.
- *Tightly-coupled* concurrent processes that operate on data that is mostly shared, synchronizing using such mechanisms as monitors [57, 70] or locks [77]. We shall view such a collection of processes as a *single* site, with the computations of the different processes broken up into short "critical sections" that are executed one at a time.

4.1.2 Object Model

We next describe a particular model of object structure that allows sites to refer to objects stored at other sites with which it communicates; this is necessary in order to permit useful distributed computation, such as real-time conferencing, despite the lack of shared memory. The model describes object structure as viewed "externally" by other sites; a given external object structure may be implemented internally within a site in any way that achieves the same functionality.

A *structured object* is a collection of *components*, each of which is a *<selector, value>* pair; no two components of a structured object can have the same selector. A component value may be an *immediate* value, such as an integer or string, or another structured object; objects can thus be hierarchically structured. Different kinds of structured objects allow different kinds of selectors, for example:

- The selectors in an *array* are integers in a given range.
- The selectors in a *record* are fixed strings representing the "fields" of the record. (Because its selectors are known in advance, a record may in fact be represented internally as an array-like object, allowing more compact transmission and

storage of selectors using integers.)

- The selectors in a *table* are arbitrary strings. A table may in addition be *ordered*, allowing sequential access as with an array but without requiring that the selectors form a consecutive integer sequence.

Variations and combinations of the above are possible, e.g., two-dimensional or multi-dimensional arrays where the selectors are pairs or tuples of integers.

If a selector is used when accessing a given structured object such that the object has no component with the given selector, we shall treat this as if the object does have a component with the special value *Undefined* associated with the selector. This takes care of situations where an illegal or nonexistent selector is used by accident. It also allows addition (or creation) and removal (or deletion) of components to be treated as special cases of updating the value associated with a given selector. Adding a new component, with a given selector and value, is equivalent to updating the component from its previous value of Undefined to the given new value, while removing a component is equivalent to updating the component from its current value to a new value of Undefined. The special value Undefined can also be explicitly stored in a component of a structured object, and tested for equality. This may be useful, for example, if after the deletion of a component it is desired that the same selector never be used again, or be used only with a higher "version number" (see below) than any previously used.

Each site maintains a *naming context*, which is a structured object whose components are objects that were created at this site. (A site's naming context can have any of the forms described above.) The site at which an object was created is called the object's *birth-site*. The only objects that can be referred to in messages are those that are contained in some site's naming context. A *unique-id* (unique identifier) is a sequence of selectors, starting with a site-address, that specifies the "path" traversed, through the components of the naming context of the given site, to reach a given object. Unique-ids can thus be sent in messages in order to refer to the object in question, and can be stored in objects as a form of abstract "pointer". (We do not consider internal pointers in this model, because they cannot be meaningfully transmitted outside a site's address space.) It is not always necessary to store the complete unique-id in order to refer to a unique object; leading selectors of the unique-id can be skipped if they are implied from the context in which the object reference is being stored or transmitted.

A site may have any number of naming contexts associated with site-addresses other than its own, each of which contains information about objects created at that other site which are somehow of interest to this site. A site may for example keep a *copy* of an object, or parts of a structured object, from another site; we shall see, for example, that the workstation of a participant in a conference keeps copies of parts of the shared space for display to the participant. Note that any naming context that a site associates with some other site's address is in effect a partial "copy" of that other site's own naming-context.

The *home site* of an object is the unique site where the most up-to-date copy of that object is available; this copy is called the *primary copy* of the object. Only the home site can initiate updates to the object; other sites update their copies only in response to update messages received from the home site, or known to have originated at the home site. (Update messages received may sometimes be forwarded by other sites.) Sites other than the home site may "request" the home site to perform an update by sending an appropriately encoded message; whether, when, and how the home site decides to act on these requests is arbitrary and not specified by this model.

We shall assume in this chapter that the home site of an object is *fixed*, i.e., is always its birth-site. An extension to this, that allows an object to "migrate" to a different home site, is presented in Chapter 5.

When the home site of an object "crashes", the object becomes inaccessible for update. Other sites may hold copies that can be read; some applications may in addition allow "alternatives" (below) to be generated by updates to these copies. When a site recovers from a crash, objects that it held prior to crashing may or may not be available. If the site saved an object on permanent storage (e.g., disk), the object will be available again on recovery. On the other hand, objects in volatile storage will be treated as having been "deleted" (i.e., set to Undefined) without warning when the site crashed. In some cases, the copy saved on disk may be older than the volatile copy; a crash is then treated as causing recent changes to be "undone" and this older version restored.

In addition to, or instead of, a copy, a site may associate various kinds of *bookkeeping* information with an object, both in its own naming context and in the naming contexts of other sites. (In some cases a site might maintain bookkeeping information and no copy; we shall treat this case as if the site has a "copy" with special value Undefined.) The various kinds of

bookkeeping information are described below.

4.1.2.1 Agreements

A site may often enter into an agreement with another site to keep the other site informed of changes to a given object, using some specified class of update messages (described below). Such agreements, as we shall see, are how real-time conferences are implemented. Information about such agreements, to send or receive updates, must therefore be remembered by the sites in question.

4.1.2.2 Version Numbers

A monotonically increasing *version number* may be associated with an object to keep track of updates to the object. Version numbers may be of several kinds. A *sequential* version number is one that is incremented by one each time the object is updated. A *timestamp* is a version number drawn from a clock or counter that is associated with a collection of objects, or with a site as a whole, or with the entire distributed system, such that the same timestamp is never assigned twice within its associated context. (In the case of system-wide timestamps, global uniqueness is ensured by appending a site's address to the value of its local clock or counter, and the local clocks of different sites may be kept approximately synchronized using a mechanism such as Lamport's [69].) Unlike a sequential version number, the timestamp of an object will in general increase by more than one with each update. Timestamps therefore will require more bits to store than sequential version numbers, but are useful when it is important to determine the relative ordering of updates to different objects; the latter cannot be determined using sequential version numbers that are independently assigned to different objects. Which kind of version number is used for a particular object will depend on how the object, and other related objects, is to be used.

The version number associated with a given site's copy of a given object may in general be "behind" the version number on the home site's copy, because updates have not been sent or are in transit. If a site does not have a copy of a given object, we shall treat this as if the site has a "copy" having value Undefined and special version number *zero*. Thus, every object has version number "zero" prior to its creation within a given parent object; the version number (or timestamp) of a component object that is not Undefined must always be greater than zero.

4.1.2.3 Update History

An object *update* is a description of an update operation and arguments that, when applied to some specified *old* version of the object yields some other specified *new* version of the object. Updates can be transmitted in messages, allowing the receiving site to update its copy if it has the appropriate old version, and can be stored in "histories" (below).

The most common type of update is a *relative* update which describes the update operation that was actually applied to the specified old version of the object, by the home site at the time, to yield the new version. However, the operation in an update description need not always be the one that was actually applied, and the new version need not always immediately follow the old one. It may be possible, for example, to compute a single update operation that when applied to a specified old version of an object will yield a specified new version several updates ahead. In the extreme, an *absolute* update is one where the "old" version, to which the update operation applies, is version number "zero". Such an update carries a complete description (possibly compressed) of the specified "new" version. The "creation" of an object, and the recording or transmission of an object's current contents, are treated as instances of absolute updates. "Deletion" of an object is also a form of absolute update, where the "new" value of the object is Undefined.

It may sometimes also be possible to derive *inverse* updates where the "new" version is actually older than the "old" version; these can be used to "undo" changes to an object in a more efficient way than restoring a complete copy of the old version.

An object's *history* is the sequence of versions and updates that it went through. This history is an abstract concept; a *stored history* will in general contain some subset of the complete abstract history, perhaps with gaps. A stored history may contain only complete copies or "checkpoints" (which are in fact "absolute" updates) of particular versions, or only "relative" updates, or some mixture, and may or may not contain "inverse" updates. A stored history may also contain additional access structures for finding versions and updates based on various criteria (such as time of update and user), so as to implement the various "review" functions described in Section 3.4.11. There is no one unique site where "the" history of an object is stored; different sites may store different histories for the same object, for different purposes and with possibly different methods for finding the stored history for a given object.

4.1.2.4 Alternative Versions

Since only the home site of an object can initiate updates to the object, only the home site issues version numbers. In some cases, a site other than the home site may update its copy on its own in anticipation of the home site approving the update. The site's version in this case is referred to as an *alternative* version. The site will assign its copy an alternative version number that is distinguishable from any bona fide version number that the home site might issue. This site will usually remember which true version of the object the alternative version was derived from; this is part of the alternative's "ancestry", below. It may also remember what update, or sequence of updates, was applied to the original version in order to yield the alternative one, or an "inverse" update that it can apply to the alternative version to get back the original true version.

4.1.2.5 Equivalence and Ancestry

Often two versions of different objects are known to have identical values, e.g., when the value of one is copied into the other. This may even happen with different versions of the same object, e.g., when changes are "undone" by making a past version the current version (with a new version number) or when "promoting" an alternative version by giving it a bona fide version number. Remembering version equivalences is useful in allowing sites to determine when they already have some data that they need, thus avoiding costly retransmission of the same data from another site. It is also useful when the need to "merge" similar objects (e.g., documents or conferences) arises; knowledge of a common "ancestor", found by tracing back through the histories of the two objects until equivalent versions are discovered, will often allow the merging to be automated or otherwise optimized.

4.1.2.6 Immutability

In some cases, such as termination of a conference, an object reaches a "final" state such that it will never be updated further; it can be marked as *immutable*. Sometimes the reason for marking an object immutable is that there is some other object that "supersedes" it; an example we shall see in Chapter 5 is when two conferences are to be "combined" such that one conference terminates and the other one continues. For such situations it is also useful to remember which object supersedes the now-immutable object, so that sites that try to access the immutable object can be directed to the object that supersedes it. (The superseding object may in turn remember the given immutable object as part of its

"ancestry", described above.)

4.1.2.7 Derived Values

The value of a given object version may often have been computed by applying a given function (free of "side effects") to some argument objects and values, e.g., the current contents of a displayed "window" may have been computed functionally from a given document version and region coordinates, or an object code module may have been "compiled" from a source code module using certain parameters. If the function and the argument object versions are remembered, this can be useful in avoiding recomputation of the derived value, or in allowing a new value of the object to be recomputed "incrementally". (Details of whether and how incremental recomputation is performed are dependent on the application and are outside the scope of this model; see [105] for an example in the domain of software management.)

4.2 Real-Time Conferences

In the terms of the above model, a real-time conference is simply a structured object representing the "shared space" of the conference, together with an agreement to replicate parts of this object at the *workstation* sites representing the participants of the conference.

4.2.1 Conference Objects

An example of a conference "object" is the type definition JEDI-CONF in Figure 4-1. This specifies the shared space of the joint document editing system of Chapter 2 as a Pascal-like "record" [63] with several components, some of which are themselves structured object types. We have taken some liberties with strict Pascal syntax and semantics, such as dynamic definition of a type based on run-time values. For example, the "subrange" type **1..npartcs** allows integers from one through the current value of component **npartcs**, which specifies the number of participants. (Not all of these "participants" are necessarily in the conference at the same time; the array **ws-statuses** specifies which participants are "Active" and which are not and which are in the process of joining.) This subrange definition is meaningful only within the context of a given object of type JEDI-CONF, and may even change within that context (e.g., as participants are added). Dynamic subrange types thus allow us to define

```
type LINE-ID = int

type RECT-COORDINATES = record
    y: int
    x: int

type RECT-SIZE = record
    height: int
    width: int

type JEDI-CONF = record

    "description", released outside conference
    controller: site-address
    conf-id: int                    unique in controller's
                                    naming context
    conf-type: "JEDI-CONF"          see Section 5.2.1
    subject: string
    owner: user-name
    allowed-users: list of user-name
    public-access: (Yes,No,Request)
                                    enumerated type
    estd-start-time: date-time      estimated
    estd-end-time: date-time
    start-time: date-time           Undefined = not started
    end-time: date-time             not Undefined = terminated
    npartcs: int NONDECREASING      size of participant-related arrays
    chair: 1..npartcs               chairperson's participant number
    partc-names: array[1..npartcs] of user-name
    ws-addresses: array[1..npartcs] IMMUTABLE of site-address
    ws-statuses: array[1..npartcs] of (Active,Maybe,Quit)
    ndocs: int NONDECREASING        size of working-set
    file-servers: array[1..ndocs] of site-address
                                    file server for each
    doc-names: array[1..ndocs] IMMUTABLE of string
    replic-line-limit: int          internal
    super-window-size: rect-size

    continued
```

Figure 4-1:Conference Object Specification for JEDI

```
documents
working-set: array[1..ndocs] of DOC-INFO
                                   declared below
replic-line-count: int             number currently replicated
proposed-line-limit: int           Undefined if no negotiation
ws-line-limits: array[1..npartcs] of int
                                   each Undefined if no response

shared "superwindow"
proposed-size: rect-size           Undefined if no negotiation
ws-size-responses: array[1..npartc] of rect-size
super-reservation: 1..npartc       Undefined = free
super-requests: list of 1..npartc
                                   queue of requests
visible-windows: list of record
    doc#: 1..ndocs                 index into doc array
                                   or Undefined = blank window
    window#: 1..(working-set[doc#].nwindows)
                                   index into window array
                                   for given doc
    position: rect-coordinate      relative to
                                   top left of super-window
pointers: array[1..npartcs] of rect-coordinate
                                   participant's pointers
ptr-shapes: array[1..npartcs] of bit-pattern
ptr-frequency: int                 maximum rate, per second

miscellany
lookup-servers: list of site-address
current-time: date-time            periodically updated
line-width: int
search-pattern: string
paste-buffers: array[1..npartc] of string
                                   copy of each
                                   participant's paste buffer
ws-invited: array[1..npartc] of
            (Invited,Removed,Undefined)
ws-requested-to-join: array[1..npartc] of
            (Requested,Declined,Undefined)

continued
```

Figure 4-1, continued

```
type DOC-INFO = record
    information about each document
    saved-v#: int                          last checkpoint
                                           zero = removed
    saved-timestamp: date-time
    changes: int                           since last checkpoint
                                           zero = "clean"
    read-only: boolean
    reservation-timeout: date-time
                                           if reserved at server
    who-can-write: array[1..npartcs] of boolean
                                           determined from server
    contents
    lines: list of record                  at least one,
        id: line-id                        in ascending order
        contents: string                   mutable, variable-length
        replicated: bool                   workstations have copy?
        redundant - cursors and marks in this line
        cursors: list of 1..nwindows
        marks: list of 1..nwindows
    shared windows displaying this document
    nwindows: int
    windows: array[1..nwindows] of DOC-WINDOW
                                           declared below
    participants' private windows
    private-windows: array[1..npartc] of list of record
        start-line: line-id
        height: int
        end-line: line-id

type DOC-WINDOW = record
    size: rect-size                        height and width
                                           zero = not displayed
    start-line: line-id
    end-line: line-id                      implied by above
                                           and document contents;
                                           NOT = start + height-1
    start-column: int
    cursor and mark
    cursor-line: line-id
    cursor-column: int
    mark-line: line-id
    mark-column: int
    command-in-progress: parse-tree
    feedback: string
    conference only
    reserved-by: 1..npartcs                Undefined = free
    reservation-timeout: date-time
```

Figure 4-1, concluded

arrays of variable size, such as `partc-names` and `working-set`. The declaration of a given object component (e.g., `chair`) as being of one of these subrange types clearly distinguishes the different uses of integers as indices into the corresponding array(s). Such indices also allow for more compact reference in messages, using the index number of a participant or document instead of its full name.

We have added some additional qualifiers, such as NONDECREASING and IMMUTABLE, to indicate how certain components of the conference object are being used. The qualifier MANDATORY is used for components that cannot have the value Undefined.

We emphasize that the data type specification of Figure 4-1 is meant to be an "external" view of the conference object, suitable for transmission, in whole or part, between sites (as is the "xrep" of Herlihy and Liskov [54]). The data type will typically be implemented, within a site, using some variations and enhancements for efficiency (such as a tree or linked list for the sequence of lines in a document). The designer will also have to translate the abstract specification into a programming language that may not support all of the features assumed above, such as dynamic arrays and explicit manipulation of the value Undefined. Note that there are no "pointers" in the specification. Instead, all references to objects not directly contained use indices into arrays, or "ids" in the case of document lines. This, again, allows a site to refer to a component of an object at another site by including an index or id in a message. Internally, however, a site may "cache" a pointer to the designated object for faster access, at the cost of keeping the pointer consistent on update (which might never happen, especially for IMMUTABLE array elements). The data type JEDI-CONF already "caches" some redundant information that must be kept consistent, e.g., the cursors and markers in each line of a document; these are also in the form of ids and not pointers.

We will refer to the specification of JEDI-CONF again in this chapter, as we use this example to illustrate our implementation techniques. First, we briefly describe the document data structure that is being assumed. Every line in a document is assigned a line-*id* that is unique and mostly unchanged (with exceptions as noted below) for the lifetime of that line within that document. Line-ids are *ordered* in that an earlier line of a document has a smaller id than a later line, but line-ids are in general *not* consecutive: there will almost always be a "gap" between the line-ids of two successive lines, to allow possible insertions of new lines without changing any line-ids. This means that cursors and windows can "point" to a given line using its line-id without having to be updated when preceding lines are inserted or

deleted; this contrasts with the use of consecutive line numbers which do change on the insertion or deletion of lines. (A reference to a given line will of course have to be updated if that line is deleted. If there is an insertion or deletion within a line, only the offsets of some cursors in the line need be updated, not the line-id.) Reducing this update overhead may or may not be an important consideration within a given site, but is especially significant when references to lines may be held by other sites.[11]

LINE-ID	CONTENTS
10	The first line
20	The second line
40	The third line
45	The fourth line
47	The fifth line
50	The sixth line
53	The seventh line
56	The eighth line
65	The ninth line
70	The tenth line
80	The eleventh line
85	The twelfth line
95	The thirteenth and last line

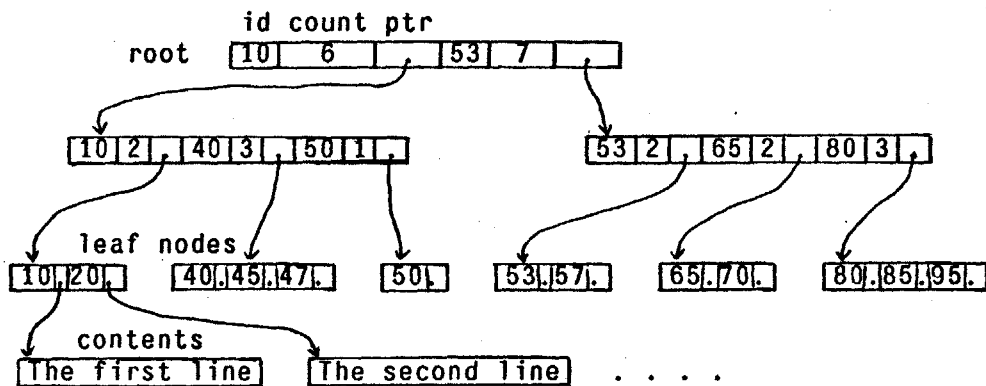

Figure 4-2: Example Document Structure

[11] An alternative method that avoids updating of references to lines is the use of *pointers* to lines in a linked list. While this does have the desirable property that line references do not have to be updated, internal pointers are not suitable for remote reference by other sites. It is also not possible to tell which of two lines is "earlier" simply by comparing pointers.

To *find* a line in a document given its id, any reasonable ordered search structure may be used. Figure 4-2 shows a short sample document and a "B-tree" [5, 21] implementation. (Note that the use of line-ids is reminiscent of editing and programming with "line numbers"; the line-ids in our case, however, are managed automatically by the system and are never visible to the users.) Each node in the tree has an ascending sequence of line-ids, and a pointer associated with each such that the pointer identifies a subtree containing lines with ids between the given line-id and the next one. A *count* of the number of lines in each subtree is also cached in every node, allowing the next or previous or Nth line relative to a given line (or relative to the beginning or end of the document) to be quickly found. This is needed, for example, in order to determine which lines are to be displayed in a "window" of given height starting at a given line-id; it could alternatively be implemented by doubly-linking the nodes containing the line contents. Deleting lines from the document by removing them from the tree is straightforward; cursors and windows that refer to a deleted line must of course be updated to refer to the last preceding undeleted line. Inserting one or more lines between two successive lines *id1* and *id2* involves assigning ids to these new lines from the range between *id1* and *id2*, and inserting these lines into the tree at the proper position (updating counts, and possibly splitting nodes of the tree if necessary). If the gap between *id1* and *id2* is smaller than the number of lines to be inserted, then one or more lines in the vicinity of the insertion will be *renumbered*, i.e., assigned new line-ids, to make room; cursors and windows referring to the affected lines will have to be updated in this case. We assume that line-ids use a fairly large number of bits and that some reasonable heuristic is followed for assigning ids to inserted lines, so that renumbering of lines occurs rarely if at all.

4.2.2 Controller and Workstation Sites

The "home site" of a real-time conference object is referred to as the conference *controller*. We assume in this chapter that a conference does not migrate from its birth site; movement of a conference to a different controller site is discussed in Chapter 5. (Different conferences may of course have different controller sites.)

Each participant in a real-time conference is represented by a *workstation* site. (For participants having terminals with no local intelligence, the workstation "site" is the corresponding process on whatever host the terminal has been connected to.) Each workstation maintains copies of some or all of the components of the conference object, and

receives *update* messages from the controller as the components are updated; these updates are then reflected on the participants' display screens. Commands entered by a participant to operate on the shared space are forwarded by his workstation to the controller in the form of *input* messages; the controller processes or rejects input messages according to the desired access and concurrency control criteria for the given conference. Except as noted in Section 4.7, workstations do not communicate directly with each other; all communication within a conference is between the controller and individual workstations only.

We will assume for most of the discussion in this chapter that the controller site in a conference is a distinct site from the participants' workstations, and that the controller site is equipped with sufficient processing power and storage (real or virtual) to manage the activity in a conference. (The controller's ability to do so will of course degrade as the number of participants in the conference increases.) The participants' workstations may not in general be so well equipped in which case relying on a separate site to provide a conference "service" is appropriate. Using a separate controller site allows a conference to continue independent of individual participants leaving the conference, and makes the conference less vulnerable to crashes of individual participants' workstations. A conference is, however, vulnerable to crashes of the controller site itself. We will assume that the controller site is in fact more reliable than any workstation, and is unlikely to crash during a conference (which might typically last from several minutes to an hour, seldom longer). We will describe crash recovery using "backup controllers" in Chapter 5.

Using a separate controller site for a conference does involve an extra site in the conference, with added communication complexity and overhead (e.g., three-party communication for a two-person conference) that should not be necessary if one or more participants' workstations do in fact have sufficient processing power and storage, and acceptable reliability, to act as conference controller. We therefore allow for a given site in a conference to play the role of *both* controller and workstation, if it is capable of doing so. The functions of controller and workstation must be suitably multiplexed on the site's real or virtual processor, and "messages" between the two roles simulated. In such an arrangement, the participant whose workstation is also the controller will be "favored" in that he will receive more immediate response to his commands than the other participants. In many cases, this may in fact be quite reasonable, e.g., if the given participant is the conference "chairperson". If it is not desirable to favor one participant over the others just because his workstation

happens to be the controller, the controller may introduce short artificial delays, approximating the message transmission delays from the other workstations, before processing the given participant's commands. This is discussed in more detail below in Section 4.3.2.

While having a participant's workstation play the role of conference controller allows a conference to be held with one fewer site, the given participant cannot leave the conference and use his workstation for other purposes. We will describe how the controller of a conference can be moved to a different site, thereby allowing the controlling workstation to leave, in Chapter 5.

In discussing how the controller and workstations interact, we assume that a real-time conference has already been set up, with the controller and workstation sites already running the software needed for the given conference. The problems of locating controller and workstation sites, and of finding and starting up the necessary software, are discussed in Chapter 5.

4.3 Input and Update Processing

This section describes how participants' workstation sites generate, and the controller responds to, "input" messages describing the participants' commands, and how the controller generates and the workstations respond to "update" messages describing the effects of these commands. Different kinds of input and update messages, and their relative merits, are discussed. What kinds of input and update messages can be used will in some cases be constrained by what information from the conference object is "replicated" at the workstations. While replication strategies are discussed in Section 4.4, the two issues are in fact closely interrelated and we will make occasional reference below to how the replication strategy influences the processing of input and update messages.

4.3.1 Workstation Interface

A participant's workstation site is responsible for displaying information to him on his display screen and for dealing with actions that he performs on physical input devices such as the keyboard and mouse. (These functions are often referred to as "window management" and "command parsing", respectively.) Some of the participant's actions will have purely

local effect, in that a command is invoked on locally stored data (e.g., the participant's "private space") and the screen updated accordingly when the command completes. Other actions may be directed at the "shared space" of a conference in which this user is participating. The participant's workstation may also occasionally have to deal with more than one conference at the same time, e.g., if while in one conference the workstation receives an invitation to another conference, it informs the participant who may then decide to leave the first conference and join the second one. (The participant may even try to combine the conferences; this is discussed in Section 5.2.6.)

The participant's actions on the shared space of a conference must be translated by the workstation into input messages that are sent to the controller site for processing. The workstation will also receive update messages from the controller describing changes to the shared space that resulted from the actions of the conference participants (including this participant). The workstation processes each update message by updating its copy of the shared space and then updating the screen (or the "viewport" of the screen allocated to the shared space) to show the effects of the update. The workstation is responsible for determining whether a participant's physical actions are intended to operate on local or remote data; this it does by providing commands for the participant to "select" a window or "mode". The workstation may also prevent the participant from using certain kinds of commands depending on various conditions, e.g., if the participant does not hold a "reservation" that is needed in order to enter commands in a shared window; this will save the controller site the burden of having to respond to input messages that it must reject.

For participant actions that are directed at the shared space of a conference, there are many ways in which the workstation could send these to the conference controller as input messages. At the lowest level, physical keystrokes and mouse actions might be forwarded to the controller for parsing into operations on the shared space. This is the approach taken by "virtual terminal" protocols, and is best suited for situations where the participant's action have little semantic content and do not need much parsing, e.g., moving a mouse "pointer" over a shared window. For higher-level operations, however, remote parsing of keystrokes by the controller has the problems already described for virtual terminals in Section 1.6. The PASTE command in JEDI, for example, cannot be implemented by remote parsing at the controller because it requires locally stored application data (the participant's "paste buffer") from the participant's private space. (Such information transfer could be implemented with

remote parsing if the controller manages not only the shared space but also every participant's private space as well. We assume that the need to parse every command entered by every participant, on private spaces as well as the shared space, will almost always impose an unacceptable burden on the controller site.)

If input messages instead take the form of higher-level application operations, then the workstation can include application objects from its private space as arguments in operations that it sends to the controller. Thus, a PASTE command is translated into an Insert-String operation with the contents of the paste buffer as argument. The workstation thus takes some responsibility for parsing the participant's physical actions into operations on the shared space. For longer "commands" that involve a sequence of physical keystrokes and/or mouse actions, e.g., SELECT 3 WORD, parsing may be done by the workstation, by the controller, or a combination of both; how this is done will depend on the interface requirements and on whether any private information must be included as an argument of the command. In JEDI, for example, the interface specification requires that such commands be "echoed" in all participants' copies of the shared space; therefore, some form of input must be sent to the controller as the participant is typing the command. This might take the form of a stream of characters, or high-level operations on a "parse-tree" that is constructed as the command is typed. Again, the higher-level approach is more flexible because it allows for private application objects to be transmitted should that need arise. If instead command echo were to appear only on the given participant's screen, the workstation could parse the entire command locally and send only the resulting operation when the participant completes the command (with a terminating key such as ENTER); in this case, the other participants will not be aware that this participant is doing something in the shared space.

A participant's workstation will in general wish to keep track of which of the operations it sent in input messages were actually disposed of (either executed or rejected) by the controller. This it can do by labeling each input message with a unique identifier, such as a sequence number, and having output messages from the controller (whether results of execution or error messages) identify which input messages, if any, they are the results of. Thus, if an operation remains outstanding for a very long time, the workstation can inform the participant thereof, who may then decide to leave the conference if he does not want to keep waiting.

The workstation may also allow operations to be *typed ahead* by the participant, in that

more than one operation may be outstanding at any given time. Thus, if the participant is typing quickly and each keystroke (or burst of keystrokes) results in an input message, the participant is not forced to wait for the results of one input message before he can enter further input. In some cases, however, the workstation may need to wait for the results of a given operation to be received from the controller because some subsequent operation by the participant depends on these results. For example, suppose a participant enters a COPY (or DELETE) command in the shared space followed by a PASTE command in his private space. The private PASTE command is executed locally, and does not involve the controller, but in order to have the desired effect it cannot be executed until the text copied from the shared space is received from the controller. This may involve a short wait that may or may not be perceptible to the user; in any case, if the workstation has to wait too long the participant always has the option of leaving the conference.

4.3.2 Input Processing

The action taken by the controller on receiving an input message from a workstation will depend not only on the particular operation requested but also on whether the given participant currently has permission (e.g., in the form of a reservation, in addition to his overall access rights) to perform the given operation. A workstation will filter out and not transmit operations that the participant is not allowed to perform, based on its current knowledge of access rights and reservations, but when the controller takes away a participant's reservation there will be a transient state during which the workstation, believing it still holds the reservation, will continue to forward operations to the controller. The controller must therefore verify every input message that it receives against the true state of access controls and reservations.

The simplest controller strategy is to process (i.e., execute or reject) input messages as they are received from the different workstations. This raises the question of *fairness*, in that if a given workstation is experiencing longer communication delays than the other workstations, then its input operations will take longer before they are processed. In the absence of preventive methods such as reservations, intervening operations from other workstations may cause this workstation's operations to have a different effect than intended, and may even cause many operations to be rejected if validation (described below) is being performed. If it is important to give fair treatment to every participant's input, the controller

may artificially *delay* acting on input messages from a fast workstation by a small amount of time, based on some estimate of the difference in transmission delays, thus giving operations from the slower workstation a better chance of succeeding. (If the controller is itself also a participant's workstation, it is treated as an exceptionally "fast" workstation with close to zero message transmission delay.) This will of course cause the response time observed by all workstations to degrade to that of the slowest workstation; if this is unacceptable, it is best for the controller to remove the slow workstation and its participant from the conference altogether. In some cases, the controller might deliberately bias its processing of input in favor of a particular workstation, especially if one participant is given power of "chairperson". (This might be an argument for making the chairperson's workstation the conference controller site, provided the workstation can store a complete copy of the shared space and can perform all processing of operations. If the chair is passed to a different participant, e.g., when the chairperson leaves the conference or wishes to relinquish the chair, the controller site can be moved as well; this is described in Section 5.2.4.)

Concurrency control, as described in Chapter 3, can be performed using reservations, or by *validating* the assumptions made by the operation at the time the participant entered it, e.g., the version number of the shared space, or of the particular object(s) being operated on. Implementing validation, if that is the approach chosen by the designer, requires that update messages carry version numbers for the objects that they update, so that input messages requesting operations can report the version number of the copy that the workstation currently holds. The controller can then compare the version number carried by an input message with the current version number of the object, and reject the operation or issue a warning if the version numbers differ, whichever method the designer has chosen. More sophisticated validation, based on whether or not the intervening concurrent operation actually conflicts with an incoming operation, can be performed if the controller maintains a history of recent operations on the shared space.

Validation of input messages using object version numbers can interfere with *type-ahead* in a subtle way. Suppose a user is typing quickly and that each character typed causes an operation (e.g, to insert the character into a document) to be sent to the controller in an input message. Let us assume that the version number of the object being operated on, say a document, is V1 at the time the first character is typed, and that the second character is typed before the workstation receives the result of the first operation from the controller. Then, both

the first and second input messages will specify V1 as the version of the document that the participant saw at the time he entered the operation. But when the second input message arrives at the controller, the first input message has already been processed and has caused the document version number to be incremented to V2, say, causing the second input message to be rejected because it was based on an obsolete version of the document.

Because the second input message above was sent before the results of the first were seen, the controller took this to mean that the two operations conflicted. But because the two operations were entered by the same participant, who in his mind assumed that the first operation would succeed when he entered the second, this ought not to be construed as an error. To avoid this problem, the controller must retain some history information indicating which participant's operation resulted in which version of the object, so that it does not reject an operation that "conflicts" with a preceding operation by the same participant. (The alternative, of a workstation only allowing one input message to be outstanding at a time, is unacceptable because it does not allow participants to type ahead.)

In many interactive systems input typed ahead is discarded if earlier input caused an error that the user has not yet seen, on the assumption that the user would have wanted to do something different if he had known that an earlier operation had failed. This can be implemented as well, if the controller remembers not only who entered which operation but also whether or not each operation generated an error; it can then determine whether or not the error message was seen at the time a given operation was entered, and act accordingly. Note that if the controller is recording a history of operations on the shared space, all of the information needed for validating input and for accepting or rejecting typed-ahead input is readily available.

4.3.3 Update Generation

As the controller processes input messages, it sends the results to all workstations in the given group in the form of update messages which each workstation interprets and applies to its copy of the replicated objects. Update messages can be generated in various forms, and the different choices available are described below.

Different *operations* may in general be used in reporting changes to a replicated object. For example, consider the Fill-Polygon operation on a bitmapped image. This is typically

implemented by applying one of several "scan-conversion" algorithms [31] that determine what coordinate ranges are covered by the polygon on each horizontal scan-line of the bitmap, and then writing into the bitmap one scan-line at a time. If the bitmap is to be replicated at a group of workstations (so that the participants can see it on their screens), the designer has one of two choices: the controller could send the Fill-Polygon operation and arguments as is and let the workstations do the scan-conversion; or, the controller could do the scan-conversion and send the workstations the resulting sequence of Write-Scanline operations. The choice should depend on the workstations' capabilities. Sending a higher-level operation (such as Fill-Polygon) is more compact and saves bandwidth but requires that the workstations be capable of performing the higher-level operation. If instead a workstation is not capable of doing scan-conversion, then lower-level operations must be sent, requiring that more bits (or packets) be sent and degrading performance somewhat.

Update messages may or may not carry some form of *version number*, sequential or clock-based (a "timestamp"). Version numbers are not strictly needed when reliable sequenced transmission is used, because a workstation will always have the correct previous version to apply the next update to. (Supplying an initial version prior to transmission of updates is discussed in Section 4.5.) However, version numbers must be transmitted with update messages if validation of input messages based on version numbers is to be performed. (With unreliable datagram communication, which we describe in Section 4.6.4, version numbers are needed so that a workstation can detect duplicate and out-of-order update messages.) Transmitting version numbers of course incurs added communication overhead, which can be minimized by making them as compact as possible, i.e., assigning them sequentially and using only as many bits as are necessary to make the probability of "wrap-around" of the version number space, and the possible confusion that may result from assigning the same version number twice, small enough that it can be ignored. (This is in fact a standard technique for assigning message or packet sequence numbers in computer communication protocols.)

Version numbers are also useful when communication is performed via "third parties", so that message sequencing guarantees no longer hold. For example, we shall see cases where a workstation may get a copy of an object from some site other than the conference controller, such as a "file server" or "lookup server". Having obtained a copy from a server, the workstation can then report the version number on that copy to the controller in order for

the controller to determine whether any subsequent updates, or a new value, needs to be sent. If communication bandwidth is scarce, one option available is to exchange and verify version numbers only when establishing that the workstation has a correct initial version, and then not include version numbers in subsequent update messages.

When the controller executes an operation on the shared space, an update message may be generated *immediately*, either in the form of the high-level operation that was just applied, or translating it into one or more lower-level operations if the protocol requires that.[12] Or, update messages may be *deferred*: the object, or affected components of it, is marked "dirty" when updated. Changes to dirty object components are then sent to the workstations periodically (with a high enough frequency to maintain reasonable interactive response) or when a sufficiently large number of changes accumulate.[13] Changes to dirty objects may be sent in *absolute* form, carrying the complete new contents, or as *relative* operations computed by comparing with the value of the object when it was last sent to the workstations. Sending relative updates will in general require less bandwidth because such updates are more compact than absolute ones, but will require that the controller remember some information about the previous value of objects and have an efficient comparison algorithm such as the various "incremental redisplay" algorithms used in many text editors [43, 111].

Immediate transmission of update operations is simpler in terms of the software required, and also gets the updates to the workstations more quickly. Deferred updating can still be useful, especially when bandwidth is scarce and the effects of a sequence of operations can be "compressed" into a single more compact operation, e.g., a character typed by the user is quickly erased. (The case of limited bandwidth is exactly the situation where incremental redisplay is most beneficial in text editing.) In all cases, generation of updates should not be deferred for so long as to degrade response time.

---

[12]The sending of lower-level operations may be done in tandem with the execution of the high-level operation at the controller; e.g., the code that implements Fill-Polygon will both update the controller's bitmap copy and send a Write-Scanline operation as it generates each scan-line of the polygon.

[13]Note that deferred generation of updates is different from buffering and batching of already-generated updates that may be invisibly performed by lower-level communication software.

4.3.4 Synchronization Points

At various points in a conference, the controller may issue an update message that expects an input message from every workstation in response. Examples of this are negotiations conducted by the participants (e.g., collecting votes or remarks), and internal negotiations automatically invoked by the controller site, e.g., to determine the workstations' capabilities as described in Chapter 5. Such points in a conference we shall call *synchronization points*.

A synchronization point *terminates* when the controller receives the expected response from each workstation. Since not every workstation (or participant) may reply quickly, or some may never reply (e.g., the workstation crashed), it is also important to allow a synchronization point to be terminated by a *timeout*. How the timeout interval is chosen will depend both on estimated transmission delays and on whether the response expected is to be generated automatically by the workstations or manually entered by the participants. Since the delay due to user "think time" is likely to be much longer than transmission delays, a much longer timeout period should be used when responses are elicited from the participants themselves; it may also be reasonable to have a participant such as the "chairperson" decide how long to wait in such a case.

If a synchronization point is terminated by timeout before all replies have been received, the controller may wish to remove the workstations who have not yet replied, or simply continue the conference without taking these workstations' responses into account. Which of these actions is appropriate will depend on what kind of response was expected and how it constrains further processing. For example, the absence of a "vote" from a given participant should rarely require that the participant be removed from the conference; the conference can continue and the given participant's opinion will simply not be considered at this time. (His opinion may or may not be considered later, if and when it arrives.) On the other hand, if the controller is no longer able to transmit messages because one or more workstations are not processing their messages (described under "flow control" in Section 4.6.2), then the conference cannot continue. Unless the controller is willing to wait, and slow down the conference, until these workstations begin accepting messages again, it must remove these workstations and continue the conference without them.

4.4 Replication Strategies

This section addresses the problem of what parts of the shared space the workstations should keep copies of, i.e., should be *replicated*, and the implications of the available choices. We first examine this without consideration of private spaces, and discuss the impact of private spaces at the end of the section.

At one extreme, it is possible to give workstations copies of only the shared window(s) in the conference, not the actual application data (e.g., documents) that is displayed in the windows. This is the "virtual terminal" approach, and has the limitations described in Section 1.6, namely that the display of shared information cannot be tailored to the workstation's capabilities and that application information located at the workstation cannot be transferred. While there are circumstances where a virtual terminal approach is useful and perhaps the only one available, e.g., accessing an existing single-user program without modification from a conference, we will assume below that workstations do have some local intelligence and are capable of storing and processing some application information.

4.4.1 Full Replication

The conceptually simplest approach to replicating application information is for every workstation to hold a complete copy of the entire shared space. This allows the controller and all workstations to run identical software for performing operations on the shared space, and these operations can be transmitted in their highest-level most compact form, thus optimizing performance by minimizing the network bandwidth needed. Full replication allows a workstation to act as conference controller; there is no need for an extra controller site. (In addition, workstations supporting full replication may serve as "backup" controllers and take over the conference from the current controller, as described in Chapter 5.) Full replication also makes it easy for participants to asynchronously "browse" over parts of the shared space within their private spaces; we shall see presently that this is harder to implement when "partial" replication is being used.

If the volume of data in the shared space is considerably larger than what is displayed (in shared windows) at any given time, considerable *delay* will be experienced at the beginning of a conference, and when large objects are added to the shared space, when the controller provides the workstations with an initial copy. This large initial delay can be avoided if the

data is transmitted *asynchronously*, in a way that spreads the delay over a period of time. That is, the controller sends the most urgent data (e.g., document lines to be displayed in shared windows) first, allowing the workstations to display it immediately to the participants. The remaining data in the shared space is sent either when needed for display, or when the conference is temporarily "idle" and the controller can take advantage of spare processor and communication channel capacity; the workstations thus acquire their copies of the shared space a little at a time while the conference is running. Some additional bookkeeping is required of the controller, e.g., flags on each document line or ranges of line-ids, indicating which lines have been sent to the workstations and which have not yet been sent. Extra processing is also needed on each operation to check whether any data needed by the operation has not yet been sent.

Full replication does require that every workstation possess the processing power and storage needed to manage the entire shared space; this may not often be the case. The next section describes *partial* replication in which a copy of selected parts of the shared space is maintained at the workstations. The techniques described below have appeared in recent "distributed editing" protocols [39, 112]. These protocols deal with only one user's workstation and a remote host; we have made necessary modifications and extensions in order to handle multiple workstations in a conference. In addition, single-user distributed editing allows the user's workstation to perform several operations locally without informing the remote host; while this saves communication bandwidth, it is not acceptable for a real-time conference in which all participants' workstations must be quickly informed of each operation so that they may update their displays. (We do consider local execution of operations at the workstation for improved response in Section 4.7, but immediate transmission of operations is still required.) Similar partial replication techniques are likely to be applicable to other kinds of structured objects, e.g. hierarchical "structured display files" used in interactive graphics applications [31]. We will assume in the following that that an identical replication strategy is to be followed for all workstations; "heterogeneous" arrangements, e.g., full replication for some workstations and partial replication for others, are also possible but will require more complex bookkeeping by the controller.

When partial replication is used, the choice of which component objects to replicate at the workstations may be determined by any or all of the following factors:

- Logical sequence, e.g., with lines of a document.

- Level of detail, e.g., with hierarchical documents [28].
- Spatial position, e.g., with graphical data structures [31].

The first of the above is illustrated below using JEDI as an example; similar principles can be applied to other kinds of data structures as well.

4.4.2 Partial Replication

Examining the shared space of JEDI-CONF (in Figure 4-1), we see that it is desirable to replicate most of the information at all participants' workstations. Information about which participants are present, who is the chairperson, who has which window reserved, and so on, is needed in order to display appropriate "status" information. Information about reservations allows a workstation to "filter" out commands that a participant cannot execute on the shared space; when the participant does not hold a window-reservation or the super-reservation, the controller site will not be burdened with commands that it must reject. It is also desirable to replicate the names of documents in the shared Working-Set in order to allow a participant to select a shared document for private browsing; this is discussed in more detail when we examine private spaces. Except for the actual contents of shared documents, the storage needed to replicate the shared space is modest; the variable-size arrays, of participants and document names and windows, will typically have a few elements each.

The bulk of the storage cost of the shared space is therefore due to the contents of the lines in the documents in the shared space, and it is this that we consider for partial replication; all other data in the shared space will be assumed to be fully replicated. If we ignore participants' private windows for now, we see that the only parts of shared documents that actually need to be replicated at participants' workstations are those parts actually displayed in a shared window. It is therefore sufficient to replicate the contents of only those lines that appear in some window. (It is possible that only part of some line may be visible in a given window, but we will not consider partial replication within individual lines because partially-visible lines will be infrequent and the added complexity of breaking up a line will outweigh the possible space savings.)

A minimal form of partial replication would have the workstations keep a copy of just those document lines that are displayed in the shared superwindow. (We do assume that each workstation has at least this much storage available.) After an editing operation, such as scrolling or insertion or deletion, the controller informs all workstations of the "results" of the

operation as follows:

1. Determine which windows need updating. There may be more than one, if a document that was modified is displayed in more than one window; the ensuing steps apply to each window in turn.
2. Determine the new "coordinates", i.e., starting and ending line-id, of the window.
3. For each line in the new line-id range that was not in the old line-id range of the window, send the contents of the line to every workstation.
4. For each line in the new line-id range that was in the old line-id range but has been changed, send an update message. (We shall see presently that this message may carry the entire new contents of the changed line, but will often be more efficiently transmitted if specified as an incremental update to the previous contents of the line.) "Update" messages must also be sent for lines in the new line-id range that have been deleted from the document.
5. Send the new window coordinates if they have changed; if not, a "window ready for redisplay" message, presumably encoded into fewer bits than needed to resend the coordinates, is sent. At this point the workstation will correctly redisplay the window.

Having redisplayed windows as specified by the controller, a workstation will discard any document lines that are no longer visible in any window. This is wasteful, however; if these lines need to be displayed again later in the conference, the controller must send them to the workstations a second time. Performance can therefore be improved if workstations remember lines previously sent, and the controller also keeps track of which lines it has sent so that it need only send those lines which the workstations do not already have. Furthermore, if the conference participants are temporarily "idle", the controller can take advantage of the unused processor and communication channel capacity by sending additional "anticipatory" lines that are not currently visible and have not yet been sent. If and when these lines need to be later displayed they will not have to be resent, again improving response time. Heuristic methods for selecting which lines to send when idle are specified in [39, 112]; for example, the controller may anticipate sequential scrolling and therefore send the next several lines following the lines visible in a window and possibly a few lines preceding the window as well.

The above scheme can be implemented quite easily if the controller associates a single "flag" with every line (of every document in the shared Working-Set) indicating whether or not the line has already been sent to the workstations. The flag on a previously-sent line must be reset whenever the line is modified. (This will almost always be followed by sending the line,

and setting the flag once again, because operations that change the document contents at a given cursor position must make the cursor visible in its window.) This scheme does overlook one problem, however, namely storage space constraints at the workstations. As a conference progresses, the number of lines replicated will grow, approaching full replication of all document lines in the limit. Since our reason for exploring partial replication is that workstations may not be capable of supporting full replication, one or more workstations may soon run out of space.

The solution to the above problem is to place an upper *limit* on the number of lines that may be replicated. The controller can select a reasonable limit by asking each workstation for an estimate of how many document lines it can accommodate in its available local storage, and selecting the smallest of the values returned by the workstations. (This is an example of *protocol negotiation*, described in more detail in Chapter 5. As described in Section 5.1.5, the controller may *timeout* and ignore a workstation that does not reply for a long period of time.) In addition to remembering which lines are currently replicated and which are not, the controller also keeps track of the number of lines currently replicated. When this number reaches the chosen limit and additional lines need to be sent for display, the controller invokes a replacement algorithm (such as "least recently used") to select as many lines as necessary which the workstations are instructed to discard.

It is also possible to have all workstations execute the same replacement algorithm as the controller, in which case they need not be told which lines to discard. This approach is somewhat more fragile, since it requires ensuring that all workstations do have the correct code for executing the identical replacement algorithm, as well as the storage for that code. It also limits the flexibility of the controller in using "non-sequential" protocols, described in Section 4.6.4, with which workstations may not always have the exactly same state as the controller. In addition, very little bandwidth is saved by this method. Explicit commands to discard lines can be encoded into a very small number of bytes (only needing to carry an "opcode" and a line-id), and are only sent when there is some other operation that needs to be sent anyway (else there would be no need to discard a line). These few bytes can almost always be combined into the same network packet as the other operation(s). Having the controller explicitly tell the workstations which lines to discard is therefore simpler than having the workstations duplicate the replacement algorithm, and the extra cost is close to zero.

4.4.3 Private Spaces

We have made several references above to participants' private spaces; we now discuss in detail how these might be implemented. We assume that each participant's private space is managed by his workstation site; the burden of processing all participants' commands on their private spaces will usually be too much for a single controller site. With responsibility for the private spaces on the workstations, we next examine interactions between the private and shared spaces. In JEDI-CONF, there are two sources of such interaction: the participant's paste buffer, for transferring text between windows, shared or private; and private windows into a document that is in the shared Working-Set. We examine each in turn.

A participant's paste buffer is update by COPY and DELETE commands and its contents are used by the PASTE command. So long as these commands are performed on private windows, they can be executed locally by the workstation without any interaction with the controller. If any of these commands are entered on a shared window, however, some interaction between the workstation and controller is required. A straightforward implementation would be as follows:

- A DELETE or COPY command on a shared window is sent to the controller, who must not only report the result of the operation (document and/or window updates) to all workstations but also return the contents of the deleted (or copied) region to the workstation that originated the command; the workstation stores the received text into its paste buffer.
- A PASTE command is translated into an Insert-String operation that is sent to the controller, carrying the contents of the paste buffer as argument. (If the workstation has not yet received the results of a preceding COPY or DELETE on a shared window, it first waits for the paste buffer to be updated by the results of the command.)

Depending on how the participant uses the above commands, some improvement may be possible. If the participant copies or deletes text from a shared window and then pastes it into a shared window, same or different, the affected text must be sent from the controller to the workstation and then from the workstation back to the controller. Sending the text back to the controller, when it originated from the controller in the first place, is wasteful. It is therefore useful to allow the controller to *replicate* parts of participants' private spaces (such as the paste buffer here), in the same way that workstations replicate parts of the shared space. Then, a workstation can send a more compact PASTE command to the controller, with the understanding that the controller will use its copy of the participant's paste buffer to perform

the operation.

Replicating a participant's paste buffer at the controller requires that the controller's copy be kept consistent. If, however, a participant is performing several cut and paste operations in his own private space, then sending the new contents of the paste buffer to the controller each time is wasteful because it is never used in a paste command on the shared space. This can be remedied by the workstation updating the controller's copy *only when needed*, i.e., when the participant does enter a paste command on the shared space. Since the controller never uses the participant's paste buffer on any other occasion, it is not necessary for it to receive every update in "real time". This strategy can be implemented by a workstation associating the following information with its paste buffer:

- A "dirty" flag that, if on, indicates that the paste buffer has been changed locally by the workstation without informing the controller.
- A *Waiting-For* sequence number or other unique identifier specifying a COPY or DELETE command whose result the workstation is awaiting from the controller. Waiting-For will be Undefined if no such operation is outstanding, and the dirty flag is never on unless Waiting-For is Undefined.

The action taken by the workstation is then as follows:

- COPY or DELETE in a private window: Update the paste buffer, set its dirty flag on, and set Waiting-For to Undefined. (The workstation need no longer wait for any results to arrive from the controller because the contents of the paste buffer are completely replaced.)
- COPY or DELETE in a shared window: Send the command to the controller, and set Waiting-For to the sequence number assigned to this command. The controller on executing this command and returning the affected text will also save the text in its copy of this participant's paste buffer.
- PASTE command in a private window: If Waiting-For is Undefined, insert the contents of the paste buffer into the given private document. If not, *wait* until Waiting-For becomes Undefined, i.e., until the awaited results are received from the controller, and then use the contents of the paste buffer.
- PASTE command in a shared window: If the "dirty" flag is off, send the command to the controller who must have an up-to-date copy of the paste buffer. If not, first send the contents of the paste buffer to the controller and set the dirty flag off, and then send the paste command.
- Results of COPY or DELETE received from controller, carrying a sequence number: If the sequence number matches Waiting-For, update the paste buffer and set Waiting-For to Undefined; the dirty flag must already be off, and need not be updated. If the sequence number does not match Waiting-For, discard the message because it carries the results of a command rendered obsolete by a

subsequent COPY or DELETE in a shared or private window.

We next examine the necessity of the controller transmitting the results of a COPY or DELETE command to the workstation. Since the workstation has some lines of shared documents replicated, it is possible that it may already have the text to be copied or deleted. The controller therefore does the following on executing a COPY or DELETE:

- If every line between the mark and cursor inclusive is currently replicated by the workstations, simply send the COPY or DELETE command as its own "result" to all workstations. Every workstation can delete the selected region (or do nothing on a COPY), while the workstation that originated the command will in addition save the selected text in its paste buffer. (The workstation in question will recognize its own COPY or DELETE command by matching the Waiting-For sequence number as described above.)
- If one or more lines in the selected region are not currently replicated, send these lines to the originating workstation (only) and then send the COPY or DELETE command to all workstations. The other workstations will delete all affected lines that they do have a copy of (or do nothing on a COPY command, again), ignoring the other deleted lines that they do not have. The originating workstation will save the selected text, which includes text from the lines specially sent by the controller, in its paste buffer. The extra lines are sent by the controller only to the originating workstation, not the others, and are specially marked. The workstation uses them only for copying into the paste buffer; it does not retain them in its partial copy of lines from the shared documents, so that its copy stays consistent with the other workstations and the controller.

The other interesting issue in implementing the private space has to do with the possible overlap of shared and private Working-Sets. We assume that every document is permanently stored at some *file server* which also enforces access controls, grants and takes away reservations, and so on, and that the file server for a given document is easily determined (e.g., embedded as part of the document "name"). In general, the workstation of a participant wishing to view or edit a document in his private space must obtain the document from its file server, which will verify the participant's access rights. (A workstation with limited local storage may negotiate "partial replication" of documents with their file servers, as in the distributed editing protocols of [39, 112].) If, however, the document in question happens to be in the shared Working-Set and is being edited in the conference, the workstation should instead get the contents of the document (or at least those lines that are to be displayed in a private window) from the conference controller. The reason for this is that obtaining the document from the file server will only get the last "checkpointed" version; the intermediate editing changes being made in the conference will not be seen. (If the participant specifically

does not want to see the editing changes as they are made, he should be allowed to ask for a "frozen" version in his command; his workstation will go to the file server in this case.)

Therefore, when a participant performs a Edit-Document command in a private window, his workstation checks its copy of the document names in the shared Working-Set (which is replicated) and acts as follows:

- If the given document is not in the shared Working-Set, obtain the document from its file server.
- If the given document is in the shared Working-Set, obtain the contents of the desired private window from the conference controller.

Before we discuss in detail how the contents of a private window are obtained from the controller, by "registering" the window as described below, we briefly examine how the above strategy is affected by changes to the shared Working-Set. A document that was obtained from its file server, because it was not in the shared Working-Set at the time, may later be added to the shared Working-Set. If so, the participant's workstation, on noticing the change to the shared Working-Set, should switch strategies for that document and obtain the contents of private windows from the controller. Conversely, having registered a private window with the controller, the given document may later (or concurrently) be removed from the shared Working-Set. If and when this happens, the controller will send the workstation a message stating that it can no longer supply the contents of the private window(s), and the workstation now obtains the document from the file server instead.

If the contents of shared documents were fully replicated at all workstations, implementing private windows would be trivial: Since it already has a complete copy of the shared document and receives updates from the controller as they happen, a workstation can display a private window without any further help from the controller. With partial replication, however, a workstation might find that it does not have copies of some or all of the lines needed for display in the private window. It therefore "registers" the private window with the controller, sending it a message specifying what range of line-ids are covered by the private window. Thereafter, the workstation informs the controller whenever the range of line-ids changes (as a result of scrolling the private window), and when the private window is destroyed (by switching it to another document, or by an adjacent private window growing to cover this one). The controller keeps track of every workstation's private windows into all shared documents, and ensures that each workstation has an up-to-date copy of all

document lines in that workstation's private windows. If the number of lines replicated at all workstations is well below the specified limit, the controller may simplify its bookkeeping by sending lines needed by any workstation's private windows to all workstations. This will no longer be possible once the number of lines replicated approaches its limit, however. Instead, whenever a workstation registers a new private window or scrolls an existing one, the controller sends the workstation those document lines needed for the private window that are not already in the set of lines replicated by all workstations or already in this workstation's private windows.

4.5 Adding and Removing Participants

Suppose a conference is already in progress, with a group of participants' workstations replicating some set of objects using some set of input and update messages, and a new participant joins. In order to correctly process subsequent update operations sent by the controller, the new participant's workstation must first be supplied with starting values of all objects replicated in the group. Furthermore, the method used for supplying these initial values must be properly coordinated with new update messages that may be sent, so that the new workstation is added to the group at just the right time, when it holds the same version of the replicated objects that the other workstations hold.

4.5.1 Immediate Initialization

The simplest approach to adding a new workstation to a conference is to immediately send it the current value (in the form of "absolute" update messages) of all objects replicated in the existing group of workstations. This might be refined slightly by "decomposing" the contents of large objects into smaller pieces and sending a sequence of operations of which at least the first one must be an absolute update. For example, the contents of a bitmap could be sent as an initial Clear-Bitmap operation followed by a sequence of Write-Scanline operations carrying the contents of each horizontal scan-line one at a time. This will allow the receiving workstation to write the scan-lines of the bitmap into the allocated storage, and display them to the participant, as they are received. This would not be possible if the bitmap contents were sent as a single monolithic "message" that cannot be processed until it is first completely received in a separately-allocated buffer. Not only is the extra storage saved by sending and processing one scan line at a time, but the effect on the participant, who is

waiting for his screen to be updated, will be more pleasing. Various other methods of decomposing large objects are possible, e.g., since a Clear-Bitmap operation sets all bits to zero, it can be followed by a sequence of operations that write contiguous "runs" of one bits. In the case where not all the replicated data is actually being displayed in a shared window (e.g., document lines), the controller can send the displayed data first so that the workstation can show it to the participant immediately while the rest of the data is being sent.

While a copy of a replicated object is being sent, in whole or in parts, to a new workstation, further updates to the object's current value must be prevented until the complete object contents have been transmitted, at which point the new workstation can be added to the group and updates reenabled. This may involve a significant delay for large objects, especially if the rate at which the new workstation can process incoming data, or the network bandwidth, is limited. Whether or not this is a serious problem will depend on the length of the delay and on whether the participants perceive it to be a problem. In many cases, the entry of a new participant into a conference represents a significant enough point of discontinuity that the participants can tolerate the delay; they might occupy themselves with a conversation on the voice channel, e.g., to briefly summarize the conference and introduce the new participant. In other cases, participants might not wish to have their commands held up. The controller could try to choose an appropriate time for giving the new participant's workstation its copy of replicated objects in order to minimize this effect, e.g., after informing all participants that a new participant is joining, the controller could wait for a few seconds in the hope that participants on learning of this will quickly complete their commands and stop typing, and only then send the replicated objects to the new workstation. Any such delaying action is only probabilistic (participants might not stop entering commands), and will in turn increase the delay seen by the new participant; this again may not be a problem if a joining participant expects a brief delay anyway.

4.5.2 Asynchronous Initialization

If not all of the replicated objects' contents are immediately needed for display in a shared window, the controller can try to "spread out" the above delay by first sending just those objects (or components) that are urgently needed, and sending the rest asynchronously. That is, the controller adds the new workstation to the existing group but also remembers which of the replicated objects it has not yet sent. The controller then does the following:

- Whenever it has the opportunity (e.g., the conference is momentarily idle), the controller sends a few more of the replicated objects to the workstation and marks them as having been sent.
- Whenever an update or display operation is to be performed that requires one or more objects that have not yet been sent to the given workstation but have been sent to the others in the group, first send the required objects to the given workstation before performing the operation.

Since replicated objects are sent to the new workstation only as needed, or when there is spare processor and communication capacity, an update to the replicated objects is delayed for only as long as it takes to send the new workstation those objects needed by that update. This will usually be imperceptible, except perhaps for drastic changes for which participants may expect some delay anyway.

The above technique is essentially identical to asynchronously supplying all workstations with replicated objects, as described in Section 4.4.1. The difference here is that the controller must keep track of which workstations if any are in a "transient" state with respect to the others, and for each of these must separately remember which objects have been sent and which have not. This will in the worst case require one extra bit per replicated object per workstation, but may often be "compressed", e.g., by remembering ranges of line-ids or array indices per workstation.

4.5.3 History-Based Initialization

An alternative to the above, which completely eliminates the blocking of further updates at the cost of taking the new workstation somewhat longer to catch up and join the group, is to use the *history* of updates to the replicated objects, assuming such a history is being recorded. The controller selects the most recent "absolute" update or "checkpoint" in the stored history, and sends that to the new workstation followed by all subsequent operations (which must be "relative" updates) in the history. Once all of these operations have been sent, the new workstation is added to the group and will receive subsequent updates. If the other workstations are in the meantime sending input messages that cause further updates, these are not a problem because the updates are simply appended to the history so that the new workstation will receive them before it is added to the group. (The controller must synchronize its actions to ensure that the new workstation is not considered to have exhausted the history, and added to the group, while a new update is being appended.)

The main problem with using a history of updates is that it may take a long time to bring a workstation up-to-date, especially when the most recent checkpoint (e.g., a saved version, or an absolute operation that clears or resets the contents of an object) is followed by a long sequence of relative updates. This is not a problem for a participant who specifically wishes to review the history of the conference when joining. If instead the participant does not wish to review the history, the transmission delay, and processing load on the controller, in sending the history will usually be worse than if the current object contents were sent.

A combination of methods may be used as a compromise. That is, a checkpoint is sent if it is recent enough that the subsequent updates can be quickly processed by the new workstation; if not, the current values of replicated objects are sent, immediately or asynchronously as described above. If the controller does have to send the current value immediately, and temporarily block out further updates, it may take the opportunity to checkpoint the state of the replicated objects at the same time, in the same traversal over the given objects. This will be useful if the designer wishes to support queries on the conference history such as "show the state when participant X joined"; or, a participant in the conference may have explicitly requested a checkpoint on learning that a new participant is about to join.

Recording an object's update history can be particularly useful when adding a new workstation if the workstation already has some recent copy of the object. This may happen if a participant leaves a conference briefly and then joins again, during which time his workstation has not discarded its copy of objects from the conference. Or, when documents are read and written from file servers, the workstation may have obtained a previous version of a document from its file server or may be instructed to do so by the controller. If the workstation reports to the controller which previous version, if any, it has, then the controller need send only the intervening updates if they are still available in the history; it might even be the case that the workstation's version is still current and no updates need be sent. Since the controller does not have to send a starting value, even a moderately long sequence of updates may be better, in terms of transmission and processing delay, than sending the entire current value.

4.5.4 Removing a Participant

A participant's workstation may be removed from a conference when the participant decides to leave the conference, or when an authorized participant forcibly revokes his permission to stay, e.g., because of misbehavior. (Which participant can do the latter, e.g., a chairperson, is up to the designer to decide.)

In addition, the controller site may remove a workstation if it is not responding for a long period of time, or if the workstation is not processing received messages quickly enough and is slowing down the conference. In either of these cases, the workstation might in fact have crashed. The controller may be programmed to remove a slow workstation automatically on a timeout, or may let an authorized participant such as the chairperson decide how long to wait. If it is the chairperson's workstation that is slow, the controller should designate a new chairperson to make such decisions.

A participant who left the conference may be allowed to join again, depending on the circumstances under which he left. If the participant was forcibly removed, the controller will refuse his attempts to join again unless he has been "forgiven" and explicitly given permission to join again. A participant who left of his own accord should of course be allowed to join again, unless his permission to join was revoked in the meantime. In the case of a participant who was removed because of communication problems, the controller must make a judgment as to whether the problem was transient, and probably will not occur again, or persistent. Unless the participant is rejoining from a different workstation, or more specific information is available from the network (e.g., the problem was a network "partition" that has been repaired), the controller can at best make a guess that may turn out to be wrong. A reasonable approach in this case is to give the participant's workstation a second chance (or some number of chances) and to remove the workstation permanently if communication problems persist.

When a participant leaves a conference, his workstation may retain its copy of replicated objects from the shared space. If the participant then joins again, the workstation can report the version number(s) on its copy to the controller; this may allow the controller to send the workstation a short history of changes instead of a complete new copy as described above. The workstation may instead discard its copy when the participant leaves, if:

- The participant indicates that he does not plan to return to the conference. (He may of course change his mind later.)

- Local storage is scarce; having left the conference, the participant may need storage space for whatever he does next, e.g., reading mail or locally editing documents.
- The participant has been away from the conference for a long time; the old copy is likely to be useless because the intervening history of updates may be very long.

If the participant does rejoin a conference after his workstation has discarded its copy of replicated objects, the controller will have to supply it with a new copy from scratch.

If a participant leaving a conference does plan to return, his workstation may even wish to continue receiving update messages from the controller, updating it copy of the shared space without displaying it to the participant. The workstation should only do this if it has sufficient spare storage and processing power, over and above what it needs to handle whatever else the participant wishes to do on leaving the conference. If the designer wishes to allow for this possibility, the controller site must distinguish between two different "statuses" of a participant: whether updates are being sent to the participant's workstation, and if so whether the participant himself is actually present. (Note that the status information displayed to the other participants should be based on the latter.) The controller may also refuse to continue sending update messages to the workstation of a participant who left, simply to unburden itself of the extra load or because the workstation is slow in processing these updates. Because the other participants are not concerned with whether or not such a workstation is still receiving update messages, the controller can remove the workstation automatically without requiring intervention from the chairperson or any other participant.

4.6 Transport Protocols

We have implicitly assumed so far that reliable sequenced transmission of messages between two sites is automatically handled by the communication network. Even with this assumption, there are some known techniques that can be used for minimizing communication overhead, which we briefly review. We also describe some further improvements in performance that may be possible if the controller and workstations handle some of the lower-level communication details themselves.

4.6.1 Message Encodings

We assume that some efficient procedure is used for *encoding* an abstract input or update message into a linear sequence of bytes suitable for transmission, and for *decoding* such a linear sequence at the receiving end. As described by Sproull and Cohen [110], a message is typically encoded as an "opcode" that designates the particular operation being reported or requested, followed by the encodings of zero or more arguments. A general-purpose encoding procedure, such as Xerox's Courier [117] or Herlihy and Liskov's procedure for abstract data types [54], may be used. Or, the designer may hand-tailor a more compact encoding, at the cost of greater software complexity, based on the expected usage. For example, in "virtual terminal" protocols the transmission of a single printable character usually means "display this character at the current position and advance the position". The "opcode" is implicit in the argument and is not transmitted, except in the case of other operations which are assumed to be less frequent. Or, a "Clear-Bitmap" operation that takes a single one or zero bit as argument can be encoded as two different opcodes each of which requires no argument to be transmitted.

The message encoding will also depend on what kind of lower-level transport protocol is being used. For unreliable "datagram" communication (described in Section 4.6.4), input and update messages must include version numbers so that a receiver can detect out-of-order and duplicated messages. Version numbers are not required in the more common case of reliable sequenced communication, although they may still be needed in certain messages as described in Section 4.3.3.

A given linear encoding of a message will usually need to be transmitted as one or more *packets* of whatever maximum packet size the given transport protocol supports. (There is no need to construct the linear encoding separately before sending packets; the encoding should be performed directly into packet buffers if possible to save storage and processing time for copying.) If a given message is larger than the maximum packet size, it must be *fragmented* by the sending site into multiple packets, which are *reassembled* by the receiver; a special "end-of-message" bit with each packet, such as provided by the Xerox Sequenced Packet Protocol (*SPP* [118]) and X.25 [13], can be used to do this. If instead a *bytestream* protocol such as the U.S. Department of Defense Transmission Control Protocol (*TCP* [14, 97]) is being used, message boundaries can be delineated either by embedding message

length descriptors or by using a special byte value as terminator.[14] If many of the messages to be transmitted are smaller than the maximum packet size, embedding message lengths or terminator bytes will allow more than one message to be sent in a single packet. Such message *aggregation* will often improve performance, but in an interactive application such as conferencing must be used with care. That is, a partially filled packet should not be buffered for too long and should probably be sent if no further messages are generated within a short time interval (some fraction of a second), or if the application explicitly requests that buffered data be "pushed" out.

For the rest of this section, we shall assume that the communication network allows arbitrary-length messages to be transmitted, with the understanding that message fragmentation and reassembly, or aggregation, is performed when necessary.

4.6.2 Virtual Circuit Efficiency

The most common form of transport protocol, available on every network in some form or another, is a *virtual circuit*; examples of virtual circuit protocols are TCP [97], Xerox's SPP [118], and the ISO Transport Layer Protocols [60].

A virtual circuit is a *bidirectional* (or "full-duplex") channel of communication for transmitting messages, reliably and in sequence, between two sites. A virtual circuit is implemented by a transport protocol "layer" [59] that hides message loss, duplication, and reordering.[15] This is done by associating a consecutively increasing *sequence number* with each message in a given direction of the virtual circuit, and *retransmitting* each message periodically (based on some estimate of the transmission delay) until an *acknowledgment* is received. Virtual circuit protocols also provide some form of *flow control* [35] to prevent the sender in a given direction from sending messages faster than the receiver can process them. Flow control is usually implemented by some form of *window* strategy: the receiver specifies a "window" of how many additional messages it is willing to accept, and advances the window as it processes received messages. The sender is not allowed to send any messages beyond

---

[14] In the latter case, occurrences of the special byte value within the body of a message must be "quoted", usually by repeating the given byte value, to distinguish them from message terminator bytes.

[15] Corruption of transmitted messages is also usually masked by attaching a "checksum" and discarding a message that is received with a bad checksum; the message thus appears to have been "lost".

the window specified by he receiver; it must wait for the window to be advanced. A sender that is unable to send additional messages because of a "closed" window may timeout and abort the virtual circuit if it has to wait too long; the same applies if a given message remains unacknowledged after a large number of retransmissions.

Acknowledgments of messages in a given direction are combined with data messages in the reverse direction whenever possible. To minimize the network load, and the number of message processing events at the other host, it is important that the number of messages carrying acknowledgments only and no data be kept as small as possible. The standard method of doing this is for a receiver to send acknowledgment-only messages only when the sender explicitly *asks* for them. (A special bit in the message header, as in SPP [118], is reserved to indicate whether or not an acknowledgment is being requested.) The sender in turn asks for acknowledgments as infrequently as it can, only when the receiver's flow control "window" is about to be closed or when retransmitting an unacknowledged message. (The sender may also ask for an acknowledgment if its queue of unacknowledged messages is too long, whether or not the receiver's window has closed.)

For interactive communication, e.g., remote login or a real-time conference, the above mechanism is not entirely sufficient; a message that is lost will not be retransmitted until much later. The following extensions are therefore useful:

1. If a message is received out of sequence, i.e., with a higher sequence number than the next message expected, there is a very high probability that the previous message was lost. The receiver therefore returns an unsolicited acknowledgment, in effect a "negative" acknowledgment, urging the sender to retransmit the missing message as quickly as possible without waiting for the message's scheduled retransmission time. (The sender on receiving an unsolicited acknowledgment will not always retransmit the next unacknowledged message immediately. The incoming acknowledgment may in fact be quite old, having been delayed in the network; a message that was sent or retransmitted fairly recently is therefore probably on its way to the receiver and is not retransmitted.)
2. When a message is sent that expects a message in response, the sender requests an acknowledgment. This ensures that the message will be retransmitted later if lost, even if no further messages are sent and the receiver's window does not close. (The receiver of a message requesting an acknowledgment should delay returning the acknowledgment for a brief interval while the message is being processed; if a message is quickly generated in response, the acknowledgment can be included in the same message [22].)

Except for case 2 above, the actions described are usually performed automatically by the

virtual circuit protocol layer, beyond the application programmer's control. The last technique, however, requires the application to specify when a given message expects a response. The virtual circuit layer will either accept an extra argument in the procedure call for sending a message [22], or allow the application to directly set the bit in the message header that requests an acknowledgment [118]. When this is possible (not all virtual circuit protocols allow this, unfortunately), the designer should exploit it: a workstation should request an acknowledgment when sending an input message that expects an update message in reply, and the controller should request an acknowledgment when it expects a response from a workstation, e.g., at a "synchronization point".

4.6.3 Reliable Multicast

The virtual circuit between the controller and each workstation is implemented by lower-level protocol software independently and without knowledge of how the other virtual circuits are being used. When the same information must be sent to all workstations, this results in the following inefficiencies:

- Each virtual circuit has its own retransmission queue; storage is wasted when the same message is duplicated in multiple queues.
- Each virtual circuit performs retransmissions independently. Multiple "timers" must be set and responded to for retransmitting the same data to different destinations, increasing the "process switching" overhead.
- If efficient *multicast* communication is supported by the network for transmitting a message to multiple destinations (e.g., on a local area or satellite network), it cannot be exploited by independently-implemented virtual circuits; the same data must be sent separately to each destination.

These problems may not be serious for small conferences (say two or three workstations), but for larger conferences will impose considerable overhead on the controller. An alternative approach is to use *reliable multicast*, which uses the same acknowledgment and retransmission techniques as virtual circuits but with a single retransmission queue and timer for all destinations.

Reliable multicast of a sequence of messages from the controller to a group of workstations works as follows. We assume that messages are transmitted in the form of unreliable *datagrams*: a datagram is not guaranteed to arrive at its destination, but the

probability of its not doing so is extremely low, usually less than one percent.[16]

1. Each new message is assigned the next higher sequence number and is sent to all workstations, by one of the methods described below.
2. For each message sent, the controller remembers which workstations have acknowledged receipt and which have not; a bit-string of the required length may be used.
3. A message is retransmitted periodically until acknowledged by all workstations, or until the workstations who have not acknowledged the message are removed by the controller, at which point the message is discarded from the retransmission queue.

4.6.3.1 Multicast Datagram Transmission

When the controller and all workstations are on a single network that supports multicast communication (e.g., Ethernet [107] or SATNET [62]), a message can be sent to all workstations using a single datagram whose destination address is a *multicast address* selected by the controller or the network for the conference. This will considerably reduce the overhead to the controller of sending an identical message to all workstations. In some cases, an underlying point-to-point network or internetwork may support multicast communication using a *forwarding tree*: the site sending a message (in this case the controller) sends it to each of its neighbors in the tree, and each site in turn forwards the message to every neighbor except the one it received the message from. Tree-structured multicast has been proposed many times in the literature [24, 116], but implementations are few. One exception to this is the *ST* protocol [32] currently used in the Darpa internet to support voice conferencing.

If not all workstations are on a single network that supports multicast, the controller can still reduce the cost of message transmission if subgroups of workstations are on multicast networks. For any remaining workstations that are "alone" on their respective networks or are on networks that do not support multicast, separate datagrams must be sent. In the worst case, the controller can "simulate" a multicast by sending a separate datagram to each workstation. Even in this worst case, the cost savings of a single retransmission queue and timer, as opposed to separate virtual circuits, may be significant with more than two or three

[16] It is assumed that damaged datagrams are not delivered, nor are datagrams delivered to the wrong destination; checksumming techniques are used to ensure this.

workstations.

4.6.3.2 Acknowledgment and Retransmission

Reliable multicast as described above has appeared before in the literature [106, 86].[17] Published reliable multicast protocols do not, however, address the problem of acknowledgment and retransmission in much detail; we try to remedy this by describing some non-obvious extensions of the acknowledgment and retransmission methods used in virtual circuits.

Since datagrams are assumed to be rarely lost, a message needing retransmission will usually remain unacknowledged by one workstation only; a datagram addressed to this workstation only, rather than a multicast datagram, should be used for the retransmission. If a given message has not been acknowledged by *any* workstations when its retransmission time arrives (e.g., the message was damaged in the network before any workstation could receive it), retransmission should of course be done using multicast. In the rare case that two or more, but not all, workstations need retransmission of a given message, the controller has a choice: retransmission using multicast will incur unnecessary processing overhead (to check the sequence number and discard the message) at those workstations that already have the message, whereas separate retransmissions to only those workstations that need it will incur more overhead at the controller. If we assume that the controller is more likely to be a bottleneck than any individual workstation, multicast should probably be used for retransmitting to two or more workstations. A different "threshold" might be chosen if different assumptions about controller and workstation performance hold, but we do note that any difference in performance is not likely to be very important because this situation will be rare.

When different workstations need *different* messages retransmitted, there is an additional optimization available to the controller: If individual messages are small and two or more will fit into a single network packet, the controller can "batch" several retransmissions, possibly intended for different workstations, into a single multicast packet. Each workstation will then pick out the messages that it is missing, if any, and discard the rest.

---

[17] Most such references use the term reliable "broadcast"; we prefer to use "multicast" to denote a *known* set of destinations, as opposed to "broadcast" which usually means an *unknown* collection of "all" the sites on a given network. This usage is consistent with Boggs [10].

4.6.3.3 Acknowledgment Bursts

The total volume of incoming acknowledgments can be minimized using the same techniques described for virtual circuits, namely asking for acknowledgments as infrequently as possible, and not returning an unsolicited acknowledgment unless there is data to send in the reverse direction or a message is received out of sequence. In addition, it is important that the controller not be sent too many messages all at once; the controller's network interface may not be able to pick up all of them quickly enough, which will degrade performance and response time because of the need for later retransmission. Preventive measures should be taken to avoid this whenever a *synchronization point* occurs, where the controller sends a single message that expects a response from every workstation.

If the number of workstations is small (say two or three), or if there is a significant variation among the transmission delays to different workstations, the above problem can be ignored because the probability of too many responses arriving in a short burst is very low. The same is true if virtual circuits or separately-addressed datagrams are used, because the workstations will receive their copies of the message at different times.[18] However, when a large number of workstations on the same local network are sent a single multicast datagram, the probability of their replies arriving in a short burst is quite high. The workstations should therefore introduce short artificial *delays* before sending back their replies, in order to reduce the probability of collision. This could be done in either of two ways:

- Each workstation waits a *random* amount of time before returning its reply. (This is an application of the randomized retry technique first popularized by Ethernet [82].)
- The replies are *staggered* by assigning each workstation a different delay interval, e.g., proportional to the workstation's index in an array.

4.6.3.4 Remarks on Multicast

In a conference, reliable multicast as described above will provide reliable sequenced transmission of messages only between the controller and all workstations; it does not address the transmission of "input" messages from a workstation to the controller, or from

[18] There will be a problem if the response from the first workstation to be sent the message arrives before the controller has finished sending the message to the other workstations; many network interfaces are unable to receive while transmitting. This situation should be very unlikely because the round trip transmission time plus the processing time needed by the workstation to generate its response will almost always be greater than the time taken by the controller to send out all copies of the message.

the controller to an individual workstation (e.g., error messages or supplying a new workstation with its initial copy). We assume that separate connections are used for such communication between the controller and each individual workstation, in the form of virtual circuits or a simulation of virtual circuits using datagrams and retransmission. These individual connections will also be used by the workstations to return acknowledgments of multicast messages from the controller, so that separate acknowledgment messages are not needed when a workstation has data (an input message) to send.

The existence of independently-sequenced connections (one for multicast, one for workstation-specific messages) between the controller and a given workstation raises the possibility that the workstation may receive messages on the two connections in a different order than they were sent by the controller. (A message sent on one connection may have been lost and is awaiting retransmission while a newer message sent on the other connection is received.) In cases where the controller cannot tolerate this reordering, i.e., a given new message on one connection must be processed after a given message on the other connection, the controller should first wait for an acknowledgment of the other message before sending the new one. The same principle applies if reliable multicast connections are used for different "groups" of workstations, as described in Chapter 5, and a given workstation is in more than one such group. An alternative implementation of multiple groups is to have a single reliable multicast connection for the sites in the union of all the groups. Every message sent on this connection must identify which group(s) (or, equivalently, which sites) it is intended for, and workstations process or discard received messages accordingly.

In summary, reliable multicast offers significant savings in communication overhead, especially when the number of workstations in a conference is large and when the network directly supports multicast datagram communication. These savings are offset by the increased complexity of programming the controller and workstations to perform acknowledgment and retransmission[19] and to enforce orderings between multicast and workstation-specific messages when needed. The designer should take this into account if considering multicast communication. (The designer may also have the controller site dynamically decide whether or not to use multicast, e.g., switching from virtual circuits to multicast when the conference grows larger than two or three participants; the techniques of

---

[19] Unlike virtual circuits, which take care of acknowledgment and retransmission internally, reliable multicast is unfortunately not generally available as a transport-level protocol service.

Chapter 5 can be used to support such dynamic selection of the protocol.) Some networks support only virtual circuit communication, in which case of course the designer has no other choice.

4.6.4 Non-Sequential Datagram Protocols

Non-sequential protocols are an extension of *real-time* communication protocols such as used for packet voice transmission [20]. Real-time protocols differ from the more common virtual circuit protocols in that the emphasis is on delivering packets within some maximum delay interval rather than guaranteeing reliable sequenced delivery. Interactive voice data, for example, has the characteristic that it loses its relevance after a very short period of time (a fraction of a second), and that occasional loss of data can be tolerated because an intelligible waveform that approximates the original one can still be reconstructed at the receiving end. Therefore, the acknowledgment and retransmission mechanisms used for reliable sequencing are both unnecessary and introduce unacceptable delays; if a packet is lost, it is more important to send the next voice packet than it is to retransmit the earlier one. Real-time protocols, such as for voice, are therefore based on unreliable "datagram" communication; packets are not acknowledged or retransmitted. Voice packets are labeled with timing information to allow the receiver to reconstruct the voice waveform properly. If a given voice packet does not arrive by the time the receiving voice decoder needs it, the decoder fills in some standard value such as zero or performs some extrapolation based on previous values; these details are peculiar to voice and are not relevant here.

In a real-time conference, a similar protocol can be used effectively for transmitting small objects such as the position of a participant's "mouse". For such objects, past values are of little use and it is more important to display the latest position on all participant's screens as quickly as possible. (Note that the timing requirements are less stringent than voice. A position report can be accepted and processed at any time so long as it is more recent than the last one processed; hence the term "non-sequential" rather than "real-time".) Virtual circuit communication is neither necessary nor useful in this case; it is more efficient to transmit mouse position reports as unreliable datagrams, at some frequency that provides reasonable response without overloading the network. Each mouse position report is an "absolute" update that carries the new position, not an increment or offset from the previous position, and a version number or timestamp to allow a receiving workstation (or the

controller) to discard obsolete (or duplicate) reports that are older than the version that the receiver already has.

4.6.4.1 Decomposing Large Objects

For larger objects, e.g., a document, transmission of a complete new copy is expensive, especially when each change is small relative to the size of the object, e.g., the insertion or deletion of a few characters. It is still possible to use a non-sequential protocol if a large object is broken into smaller components with a suitable naming scheme, such that:

- The components are of fairly limited size.
- The number of components is fairly limited, so that the overhead of maintaining a version number for each one is not prohibitive.
- Operations on the object do not cause the names of the components to change, or do so only rarely.

This is the approach used by David Reed in some experimental protocols developed at M.I.T. These protocols were mainly designed for two-party communication; we have extended Reed's work below by adding the mechanisms needed for handling multiple workstations holding copies of an object, and by proposing additional acknowledgment and retransmission schemes.

Limiting both the size and the number of components of a large object will usually involve a tradeoff. A bitmap, for example, can be treated in the extreme as one monolithic object, or as several small one-bit objects. The former choice is not practical for an non-sequential protocol because the entire bitmap will have to be sent every time any part of it is updated. The latter choice, on the other hand, requires remembering a version number for every bit, which will take up an amount of storage several times that of the bitmap itself. Reed's solution, which we also use in the prototype system of Chapter 6, is to treat the bitmap as a rectangular array of blocks, each of which is a rectangle of M by N bits, for some fixed M and N. Larger values of M and N will require fewer version numbers to be remembered, but will also require sending larger update messages whenever any part of a block changes. These considerations must be balanced against each other when selecting a block size.

Array- and record-like objects, as well as name lookup tables, are well-suited for non-sequential protocol implementations because the component names do not change. On the other hand, sequences that allow insertions and deletions at arbitrary positions (e.g., a

document whose lines are numbered consecutively) are not well suited for non-sequential protocols. This is because an insertion or deletion of a line at position N, say, causes the "names" of lines N + 1 and above to shift by one. Using a non-sequential protocol, this would require sending an update message for every line number from N through the end of the document. This problem can be remedied if the data structure is redesigned, as was show in Figure 4-2, so that line numbers are no longer a consecutive sequence. Then, the results of an operation on a document can be transmitted as "absolute" updates as follows:

- An insertion or deletion within a single line causes that line's version number to be incremented, and an update message is sent to all workstations carrying the line's id, new version number, and entire new contents.
- Deletion of the EOL between two lines results in two update messages. The preceding line is updated (and version number incremented) to contain the concatenation of the original two lines. The value associated with the id of the second line is updated to the special value Undefined. Ids of deleted lines, and their incremented version numbers, are retained in the document data structure so that the "deletion" (new value of Undefined) can be retransmitted to workstations that missed it, and so that if the same line-id is later reused (to accommodate an insertion) it uses higher version numbers that cannot be confused with the older version numbers associated with the same line-id. Inclusion of deleted lines in the data structure will require that the code for accessing the next or previous line, or to scan N successive lines for display in a window, be modified to skip over lines with associated value Undefined.
- Insertion of an EOL within a line will usually result in two updates. The existing line is updated to contain the characters preceding the insertion, and the remaining characters of the line are the assigned to a newly-created line. If the new line is assigned an id that was never used before, any non-zero version number, say one, can be assigned. If the id of a previously-deleted line is reused, the existing version number associated with that line-id is incremented.

Insertions or deletions of text strings containing one or more EOLs will result in multiple line updates, by a simple extension of the above. In the rare case where insertion of one or more EOLs causes "renumbering" of lines, because the gap in line-ids is not large enough for assigning new line-ids, then "updates" and new version numbers must be sent for the old and new ids of all renumbered lines.

4.6.4.2 Acknowledgment and Retransmission

Update messages, transmitted as datagrams, may not be received by one or more workstations; if the same object or component (e.g., a document line) is not quickly updated again, a workstation's copy will remain behind the controller's. Some "acknowledgment" of

which object versions a workstation has received, and retransmission of lost updates, is therefore needed. Reed's solution for a bitmap is for the controller to periodically request a workstation to return an "acknowledgment" carrying the version number on each block of the workstation's copy of the bitmap; the controller then retransmits those blocks for which the workstation's copy is behind. Note that, unlike a virtual circuit, only the latest value of an object need be retransmitted, not all previous ones. If it is important for the controller to ensure that all workstations receive and process the current value of an object (e.g., when an important error message or prompt is being displayed), the controller can inhibit further updates to the object until all workstations have acknowledged that they have the current version. (As with any other "synchronization point", the controller may have to wait a long time, and may choose to remove or ignore workstations that do not acknowledge.)

A non-sequential protocol allows the controller to tailor its acknowledgment and retransmission strategy to the needs of the application. Instead of asking for all objects or components to be acknowledged at the same time (as with the bitmap above), the controller may selectively ask for different acknowledgments at different times, with different frequencies. For example, in JEDI it is important that workstations be quickly provided with those document lines that are displayed in shared windows. The controller, after every change to a shared window (scrolling, or changes to one or more displayed lines) sends the new contents and version numbers of changed lines followed by a *window update* that carries a window version number and an array of <line-id,version-number> pairs for the document lines in the window. A workstation receiving a window update returns its array of version numbers for the designated lines (some of these version numbers may be "zero", if the workstation does not have any copy of a given line), and the controller will retransmit any displayed lines for which the workstation does not have the latest version. Unless the window is further updated, the window update itself is retransmitted by the controller until every workstation's acknowledgment indicates that it has all displayed lines up-to-date.

With a replication strategy that allows workstations to hold copies of lines not currently displayed, the controller may similarly request acknowledgments and perform retransmission of these other lines. However, the controller has more flexibility available to it. It may for example send a few extra document lines while "idle", but request acknowledgments of these lines only when the next window update is sent; a workstation can therefore return acknowledgments for the lines in the window and for the extra lines in a single message.

Furthermore, if an acknowledgment from a workstation indicates that it did not receive one of the extra lines that was sent, the controller does not have to retransmit the line immediately. It may again wait until it is idle, or defer retransmission of the line until it is actually needed for display, if that ever happens. Commands to *discard* previously-sent lines need never be acknowledged or retransmitted; at worst, a workstation will fail to discard a line that it was supposed to.

The important point above is that use of a non-sequential protocol allows the controller to tailor the reliability of transmission to the relative urgency of the information being transmitted; additional document lines are sent, acknowledged, and retransmitted in a way that never interferes with ensuring that the workstations quickly receive window updates and displayed lines. This is not the case with reliable sequenced communication; if the controller decides to take the opportunity to send one or a few extra lines, and a shared window update quickly follows, a workstation cannot process the window update until it has received the preceding extra lines, after a long retransmission delay if one of the preceding extra lines was lost in transit.

The price paid for the above flexibility is that a non-sequential protocol will almost always require a larger volume of data to be transmitted (e.g., the entire contents of a line), compared with sequential protocols that carry more compact "relative" updates (e.g., an Insert-String operation). Non-sequential protocols are therefore best suited for environments where bandwidth is not a scarce resource, e.g., a local area or satellite network that is not heavily loaded, or a few such networks linked together by high-bandwidth gateways.

4.6.4.3 Non-Sequential Initialization

While non-sequential protocols are expensive for transmitting small incremental changes to large objects, they can be expected to perform no worse, and perhaps better, than a reliable sequenced protocol (whether virtual circuit or multicast) when data such as document lines are sent to the workstations for the first time. In this case, the entire contents must be sent anyway, and there is no special reason why a workstation must receive all replicated document lines in the exact same order that they were sent. Preliminary experiments with a non-sequential "file transfer" protocol designed by Reed show that the overall delay in transferring a file is smaller than with a conventional protocol that requires blocks of a file to be processed strictly in sequence, provided appropriate "flow control"

measures are taken to ensure that the receiver is not sent data faster than it can process.[20] It therefore seems plausible that a non-sequential protocol could be used for adding a new workstation to a conference, or for sending a new large object to the existing group of workstations. Normal processing can then proceed, using a sequential protocol for relative updates, once acknowledgments have been collected and all workstations are known to have the data that was sent. Or, data can be sent asynchronously using a non-sequential protocol, provided acknowledgments for each component object are collected no later than when the object is needed for display or for a relative update operation; this was illustrated above in the case of "extra" document lines sent by the controller. While experience with non-sequential protocols is currently limited, this combination approach seems promising enough to warrant further experimentation.

4.7 Improving Interactive Response

The conference architecture we have assumed so far requires a command from a participant to be forwarded by his workstation to the controller site, which then reports the results of the command to all workstations, including the originating workstation. The observed "response time" for all actions on the shared space is therefore always at least two message transmission delays. (Processing delays are also incurred, which may be significant if the controller or workstations are overloaded, but two message transmission delays is the absolute minimum that can be achieved. The only exception to this is for a workstation that is also the controller of the conference.) This contrasts with actions on a participant's private space, which are executed locally by the workstation to give the participant almost immediate response. The reason for the extra delay in a conference is that the shared space may be updated concurrently by other participants, and approval of commands from the controller is therefore required in order to ensure consistency.

Response time in a conference can be improved if a workstation is allowed to locally execute commands on the shared space, and display their effects to the participant, *before* approval is obtained from the controller. The workstation makes an optimistic guess that the controller will approve the operation and will do so relative to other participant's operations in

[20] This information was obtained from Larry Allen in a private conversation.

such a way that its effects will be the same. Since this may not turn out to be true, e.g., the controller takes away this participant's reservation or processes a conflicting operation from another participant before this participant's operation, the workstation must perform the operation *reversibly*. That is, the workstation must save sufficient information (old values of changed or deleted objects, or an "inverse" operation) to allow the operation to be "undone" and then reexecuted on the correct state as specified by the controller.

Even with the above improvement, *other* participants' workstations will see the effects of a given participant's operations only after two message transmission delays: the results must still be received from the controller, which must first receive the input message from the originating workstation. This delay can also be reduced, to a single message transmission time, by having workstations send input messages directly to each other and allowing workstations to immediately execute operations received from other workstations before approval by the controller. Again, the operations are performed reversibly, allowing them to be undone and redone if necessary.

With direct transmission and immediate local execution of operations (carried in input messages), a consistent ordering of input messages must still be ultimately established; this is where the controller site comes in. The following "many-to-many" reliable multicast protocol (as opposed to the "one-to-many" reliable multicast of Section 4.6.3) can be used to achieve this. The protocol is described below in the form presented by Section 3.5.2 of Montgomery's thesis [87]. The original protocol does not allow for immediate execution of operations out of sequence, which we will discuss after the basic protocol is presented.

4.7.1 Many-to-Many Reliable Multicast

The following description assumes "full replication" of the shared space, so that every workstation is capable of executing input messages locally; partial replication will be discussed presently. While full replication at all workstations does not require a separate controller site, the protocol works equally well whether or not the controller site is itself a workstation.

We also assume that all messages below are transmitted as unreliable *multicast datagrams* using any of the methods in Section 4.6.3.1; retransmission procedures are therefore included. Reliable virtual circuit communication may be used between pairs of

workstations, avoiding some of the retransmissions below but not all. In particular, if a workstation crashes after sending a given message to some workstations but not all, the other workstations will have to request the controller to retransmit the message; pairwise reliable sequencing is not sufficient when there are more than two sites communicating. Virtual circuit communication between each pair of sites will also require a number of virtual circuits that grows as the square of the number of sites, i.e., N*(N-1)/2 for N sites. For large conferences (more than three participants, say), this overhead will usually be considerable. Unless the network supports multicast, or is of sufficiently high bandwidth to support a very large number of pairwise virtual circuits, direct communication between workstations should not be used for large conferences.

The protocol for many-to-many reliable sequenced multicast is as follows:

- Each input message sent by a workstation carries a unique *id* which consists of a site identifier (which might be an index into an array; a system-wide site-address is not required) and a local consecutively assigned sequence number. The message is sent to all workstations and the controller, unreliably, by the most efficient multicast method or simulation available. The workstation also pretends to "send" the message to itself, and performs the actions of a receiving workstation as described below.

- The controller on receiving an input message from a workstation (or from itself) assigns the message the next global *sequence number* in a consecutive sequence, and multicasts an "acknowledgment" message carrying the assigned sequence number and the id of the input message. (The controller will typically assign sequence numbers to messages from different workstations as it receives them, but may choose to delay messages from different workstations as described in Section 4.3.2.) For some number of recently-assigned sequence numbers (how many is discussed below), the controller remembers the id and contents of the input message that was assigned each sequence number in a retransmission queue. The controller can thus handle duplicate input messages and respond to retransmission requests as described below.

- A workstation retransmits an input message, using multicast, as often as necessary until it receives the acknowledging sequence number from the controller; the controller may have missed the original message, or the workstation may have missed the acknowledgment. If the controller receives the same input message twice (determined by comparing message ids in the controller's retransmission queue) as a result of such

retransmission by the workstation, the controller simply reports the sequence number that has already been assigned to the message. A workstation may "pipeline" input messages by sending a new one before the acknowledgment of the previous one has been received; the controller on receiving an input message without having received the previous one from the given workstation will not assign it a sequence number until the previous input message is retransmitted and assigned a sequence number.

- When a workstation receives an input message from another workstation, or from itself, it buffers the message until it receives the acknowledging sequence number from the controller and has processed all messages with the preceding sequence numbers.

- If a workstation receives an acknowledging sequence number from the controller for which it does not have the input message with matching id, it sends the controller a request for retransmission of the original message. The controller on receiving such a request looks up its retransmission queue and sends the original message (id plus contents) to the workstation. The retransmission request is itself retransmitted as often as necessary until the original message is received from the controller.

- If a workstation receives an acknowledging sequence number from the controller out of sequence, it does not know which message id was assigned the preceding sequence number. It therefore sends the controller a request for retransmission of the missing acknowledging sequence number and associated message id. The controller responds to such a request in the same way as above, extracting the sequence number and message id from its retransmission queue.

- A workstation issuing an input message also includes a sequence number indicating that it has received all messages through the given global sequence number; this will allow the controller to discard messages on its retransmission queue that it knows have been received by all workstations. If however a given workstation issues no input messages, and therefore no acknowledgments, for a long time, the controller's retransmission queue may grow arbitrarily long. The controller should therefore request an acknowledgment when its queue grows long, and remove a workstation that does not respond after the request has been repeated a few times.

An alternative method for controlling the size of the retransmission queue is described by Chang and Maxemchuk [15]. In this protocol, the controller site is moved around a virtual *ring*

that includes every workstation.[21] The current controller site passes the "token" to the next workstation in the ring, making the next workstation the controller, after receiving and acknowledging K input messages, for some fixed K. A workstation does not keep track of where the controller is located at any given time (except of course when it is the controller); requests for retransmission are multicast so that whichever site happens to currently be the controller can respond to them. Because a change of controller requires that the new controller acknowledge all previous messages, a complete traversal of the token around a ring of N workstations means that every workstation must have received and acknowledged at least all messages preceding the last K*N. The retransmission queues of all workstations (each of which must handle retransmission requests during its turn as controller) therefore need be only K*N messages long. Instead of collecting acknowledgments from all sites at once in order to bound the length of the retransmission queue, this method collects acknowledgments from one site at a time every K messages; the problem of acknowledgment "bursts" (Section 4.6.3.3) does not arise. (If the workstation holding the token crashes, the token is regenerated and the ring reconfigured by one of the other workstations playing "backup", using a procedure similar to that described in Section 5.2.5.)

4.7.2 Adding and Removing Workstations

Direct communication between workstations requires some additional synchronization when workstations are added to or removed from the conference.

When the controller decides to remove the workstation of a participant, for any of the reasons described in Section 4.5.4 (including a crash of the workstation), the workstation may have already sent an input message to some or all workstations that the controller will never acknowledge with a sequence number. This is not a problem, because when the other workstations learn from the controller that the given workstation has been removed, they can discard any input messages received from the given workstation that were not acknowledged by the controller.

Adding a new participant's workstation requires providing the new workstation with up-to-date state information, using any of the methods of Section 4.5. Once the controller has done

---

[21] Use of this method requires that it be permissible to move the controller site of a conference, a possibility that we discuss in Section 5.2.4.

this (outside of the multicast mechanism, e.g., using a separate virtual circuit or non-sequential protocol), it instructs the new workstation to start listening for multicast messages and acknowledging sequence numbers. The controller also tells the new workstation the next global message sequence number that it should process; this allows the workstation to discard older messages that it may receive, and to request a retransmission if it misses the expected next message.

4.7.3 Displaying Unconfirmed Results

The above protocol establishes a consistent ordering for the input messages issued by all workstations in a conference. Our extension allows a workstation to process a received input message *immediately* without waiting for the acknowledging sequence number to arrive from the controller, in the hope that the controller will assign sequence numbers in the same order that the workstation processed the messages. Then, if a workstation learns from the controller that it processed a given message or messages out of sequence (i.e., the next sequence number received by the workstation is assigned to a different message), the workstation will undo the effects of the given message(s) and then "redo" them in the correct order. (If the controller instead indicates that a workstation has been removed, unacknowledged input messages from that workstation are simply undone and discarded.) A further optimization allows workstations to reduce the amount of undoing and redoing: If two input messages are known to *commute* (e.g., insertions or deletions in different documents or in disjoint regions of the same document), they do not have to be undone and redone when it is determined that they were executed out of sequence. Once a message is known to have been processed in the correct sequence, its undo information can be discarded by the workstation.

Since workstations do not wait for the acknowledging sequence number when processing an input message, the controller need not acknowledge every input message immediately on receipt. The controller can instead wait so as to "batch" a series of several acknowledgments into a single transmitted packet. This is possible because each acknowledgment, consisting only of a sequence number and the original message id, is quite small; a series of sequence numbers can be further compressed into a single sequence number and a count. This batching is especially useful when workstations are generating input messages quite rapidly; issuing an immediate acknowledgment for each input message will further increase the

already-high network load and degrade response because of packet collisions and the need for subsequent retransmission. In certain situations, the controller should not batch acknowledging sequence numbers but instead send them out immediately. If the controller sees two conflicting (i.e., non-commutative) operations from different workstations arrive close together, it should inform workstations of the correct order immediately so that any workstation that did perform the operations out of sequence will rectify the inconsistency seen by the participant as quickly as possible. The controller should also not delay the acknowledging sequence number for a "drastic" operation (discussed below) which it knows the workstations will not process until acknowledged, nor should it hold on to one or a series of unsent acknowledgments for a very long time.

Immediate execution of input messages received by a workstation allows participants to see results of other participants' commands after only one message transmission delay instead of two; the participant issuing a command gets immediate feedback, just as if he were running the application locally on his workstation. (No message transmission delay is involved when a workstation "sends" an input message to itself.) The price paid for this improved response is that the results shown to the participants may sometimes be inconsistent. Updates that "never happened" may be shown to the participants and then be undone and redone with different effect. Whether it is in fact permissible to let participants see temporarily inconsistent data is a question for the designer to resolve. Our intuition is that it is not unreasonable for minor changes, such as small insertions or deletions, so long as the frequency with which displayed results have to be undone is low. For more drastic changes (e.g., the contents of a shared window completely replaced by a new document) it is better to wait until the correct order is confirmed by the controller. A workstation is therefore not required to execute all input messages immediately on receiving them; if an input message would cause a drastic change, it is buffered and processed only after the controller has supplied an acknowledging sequence number and all previous messages have been processed. For such operations, the response time seen by the participants is again two message transmission delays; this is not unreasonable because the more drastic operations are the ones for which participants are likely to expect and tolerate a longer delay.

In addition, steps can be taken to reduce the probability of conflicting messages being executed out of order, so as to minimize the unpleasantness to the participants. We first require that a workstation not execute input messages out of sequence if they were sent by

the same other workstation, on the assumption that consecutive commands from a given participant (e.g., two characters to be inserted into a document in order) will usually not commute. This is easy to ensure. If a given message from a workstation is lost, a receiving workstation will detect a "gap" in that workstation's message id sequence when it receives the next message from that workstation, and can wait until the retransmission mechanism delivers the previous message. We therefore look at possible conflict between input messages issued by different workstations. In the case of objects for which only one workstation can ever initiate updates (e.g., the positions of the workstation's "mouse"), commutativity is automatically ensured. (It may still be necessary to undo an unacknowledged message if the workstation crashes.) Or, if "reservations" are set so as to discourage workstations from issuing concurrent conflicting operations, input messages from different workstations will commute most of the time. (Using reservations does not completely eliminate the problem. A workstation that believes it holds a reservation may issue an input message while the controller is taking the reservation away; the execution of the message will have to be undone when it is learned that the reservation was taken away before the controller received the input message and assigned it a sequence number.) On the other hand, if reservations are not used the probability of conflicting concurrent operations being executed out of sequence is considerably higher. (This probability will also increase with the delay and unreliability of the multicast transmission method used.) If "validation" of input messages is being performed, based on which version of the shared space was seen at the time, the situation is even worse because many commands may have been based on completely inconsistent data and will therefore be rejected when the correct ordering is determined. We observe that reservations, which are intuitively appealing from a user interface standpoint, are also useful in allowing the implementation to be optimized.

The above description assumed that the shared space is fully replicated and all workstations are capable of executing all operations locally. The ideas can still be used with partial replication, on a limited basis. Thus, with partial replication of document lines as in JEDI, a workstation can locally execute, and directly transmit, editing operations for which all of the affected lines are in the currently replicated set; if not, the operation is forwarded to the controller for processing and generation of results. For operations received directly from a workstation, the same undo and redo strategy described above can be used when the correct ordering, relative to all other operations, is determined from the controller.

4.8 Using the Implementation Techniques

The techniques presented in this chapter are intended to help the designer maximize performance, especially speed of response to participants' commands, for a given conferencing system functionality, architecture, and implementation environment. We do not address a complementary problem that the designer must face, namely how to choose the functionality and implementation method in order to satisfy a given response time constraint. (For example, response time for character echoing and other simple commands should usually be within a quarter or a half of a second.) This must be based on knowledge of the processing and bandwidth requirements of the different functions being considered for the specific application, and some functionality will usually have to be sacrificed if bandwidth is a scarce resource. (For example, the shared bitmap facility of Chapter 6 is ideally suited for a high-bandwidth network; performance and response time may not be adequate over long-distance networks of lower bandwidth.) Performance will also degrade as the number of participants in a conference increases, because of the increased load on the network and on the processors, especially that of the controller site. The designer, or the participants in a conference, may have to place an upper limit on conference size.

Since different implementation methods are appropriate for different conference sizes and different performance characteristics of the workstations and of the network, the designer may wish to have the conferencing system select an implementation method for each conference dynamically at run time. This problem, together with other architectural considerations, is discussed in the next chapter.

Chapter Five

Conference System Architecture

The previous chapter presented implementation techniques and discussed how they might be selected for a given real-time conference. This chapter describes an architectural organization that permits many of the implementation decisions for each different conference to be made at "run-time" based on information about the performance and capabilities of the workstations and the network. We then discuss several issues related to the interaction between individual conferences and the overall system environment, such as initiating, moving, and combining conferences, and interaction with sites providing permanent storage and other services. Finally, we briefly review known alternatives to the method of centralized control that we have presented for real-time conferences. Close examination of these decentralized approaches reveals that their performance is likely to be worse than with centralized control, except for the special case of conferences in which only two sites participate. (This conclusion applies to real-time conferences in particular, not to distributed systems in general.)

5.1 Dynamic Conference Control

For a given real-time conference, there are several parameters describing the protocol used by the controller and workstations for replicating information and communicating input and update messages:

- The "level" of replication: which objects or components of objects are replicated, and sizes of replicated objects or limits on the number of replicated objects of a given type.
- What operations are to be used in input and update messages, and what code is needed in order to interpret them.
- Which operations, if any, are to be implemented by direct transmission between workstations and immediate local execution.
- Parameters relating to the lower-level transport mechanism, such as the encoding of operations into byte streams or packets, and addresses to be used

for multicast or non-sequential datagram communication.

If all of the above decisions are statically determined by the designer, for all conferences that will be run using his conferencing system code, this will yield an inflexible system. If the parameters are chosen assuming powerful workstations and a high-bandwidth network, then high performance will be achieved with such workstations but it will not be possible to participate in a conference from a lower-grade workstation. On the other hand, if the system parameters are chosen to accommodate lower-grade workstations, the better performance that is obtainable with higher-grade workstations will be ruled out. It is therefore desirable to leave at least some of the above decisions until "run-time", when a conference is started and perhaps even during a conference when participants join and leave or as network conditions change. The architecture we are about to present, which we have named *Ensemble*, provides a mechanism for such dynamic selection of parameters. The Ensemble architecture also allows for multiple groupings of workstation sites, for cases where different participant "subgroups" are allowed to see different information, or to implement different levels of replication and protocol for different sets of workstations based on their capabilities and available communication bandwidth. For example, the controller might use full replication and/or a more verbose update protocol for a workstation that is connected to it via a high-bandwidth local network, but might use partial replication and a more compact protocol for another workstation that is connected via a slower long-distance line or network. While such complex configurations may not often be needed in practice, the architecture is meant to be general enough so as not to exclude them.

5.1.1 Groups of Sites

Every real-time conference has a controller site, and zero or more other sites at any given time. These other sites may be participants' workstations or "server" sites providing file storage or lookup facilities; for simplicity we shall refer to them as "workstations" in the following.

The sites involved in a conference are organized into *groups* whose memberships may change over time. (A given group may temporarily contain as few as one site or even none.) Groups may overlap, allowing a given site to be in more than one group. The controller itself may be in some group, e.g., if it is also a participant's workstation.

Each site group represents an agreement by the controller to replicate some subset of the conference object at every site in the group, using some set of allowed input and update messages that is the same for all sites in the group. (Different groups may use different protocols for the same object(s).) Thus, multiple groups can be used to represent different levels of replication or message types, as well as to represent "server" sites whose role in the conference is typically different from those of the participants' workstations. Different groups will also be used to represent workstations in various "transient" states, e.g., while being brought up-to-date prior to being added to some group(s).

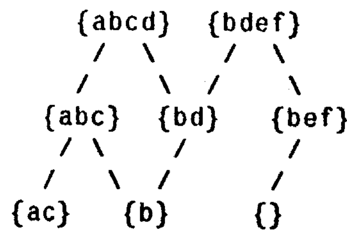

Figure 5-1: Example Group Graph

The different site groups in a conference are themselves organized in a partial order which we call the *group graph*. An example group graph, with group memberships indicated, is shown in Figure 5-1. Every site in a given group must be a member of every immediate ancestor of the group. Groups in the graph that have no ancestor groups are referred to as *root groups*.

The parameters governing the protocol in a given group may be *static*, i.e., embedded in the code executed by the controller and workstations, or *dynamic*, i.e., determined at run-time and modifiable for each individual conference. If the protocol is dynamic, some *meta-information* that uniquely describes the protocol must be included in the conference object, and this meta-information must be replicated in the immediate ancestors, in the group graph, of the given group. (For a group that has more than one immediate ancestor, it is sufficient that the union of the information replicated in the immediate ancestors include all the necessary meta-information.) The protocol for a "root" group must be static and well-known throughout the system.

Protocol meta-information for a given group may take any of several forms, depending on

what aspects of the protocol have been chosen by the designer to be variable. Examples are:

- Numeric parameters, e.g., size of a shared window or upper limit on number of document lines to be replicated.
- Boolean "flags" indicating whether or not a given option, e.g., direct transmission and local execution, is in effect.
- Several related boolean flags may be combined into a bit-string, e.g., indicating which of several possible operations may actually be used in update or input messages.
- Identifiers specifying what software a workstation will need in order to interpret certain update messages in the protocol associated with the group, that a workstation may use in order to retrieve the software from a "code server" (Section 5.2.7). The protocol meta-information may also include the necessary code itself, allowing the controller to "download" code to the workstations. This allows the controller to download specialized routines that are not known to the system's code servers, or to "multicast" the routines to all workstations more efficiently than if each workstation separately retrieved them from a server.

In order for a site in a group to send correctly encoded input messages to the controller, and to correctly interpret update messages, the site must know all of the protocol meta-information for the given group. Unless the group is a root group, its protocol meta-information can only be obtained from its ancestor groups. Thus, a site can be added to a group (other than a root group) only if it is already a member of all immediate ancestor groups. (Conversely, a site can be removed from a group only if it is a member of no immediate descendant group.) Because the protocol for a root group is well-known throughout the system, any site can be added to a root group at any time.

A *private group* has a fixed membership consisting of only one site. Private groups are used for replicating site-specific information at the given site and no other, such as the private paste buffer and private window positions in JEDI-CONF. A private group is also used for replicating meta-information describing what class of protocols the given site is capable and desirous of supporting, e.g., number of lines of storage or level of update operations. A private group may be a root group, if its protocol is well-known throughout the system. Or, a private group may be a descendant of another group, one that contains meta-information describing the private group protocol, or one in which membership is required in order for the private group to exist.

The group graph of a conference may itself be statically determined by the conference system code, or may be dynamic. The information replicated in a group that has a variable

number of immediate descendant groups must contain sufficient meta-information to identify all of these groups, in addition to the necessary protocol parameters.

5.1.2 Conference Descriptions

The information replicated in a given group may include application data in addition to the protocol meta-information required for each immediate descendant group. The information replicated in a root group is called a conference *description*. Conference descriptions may be different for different root groups, but we expect that in many real-time conferencing systems there will be only one root group or the different root groups will replicate the same information; we therefore loosely refer to a conference description as "the" conference description. The conference description will contain both protocol meta-information (for the descendants of the root group) as well as some user-oriented information (such as subject and time and document names) that will help a user decide whether or not he wishes to join any descendant groups. A site can be added to a root group when the controller directly sends it a copy of the conference description, or when the site informs the controller that it has obtained a copy of the description from some other site, typically a "lookup server". The root groups, and their associated conference descriptions, are thus the starting points for joining other groups of interest in the conference.

5.1.3 An Example

Not all of the above generality will necessarily be used in a given real-time conference system; we expect that a fairly simple group structure will often be adequate for the designer's purposes. For example, JEDI-CONF has a fixed number of multi-site (non-private) groups arranged in a static structure; the only dynamically-created groups are private groups of workstations that join and leave, and of file servers. This is illustrated in Figure 5-2.

The group *Actives* is the main group of workstations actively participating in the conference; copies of application objects such as document lines and shared window positions are replicated in this group. Each active workstation is also in a private group, used for replicating the workstation's paste buffer and information about its private windows into shared documents.

The root group *Workstations*, the only ancestor (and therefore a superset) of Actives, consists of workstations at which the conference "description" is replicated, using a well-

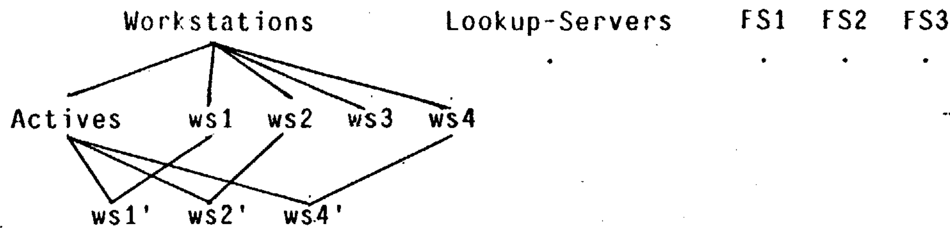

Figure 5-2:Group Graph for JEDI-CONF

known protocol. The conference description includes protocol meta-information for the Actives group, namely `line-limit` and `super-window-size`. The following information is also included in the conference description, for the benefit of potential participants:

- The conference `subject` string.
- Access control information: `owner`, `allowed-users`, and `public-access`.
- Estimated and actual start and end times of the conference. (Actual times may be Undefined; an actual end-time that is not Undefined indicates a conference that has terminated.)
- The current participants: `npartcs` and the arrays `partc-names`, `ws-addresses`, and `ws-statuses`.
- The names of the documents in `working-set`, and their file server addresses. A user can thus get an idea of which documents are being viewed and edited in the conference, and can even retrieve copies of one or more from their file servers if he wishes to browse before or while joining the conference.

Each workstation in the Workstations group is also in a private group used for replicating information about whether the workstation requested to join the Actives group, or was invited to do so, or both, and for transmission of document and window contents prior to adding the workstation to Actives. Note that active workstations are in two private groups, because additional private information (paste buffers and private windows) is replicated only for active workstations. Workstations not in Actives (`ws3` in the figure) are in only one private group. A workstation that is in Workstations but not in Actives is in a "transient" state prior to possible addition to the Actives group, or just after removal from Actives.

The root group `Lookup-Servers` is a set of server sites (Section 5.2.7) to whom the conference description has been sent, so that users may find out about the conference when they query a lookup server. The information replicated in this group (the conference

description) is in fact the same as for the Workstations group, but these are distinct groups because lookup servers and participants' workstations are treated differently; updates to the conference description are sent to lookup servers with less urgency than to workstations, and lookup servers are never considered for possible inclusion in Actives.

Each *file server* in JEDI-CONF (**FS1** through **FS3** in the figure) is in a private group in which documents, and associated reservation information, located at the file server are replicated. Protocols for communicating with file servers are assumed to be well-known throughout the system, and each of these groups is therefore a root group.

The group graph and membership for a real-time conference need not actually be represented in a general graph data structure. Especially when much of the graph is static, its structure and group membership may be implicit in specially tailored data structures. Thus in JEDI-CONF the workstations in the Actives group are those whose corresponding element in the array **ws-statuses** is Active, while workstations in the Workstations group are those whose status is Active or Maybe. (A workstation's status can have a third value, Removed; this represents workstations which were previously in the Workstations group but are no longer in any group. The main function of this is to avoid the trouble of compacting workstation arrays and reassigning workstation "numbers" when a workstation leaves; thus, a given workstation gets a fixed workstation number for the duration of a conference.)

5.1.4 Adding and Removing Sites

We next describe the dynamics of how sites are added to and removed from groups; changes to the protocol within a group are described in the next section. Our discussion will use the generic operations on groups shown in Figure 5-3. Each operation name is prefixed with which kind of site can perform the operation: controller (**C**) or workstation (**W**). For each operation type, a corresponding message type is transmitted between controller and workstation or vice versa. We note that these are "abstract" message types and need not be actually represented in the exact same way in an implementation. Efficient message encodings tailored for the particular real-time conferencing system will typically be used; for example, if there is only one site group that workstations may be invited to or request to join (e.g., Actives in JEDI-CONF), then there is no need to include a "group" argument in any of the messages. Furthermore, a given transmitted message may specify more than one of the operations shown. For example, an "invitation" message from the controller to a participant's

workstation in JEDI-CONF carries both an Add-Workstation operation for the group Workstations (in which the conference "description" is replicated) as well as a Will-Add operation for the group Actives indicating that the controller would like to add the workstation to the active group; the latter invites the workstation to join by issuing a Please-Add operation.

```
C:Will-Add(group,workstation)
W:Please-Add(group,workstation)
C:Add-Workstation(group,workstation)

C:Will-Remove(group,workstation)
W:Please-Remove(group,workstation)
C:Remove-Workstation(group,workstation)

C:Propose-Protocol(group,activity,value)
W:Set-Capabilities(group,activity,value)
C:Set-Protocol(group,activity,value)
```

Figure 5-3:Ensemble Operations on Groups

The protocol used by the sites in a given group is assumed to consist of one or more *activities* such that the parameters governing a given activity can be varied independently of the other activities. The parameters for one or more activities may in fact be statically determined and fixed while other activities may vary. For example, it is possible to vary the activity used for replicating document lines without changing the activity used for requesting and releasing reservations and reporting the results of reservation operations. Or, the designer may wish to define independent activities for input, from workstations to the controller, and for updates, from the controller to the workstations. For each activity that is variable, a workstation can inform the controller of its corresponding *capabilities*, i.e., what parameters or range of parameters the workstation can support, using the Set-Capabilities operation. A workstation will typically perform such an operation in response to a Propose-Protocol message, described in more detail below in Section 5.1.5. The controller will remember every workstation's stated capabilities in its data structures and assume that they hold until further notice. If a workstation has never issued a Set-Capabilities for a given activity, its capabilities are unknown to the controller; this the controller can represent using the special value "Undefined". A workstation may "change its mind" about its capabilities at any time (e.g., if its available storage changes drastically, or the participant makes less or

more screen space available), and perform an unsolicited Set-Capabilities operation that invalidates any previous Set-Capabilities for the same activity.

A site is added to a group when the controller issues an `Add-Workstation` operation, which also causes a message to be sent to the given site informing it thereof. (Other sites in the group may or may not be informed, depending on whether site "status" information is replicated in the group.) Before a site can be added to a group, certain prerequisites must be met in order that the site may issue correctly encoded input messages and correctly interpret update messages on joining the group. First, the site must be provided with up-to-date copies of objects replicated in the group, using any of the methods described in Section 4.5 for initializing a workstation. Second, the site must know the current parameters governing the group protocol; except for a root group, whose protocol parameters are static, this is done by first ensuring that the site is in all immediate ancestors of the group to which it is being added.

In addition to the above prerequisites, there are certain conditions that the controller should, but is not required to, establish. First, the workstation should have indicated its desire to join the given group, by a Please-Add operation. Second, the controller should check, by having been informed of a Set-Capabilities operation, that the workstation is in fact capable of supporting the group's current protocol. (The workstation may have issued a Set-Capabilities of its own accord, or in response to a Will-Add operation by the controller that "invites" the workstation to join the given group.)

The controller in some cases may add a workstation to a group "optimistically", before it has determined the workstation's desire to join the group and ability to support the current protocol. A simple example of this is the root group "Workstations" in JEDI-CONF, to which the controller adds workstations unilaterally when it "invites" them to join the Actives group. Or, the controller might add a workstation to the Actives group, say, even though the workstation's stated capabilities do not match the current protocol. The controller indicates by doing so that it wishes to allow the workstation into the group but does not wish to change the protocol to accommodate the new workstation's reduced capabilities. The workstation must instead choose between somehow changing its capabilities to match (if and when that is possible, e.g., by freeing up storage and/or screen space) or the leave the group (using a Please-Remove operation). If the workstation does not change its capabilities or leave within some period of time, the controller might then remove it from the group.

The controller removes a workstation from a group using the Remove-Workstation operation; if the workstation is still up and accessible, it is informed of this. When the controller decides to do remove a workstation may be any of the following, as chosen by the designer:

- An authorized user (e.g., the chairperson) issues an explicit command to remove the participant's workstation.
- The workstation indicates its desire to leave by issuing a Please-Remove operation.
- The controller times out on some needed response from the workstation and needs to remove the workstation in order for activity in the group to continue. An example is when communications with the workstation are blocked due to "flow control" (i.e., messages sent are not being processed and queues are full; the workstation may in fact have crashed) for a long period of time.

The only prerequisite for removing a workstation from a group, under any of the above circumstances, is that the workstation not be a member of any descendant groups.

5.1.5 Negotiating a Change in Protocol

The controller can change the protocol parameters for a given group using the Set-Protocol operation; it may change only one activity of the protocol, or issue several Set-Protocol operations simultaneously. The new protocol parameters will then be available to all immediate ancestor groups (whose union includes all sites in the given group) of the given group. When the controller should choose to change the protocol, and whether it should do so automatically or only on instruction from a user, is an issue for the designer to resolve; we describe below some plausible reasons for changing protocols.

In order to discuss prerequisites and desirable conditions for changing the protocol, we introduce some definitions regarding different types of changes. The *intersection* of two protocols, *p1* and *p2*, is the set of messages, permissible under both protocols, that a workstation could correctly send and interpret regardless of whether it believed the current protocol to be p1 or p2. Protocol p1 *dominates* protocol p2 if the intersection of p1 and p2 is identical to p2, i.e., every message in p2 can be correctly interpreted using p1. For example, if protocol p2 allows some collection of low-level update operations and p1 allows the same collection of operations, identically encoded, as well as some higher-level operations, then p1 dominates p2. Similarly, a protocol with a higher allowed message rate or storage limit dominates a protocol that carries an identical set of messages (or a subset) but with a lower

limit. Every protocol dominates the *null* protocol, which allows no input and update messages. A change in protocol *lowers* the protocol if the old protocol dominates the new protocol, and *raises* the protocol if the new protocol dominates the old. A change in protocol is *incompatible* if neither the old protocol nor the new one dominates the other.

"Lowering" the protocol for a group will in general result in degradation of either performance (e.g, more verbose messages requiring more bandwidth and delay) or interface functionality (e.g., smaller shared window or less interesting display formats). A plausible reason for lowering the protocol and accepting such degradation is to accommodate as many participant's workstations as possible; for example, the controller might consider lowering the protocol prior to or after adding a new workstation that cannot support the current protocol. Another reason for lowering the protocol is if the current performance is unacceptable and can be improved by reducing the functionality offered and thereby reducing the needed bandwidth. (Performance can also be improved by removing one or more workstations, and possibly raising the protocol.)

In general, lowering the protocol in a group does not require any consent from the workstations in the group; if they are already capable of supporting the current protocol, they are certainly capable of supporting a lower protocol. However, if the controller lowers the protocol without giving the workstations a chance to react, the workstations may continue to issue input messages using the old protocol that are no longer valid under the new protocol. (When lowering the protocol, the old protocol permits a larger set of messages than the new protocol.) The controller can of course discard any such invalid input that arrives after the change of protocol, and inform the workstations of the "error", but this will require that participants retype commands that were thus rejected. A better approach is to first warn the workstations that the protocol may change, so that they stop issuing input messages, and then change the protocol after a brief interval. During the transient period, after the warning but before learning that the protocol has been changed, a workstation may buffer its participant's commands in a high-level form so that they can be later translated into input messages using the new protocol. (If the transient period lasts too long or the workstation's buffer grows too large, the workstation may instruct its participant to temporarily stop entering commands.) If one or more workstations are slow in responding to the controller's warning, or the warning itself is delayed, the controller may receive some invalid input messages after it has changed the protocol. The controller may still discard these messages; the important

point here is that warning the workstations first reduces the probability that this will be necessary.

"Raising" the protocol in a group will in general improve either performance or functionality. When one or the other is unsatisfactory, the controller may wish to raise the protocol at the cost of accommodating fewer workstations, because not all workstations may be able to support a higher protocol. (The controller might also raise the protocol when a workstation leaves or is removed and there is a chance that the remaining workstations can support a higher protocol.)

When raising the protocol, there is no need to halt the generation of messages because all messages issued under the old protocol will continue to be valid under the new protocol. However, not all workstations may be capable of supporting the desired new protocol, and it is useful (but not essential) to first determine whether or not they can. The controller may do this by checking the capabilities information that it previously received from the workstation; if the capabilities of all workstations already match the desired new protocol, the protocol can be raised immediately without any additional negotiation. Since the controller will usually have set the protocol as high as the workstations' capabilities allow, this situation will not occur often, except after a workstation with lower capabilities than the others leaves the group.

In the more common case where the controller wishes to raise the protocol and not all workstations' currently specified capabilities match, the controller can try to get the workstations to raise their capabilities by "proposing" a change in protocol and waiting for the workstations to issue new Set-Capabilities operations. A workstation might not be able to raise its capabilities, in which case it repeats its current capabilities in a new Set-Capabilities operation. When all workstations have replied, or when the controller gets tired of waiting after some timeout period, the controller can then make a decision to raise the protocol or retain the current protocol. If the protocol is raised, not all workstations may be capable of supporting it, and the controller may remove such workstations from the group; in the case of a workstation whose reply has not arrived, the controller may give it a little more time to respond to the change in protocol before removing it.

In the rare case of an "incompatible" protocol change, that neither raises nor lowers the protocol, it is useful to both "throttle" input messages from the workstations and elicit new

capabilities information from them. The controller can use the following *negotiation* procedure:

1. Issue a Propose-Protocol operation, which instructs the workstations to temporarily stop sending input messages and to issue a Set-Capabilities operation in response. A workstation may try to adjust its capabilities based on the proposed new protocol, or may be unable to; it will repeat its current capabilities in the latter case. The selection of new capabilities may involve an interaction with the participant, e.g., who may or may not wish to rearrange his screen.

2. After waiting for all workstations to respond, or for some timeout interval, select a new protocol (and decide whether any workstations should be removed from the group) and issue a Set-Protocol operation. The argument to Set-Protocol may be the same as the proposed protocol, or the same as the current protocol (which in effect "aborts" the proposed change), or some compromise. The designer can program the controller code to use any decision criterion he wishes, or to interact with a participant such as the chairperson in order to obtain a decision.[22] Workstations that are capable of supporting the new protocol can now resume sending input and processing updates.

Note that raising and lowering the protocol, as described above, are special cases of this negotiation procedure. A lowering of protocol generally does require halting input messages but does not require workstations to adjust their capabilities; a workstation therefore stops sending input but does repeat its current capabilities in a Set-Capabilities to acknowledge its awareness of the proposed change. A raising of protocol does not require halting input, but generally does require that new capabilities be specified.

"Starting" and "terminating" a conference are treated as extreme cases of raising and lowering the protocol, where either the old protocol or the new one, respectively, is the "null" protocol that does not permit any input and update messages. In JEDI-CONF, for example, a conference starts in earnest when values are chosen for **line-limit** and **super-window-size** (which are zero and Undefined, respectively, when the conference object is created). The controller may propose new values of "infinity" for these parameters, indicating that it wishes to select values as high as the workstations will permit; more typically, the controller's proposed new values will be based on its own internal resource constraints.

---

[22] If interaction with one or more participants is required during the negotiation, some "activities" of the protocol must remain unchanged so that the controller can present the available options to the participants and obtain a selection; the controller may also automatically "abort" the proposed change, or ask a new "chairperson", if no decision is received for a long time.

When terminating a conference, the controller issues a Propose-Protocol before actually doing a Set-Protocol to "null"; this gives conference participants a chance to finish any commands that they may be entering. A misbehaving workstation might not stop sending input messages, but the controller can always time out the negotiation and terminate the conference.

5.2 Conference Management

This section discusses several issues relating to the management of real-time conferences in the overall distributed system environment.

5.2.1 Multiple Conference Types and Applications

Most of our discussion has been oriented toward implementing a single real-time conferencing system for one application. In an large distributed system, there may in fact be many different real-time conferencing systems, for different applications. To organize all of these properly, it is useful to assign each such system a conference *type*, which is a globally unique string that identifies the associated code modules that controller and workstation sites must execute. Then, the conference type can be included in every conference object so that a workstation or controller knows what code is needed to interpret it and join or run the conference. If this code is not already available at a site, it can be retrieved from a "server" site by manual or automatic means. This can be further refined, as described by Schmidt [105], by assigning *version numbers* to newer versions of the code for a given conference type, and ensuring, when a workstation makes contact with a controller, that the version numbers on the code that they are running match. (The types are of course checked first to ensure that they match, i.e., the workstation and controller are not talking about totally different conference types.) Again, if the workstation has the wrong version of the code, or no version at all, it can retrieve the correct version from a code server.

An improvement on the above, which may not be possible to implement in every environment, is to allow many different "applications" to run in the *same* conference, e.g., a circuit editor and a document editor and a project planning subsystem in different "windows". We treat such a conferencing system as one integrated "application" that happens to include a very large number of object types and operations. (This is in keeping with the current trend

toward "integrated" user interfaces [108, 74].) If the associated code modules for this large application were to include routines for every possible object type and operation, they would be enormously large and may not even fit in the address space of a controller or workstation. This problem can be avoided if the system permits *dynamic linking* of routines. A conference can then be started with the controller and workstations running only a "kernel" conference module and then linking in additional routines for only those object types and operations actually used in the conference. When such dynamic linking is possible in a conferencing system, the protocol meta-information for a given workstation group will include identifiers of the routines actually used by that group. When linking in a new routine, the controller may use the negotiation procedure of Section 5.1.5 to check whether or not all workstations already have the routine. Those workstations that do not may first obtain it from a server and report back to the controller, at which point the negotiation terminates successfully and the routine can be used in the protocol for the group (i.e., invoked by input and update messages). (The controller may of course time out if a workstation does not reply for a long time.) The controller may also "download" new routines to the workstations, either unilaterally or after checking which workstations already have a copy.

5.2.2 Selecting a Controller Site

When a user wishes to start a new real-time conference, the following sequence of steps and decisions must be followed. Some or all of these may be automated, whereas others may require action by the participant initiating the conference; the choice is up to the conference system designer. It may even be possible to automate the initial decision to start a conference, by previously instructing some site to start the conference at some specified time.

First, a controller site must be selected for the new conference. This may be based on several criteria, such as processing capacity and proximity in the network to the desired participants' workstations or to data needed by the conference. (The controller might run on the same physical host as a file server, for example.) The initiating participant's workstation might be selected to be the conference controller if it has sufficient processing power. In other cases, a "server" site might be selected; this may be necessary if a trusted server is needed in order to enforce an access control policy based on all participant's privileges rather than just the "chairperson's". The addresses of conference control server sites in the system should be well-known to the workstations so that the latter may send a message

requesting that a new conference be started.[23]

In some cases, a conference control "server" may not itself control any conferences but provide a mechanism for dispatching new processes on its host machine to act as conference controllers. This particularly useful when there are multiple conference "types" in the system; a single site can be the controller server for many or all of the different conference types, and can load the appropriate code into the new process that it creates for each conference. (In such cases, a workstation should be prepared to be "invited" to a conference by a different controller site than the one it initially contacted.) In a system without well-known conference control servers, a user may have to use remote login to "manually" start a conference controller on the desired host.

Once a controller site has been selected for the new conference, the shared space of the conference must be set up. First, some small amount of information (the conference "description" and possibly other data that has been selected in advance) must be collected by the participant's workstation outside of the conference and sent to the selected controller when making the initial request to start the conference. The conference controller must be supplied with at least one participant's workstation address in order for anything further to happen; the initiating participant's workstation therefore includes its own address in the conference description. (If the workstation is the conference controller, then there already is one participant in the conference.) Addresses of other participants to invite may also be provided to the controller in the conference description, or the controller may first make contact with the initiating participant's workstation ("inviting" it to "join") and then obtain these addresses via commands issued within the conference by the initiating participant. In either case, it will be convenient to allow the participant to supply participant *names* rather than workstation addresses, and let the controller perform name lookup (Section 5.2.7.1) in order to obtain the addresses.

5.2.3 Backup Controllers

A real-time conference may have one or more *backup controller* sites that are ready to take over the conference if its current controller crashes or wishes to relinquish control. (Takeover procedures are discussed below in the next two subsections.) A backup controller

---

[23] In some networks, *broadcast* can be used to locate servers without knowing their addresses [10, 9].

must support "full replication", i.e., maintain a complete copy of the entire conference object. In some cases, certain information in the conference might be considered "volatile" and not worth the expense of backing up, e.g., cursor positions and reservations. If so, this information is lost when a backup takes over the conference; the backup might assign some default cursor positions and let the participants take it from there.

Backup controllers may be added to a conference at any time, e.g., at the beginning of a conference to provide crash protection, or dynamically when the current controller wishes to transfer control of the conference. If the participants' workstations do not support full replication, the overhead of keeping separate backups and sending them all shared space updates may be prohibitive. It may therefore not be worth the trouble of having a backup controller, or one or two backups may be retained but sent updates very infrequently so as to maintain adequate response for the participants' commands. Again, there will be some loss of state if the current controller crashes and a backup takes over.

If on the other hand all (or some) of the participants' workstations do support full replication, each can serve as a backup controller. (One of the workstations may be the current controller as well.) Since a workstation receives updates from the controller in real time, there will be little or no loss of state if the controller crashes and the workstation must take over.

5.2.4 Moving the Controller Site

During a conference, it may be desirable to select a different controller site, for any of several reasons:

- The current controller site is anticipating a shutdown.
- The controller is a participant's workstation, and that participant wishes to leave the conference.
- To improve performance, e.g., after the participant group changes over time, a different controller site might be "closer" to the workstations; or, a new "chairperson" has been designated and it is desired to make his workstation the conference controller.

To describe how the controller site of a conference can be moved, we introduce a simple extension to the object model of Section 4.1.2 that allows objects to *migrate*. An object's "home site", where its primary copy resides, is initially its birth site, but the current home site is allowed to designate a new home site at any time as follows:

1. The current home site, say C_n, selects a new home site C_{n+1} and sends C_{n+1} a message indicating C_n's desire to transfer the object.
2. C_{n+1} returns a message indicating its willingness (it may refuse instead) and specifying which version of the object it currently holds. (The version number will be "zero" if C_{n+1} has no copy of the object.)
3. C_n sends C_{n+1} necessary updates, absolute or relative, to bring the latter's copy up-to-date, and then a message indicating that C_{n+1} is now the new home site.

The problem of object migration has been studied before in the distributed systems literature (e.g., [75, 38]). We describe below extensions that are needed in order to inform and resynchronize with sites (e.g., workstations in a conference) that are interested in the migrating object.

5.2.4.1 Resynchronizing with Workstations

The above procedure is used to move the home site of any object. For a real-time conference, where the home site is the conference "controller", a change of controller requires additional interaction with the participants' workstations. Specifically, workstations send "input" messages to the current controller, and must be made to stop sending such input and to start sending input to the new controller instead. This is accomplished using the following procedure. (The procedure is an extension of the procedure used in Section 5.1.5 for negotiating a change in protocol; moving the controller site can in fact be viewed as a special kind of "protocol" change.)

1. The current controller C_n informs all workstations that it is considering moving the conference. C_n does this at the same time that it tries to persuade C_{n+1} to be the new controller and brings it up-to-date; if C_{n+1} is itself a workstation supporting full replication, a single message will serve both functions.
2. Each workstation stops sending input messages and informs C_n that it has stopped.
3. Once all workstations have acknowledged and C_n has finished processing their input and has no more updates to send, C_n informs C_{n+1} that it is the new controller. (C_n may time out and go ahead with this step if one or more workstations are slow in halting input; once C_n names C_{n+1} to be the new controller, C_n will simply discard any further input that it receives.)
4. C_{n+1} takes over the conference and sends updated conference "descriptions" to all workstations, inviting them to "join".
5. When all workstations have replied, or when C_{n+1} decides to stop waiting, C_{n+1} starts processing input from the workstations.

Because of message transmission and retransmission delays, a workstation being invited by the new controller C_{n+1} may not have received all prior updates sent by C_n. Therefore, each workstation on "joining" reports back to the new controller the version number(s) on its copy of the conference object (or components thereof), so that the new controller can send the workstation any updates that it did not receive. If a workstation did not receive all prior updates, or if C_n moved the conference before every workstation stopped sending input, a workstation may be holding on to some input messages that are not known to have been processed by C_n. Therefore, C_n on moving the conference informs C_{n+1} of the last input message it did process from each workstation; C_{n+1} in turn informs each workstation as to which input messages from that workstation have already been processed. A workstation can thus determine which, if any, of its previous input messages were not processed by C_n, and retransmit these to C_{n+1}.

If input messages are directly transmitted to all workstations by "many-to-many" multicast, as described in Section 4.7, moving the controller site to a different workstation is even simpler. Since input messages are sent to all workstations anyway, there is no need to stop sending to one controller and start sending to the other; instead, the previous controller stops issuing acknowledging sequence numbers and the new controller starts doing so. It is not even necessary to explicitly inform every workstation of the new controller; they will learn about the change in controller implicitly when they start receiving acknowledging sequence numbers from a different site. Requests for retransmission of missing messages or acknowledgments are multicast, so that whichever site happens to be the controller at any given time can respond to them.

5.2.4.2 Locating Migratory Objects

When objects such as conferences are allowed to migrate, there may be sites holding obsolete location information that are not informed about the new home site. For example, a user not in the conference may have obtained a slightly out of date conference description from a "lookup server", and may try to join the conference by sending a message to the previous controller site. The following procedure deals with such situations. Again, this a general-purpose procedure that works for any object; it takes care of situations where an object moves many times during its lifetime, although a real-time conference will typically move only a small number of times if at all.

Sites holding references to, or copies of, an object also remember a *chain* of sites addresses that begins with the object's birth site and includes in sequence every site to which the object was moved; the last address in the chain is that of the site currently believed to be the home site. A site may send location information about an object, i.e., the chain, to other sites, either unilaterally because it believes the other site is likely to be interested, or in response to an inquiry or an attempt to access the object. A site that receives location chains about some object from more than one site retains the most recent, i.e., the longest of the received chains.

After an object migrates from C_n to C_{n+1}, the new home site C_{n+1} makes an attempt to inform every previous home site, C_0 (the birth site) through C_{n-1}, of the new location chain, because these are the sites that other sites holding obsolete location information will contact. C_{n+1} also sends C_n the updated location chain, to confirm a successful migration; until C_n receives this confirmation, it should not inform other sites that C_{n+1} is the new home site. (C_{n+1} may also inform other sites believed to be holding location information about the object, e.g., the workstations of participants in a conference.)

Using the above, if a site is unable to contact the site that it currently believes to be the home site of an object, the site may inquire of past home sites and may discover that the object has since moved. The inquiring site may then make contact with a more recent home site and either succeed or be informed of yet another migration of the object. If all past home sites are down or inaccessible, a site may even ask any other site that it believes is likely to know where the object has moved to. For example, a new participant's workstation, unable to contact the site that it believes to be the current controller, may ask other participants' workstations if they know whether the conference has since moved.[24] A site that is trying to locate an object may even *broadcast* a query [10], or search an expanding "perimeter" [51] by querying a few neighboring sites each of which queries another few sites until the object is found or the search aborted.

Remembering and propagating the entire "chain" of past home sites allows a receiving site to "authenticate" that a site claiming to be the home site was in fact passed the object, e.g., by verifying the "signatures" of sites in the chain. (A migration from C_n to C_{n+1} must be

---

[24] In the case of a controller crash, discussed below in Section 5.2.5, a workstation may return information about a new conference instead.

"signed" both by C_n on initiating the migration and by C_{n+1} to acknowledge successful completion.) If all sites are "trusted" then such authentication is not necessary. It is still useful to keep the entire chain, so as to know which sites to inform or ask about future migrations, but the chain may grow rather long. An overly long chain, unlikely as that may be, can be truncated to save storage and bandwidth; at worst, some past sites will not be informed if and when the object migrates again. If parts of the chain are thus forgotten, a *count* of the number of times moved must be maintained so that a site can determine the most recent location information received.

5.2.5 Controller Crash Recovery

If the current controller of a conference crashes, it is too late for it to designate a new controller using the above procedure. Instead, a backup controller that notices the crash (by monitoring the controller site and timing out) starts a *new* conference whose state is copied from whatever version of the original conference it has at the time. The backup then "invites" all participants' workstations to join the new conference, in the same way as described above. The description of the new conference will indicate that its state was copied from a particular version of the original conference; workstations can therefore continue to use their copies of objects from the old conference, reporting their version numbers to the controller, in the new conference.

If a workstation's copy of the original conference is behind the backup controller's, the backup can bring the workstation up-to-date in the new conference as described above. A controller crash raises another possibility that did not arise when explicitly migrating the conference, that one or more workstations may have received an update message that the backup controller did not. If so, the workstation's copy is "ahead" of the backup's, and must be undone; the backup either sends the workstation one or more "inverse" updates, if it has recorded these in a history, or else sends complete versions of objects for which the workstation is ahead. An alternative to this that does not lose updates is for the backup, on discovering that a given workstation is ahead, to ask the workstation for the missing update and bring the backup's version up-to-date; the missing update is also propagated to workstations that did not receive it.

To forestall the possibility that two or more backup controllers simultaneously try to start new conferences to replace the original one, a "priority" ordering can be defined for the

different backup controllers. A given backup controller then starts a new conference only if it is unable to communicate with all controllers with higher priority, including the current one.

It is still possible for a backup to start a new conference by mistake, because of a temporary communication problem. If so, workstations that are still in communication with the current controller, or are joining a conference started by a higher-priority backup, will instruct the backup to abort its new conference. A backup that is no longer able to communicate with any workstations will ultimately time out and abort its conference when no workstations join.

When a backup controller times out and starts a new conference, it may be the case that the original controller has not actually crashed but that a network *partition* has made it impossible for the two sites to communicate. Many solutions to the network partition problem allow a controller, whether the original one or a backup, to continue processing only if is communicating with a *majority* of sites. While this prevents the emergence of divergent conferences, it is very restrictive in that many partitions or patterns of crashes may leave *no* connected group of sites with a majority. (For example, one critical site out of 2N + 1 could crash and leave the network partitioned into two groups of N sites each.) In the interests of letting work continue, it is usually better to take the chance that there may be another divergent conference elsewhere in the network, with the intention of later "combining" or "merging" the two conferences (as described below in Section 5.2.6) if and when they can communicate again. This is the approach taken with replicated files in LOCUS, for example [94]. (Locus treats divergent copies as being alternative versions of the "same" object, whereas we treat a backup controller's new conference as a "different" object that was derived from the original one. This difference is for system modeling purposes only and is purely cosmetic. Merging of divergent copies requires the same use of version "ancestry" whether we treat the new conference as being the "same" or a "different" one. And, a participant in the conference need not be told that a "different" conference was started when the original controller crashes, although he should be given some feedback about the crash and takeover because he will notice a significant delay.)

5.2.6 Combining Conferences

As described above, it is possible for two or more conferences, copied from the same original conference, to evolve when there is a network partition. If and when the network is connected again and the two controllers are able to communicate, it is desirable to "merge" the shared spaces of the conferences and combine their participant groups once again. Even in other situations where the conferences in question are not known to have been copied from the "same" one, they may be similar enough to provide a reason for combining them. For example, a participant might find himself invited to two different conferences in which the same or related documents are being shared; he may wish to combine the conferences, or at least suggest combining them if he himself is not authorized to do that.

The following sequence of steps should be used when combining conferences. Note that each step may be triggered by an automatic decision by a conference controller, or be performed manually by a participant, or a mixture. We assume below that there are only two conferences involved; the rare case where three or more conferences are to be combined is handled by repeating the following steps for two conferences at a time.

- *Detection* of a need to combine conferences. In the case of two conferences derived from the same original conference, this will happen automatically when the their controllers are able to communicate once again. (A backup controller starting a new conference should periodically try to communicate with the original controller and other backups that it is unable to contact.) In other less obvious cases, a user may decide, on examining the descriptions or even the shared spaces of two or more conferences, that they should be combined.

- *Selection* of a *winning* conference, which is to take over the *losing* conference. This may be based on any number of possibly conflicting criteria. The "winning" conference could be chosen to be the one whose controller is "closest" in the network to more participants' workstations. Or, the conference with less data to transfer to the other could be selected as the "losing" one. If the state of one conference is identical to a previous version of the other, then the other conference should probably "win" because combining the two requires no data transfer at all. The selection might even be made arbitrarily by the participant who is bringing the conferences together. We assume that the two conference controllers arrive at a consistent decision; the case where each tries to take over the other, or each waits for the other to take over, is discussed below.

- Actually *combining* the conferences. The controller of the winning conference instructs the losing one to send the contents of its shared space. The winning controller then takes the "union" of the participant sets and of the data in the shared spaces. If the same object, e.g., a document, appears in both shared spaces, the more recent version will be chosen; if the object has been modified in divergent directions in the two conferences, the alternatives should be merged using an application-specific procedure. In either case, "relative" updates may be sent to reduce the amount of data movement. During this information transfer, all participants are informed of a merger in progress and input messages from workstations are throttled in both conferences, except possibly for commands to control the merger and to abort it.

- *Terminating* the losing conference, but retaining some information (e.g., a "superseded-by" or "copied-to" field) in its conference description so that participants who try to join it will be notified of the winning conference instead.

- *Continuing* the winning conference. The controller of the winning conference must now select a protocol for the new workstation group (or decide whether and how to use multiple groups) by conducting a negotiation to determine the capabilities of the workstations and which versions of which objects (from either conference) they hold. The conference then proceeds once all workstations (except for those that choose to leave, or are removed) have been brought up-to-date for the new protocol.

In some cases, detection of "similar" conferences may occur only after one or more of the conferences has terminated. It may still be useful to merge the "results" of the conferences, e.g., permanent document versions written, in some manner specific to the application.

Two problems must be avoided in the above procedure: a *deadlock*, in which the controller of each conference thinks the other one is the winner, and a *race*, in which each controller thinks it is the winner and tries to take over the other. The standard solution (e.g., in [81]) is to associate a globally unique *priority* (which may be simply the controller's site-address) with each conference and to always have the conference with higher priority be the winner. We propose an extension that is more flexible, in particular allowing a user to arbitrarily select the winner and loser, that uses priorities only when needed to break a deadlock or race.

Each controller, on learning about a possible merger with another conference, *suggests*

to the other controller which one should be the winner; the suggestion may be based on any of the above criteria, or specified by an authorized participant. If the two controllers happen to make the same suggestion, each acknowledges the other's suggestion to confirm the winner and loser; data can then be transferred and so on as described above. If instead the two suggestions are different, each suggesting that it be the winner or each suggesting that it be the loser, then priorities are compared. The controller that made the "wrong" suggestion, i.e., suggested that the lower-priority conference be the winner, then reverses its suggestion so that the two controllers agree. Note that the priority mechanism is invoked only when needed. If, for example, the decision is made by a participant who is in both conferences, the participant will (in a single command to his workstation) instruct one conference to be the winner and the other to be the loser; the "suggestions" of the two will therefore match, and priorities are not checked. Or, if one controller makes a suggestion before the other controller is aware of a possible conference merger, the second controller may simply agree to the suggestion.

Either controller may at any time *abort* the merger and inform the other controller. This it may do if the priority mechanism forces a choice that the controller, or a participant authorized to abort, does not like, or if a timeout period expires during which no response is received from the other controller.

5.2.7 Server Sites in a Conference

We describe some important functions that sites other than the controller and workstations might perform for a conference. These functions include reading and writing of objects or files from permanent storage, and lookup facilities for conferences and user names; unlike participants' commands, such functions will typically be invoked at discrete points in a conference rather than continuously. These functions are not strictly part of the conference architecture itself, but are important components of the overall distributed system. The techniques described below are standard in many modern distributed systems; our intention here is to describe how they would be used for real-time conferences in particular. Note also that different functions do not have to be performed by different sites; the same server site can perform more than one server function, and it is even possible for a participant's workstation to perform one or more server functions.

5.2.7.1 Name Lookup

A name lookup server resolves user names into workstation addresses. (Phone numbers might also be supplied, for manual or automatic dialing.) This chapter deals with participants in terms of their workstation addresses only, because the structure and semantics of user names will vary from system to system. But some form of user name lookup (such as provided by Grapevine [9] and Clearinghouse [92]) is needed as an alternative to manually typing in addresses supplied over the phone. A conference controller site can "invite" the user with a given name to the conference by determining the user's workstation address from a name lookup server and sending a copy of the conference description to that address; the workstation can then present the description information to the user and let him decide whether or not he is interested in joining.

In a system with multiple conference types, a workstation will typically register only a single "kernel" address with the name lookup server(s). On receiving a conference description, the workstation can check the "type" of the conference and perhaps spawn a new process ("site") that runs the associated code. Or, in an integrated system with dynamic linking, the workstation can link in the necessary code. The conference controller should be prepared for the response from a given user to arrive from a different address than the address to which the description was sent. There are in fact other reasons why the controller should be thus prepared, e.g., the address obtained by name lookup is obsolete and the user is now at a different workstation address.

Because addresses obtained from a name server may be obsolete, or a manually entered address may be incorrect, a workstation receiving a conference description from the controller should check that its user in fact has the same name as specified in the participant names array sent by the controller. How a name is determined to be the "same" we do not specify, and it may not always involve an exact string match. For example, a nickname or first or last name only may have been entered for the given participant. The user himself might have to look at the conference description and judge whether it is really addressed to the right person. Even when the user name does appear to match, the wrong person may have been invited; e.g., the user name was misspelled, or the lookup was done on a different host where the same user name is in fact assigned to a different person. In the extreme, the user may discover this only after joining the conference and observing the information or speaking to the other participants. Note that this problem is no different from that of misaddressed

electronic mail; "solutions" are beyond the scope of this research and usually require human judgment and intervention.

The controller site in turn, on receiving a message from a participant who wishes to join, may wish to verify that the participant is the person he claims to be, or that the participant is the same user that was invited, or both. This may involve an authentication procedure, e.g., based on encryption [89]. This may not always be available or sufficient, however, and "authentication" of participants may often be performed manually, e.g., by speaking over the phone with the joining participant.

5.2.7.2 Conference Lookup

A conference lookup server allows users to inquire about and discover conferences that they may wish to join. A conference can thus be made known to users, other than the ones explicitly "invited", by sending its description to a lookup server whose address is well-known to users' workstations. (More than one lookup server might be used for increased availability; if instead the system's lookup service has builtin replication, e.g., Grapevine or Clearinghouse, only a single copy of the conference description need be sent.) A conference lookup server will respond to queries about conferences from users' workstations; what kinds of queries it supports and what access control restrictions it places on conference descriptions will depend on the particular system. A conference lookup server may also provide both "planning" and "recording" services (below), thus allowing uniform access to information about all conferences past, present, and future.

5.2.7.3 Planning and Scheduling

A planning server keeps track of projects, tasks, appointments, and so on, and may perform automatic scheduling functions as well. Such a service, if available, can be used to "call" and plan a real-time conference well in advance of its actually being held, without tying up a controller site for several hours or days. Information that will ultimately go in the conference "description", e.g., allowed participants, scheduled times, or names of documents, may be entered and updated as the planning and scheduling of the conference evolves. This information could be used by a participant to browse over the information and prepare comments and suggest alternatives in advance of the conference, or by the participant's workstation to retrieve necessary code and copies of large objects in order to be

prepared. Addresses of participants' workstations could be entered in advance, avoiding the need to perform name lookup when the conference actually starts. (If an attempt to contact a given participant at the specified address fails when the conference starts, name lookup can still be performed to determine if the participant's workstation address has changed.) Similarly, a set of backup controllers can be specified in advance, one of which will probably be selected as the controller when the conference needs to be started. Who can do this and when, and whether concurrent updates to distributed copies are allowed, will depend on the access and concurrency control facilities available in the particular system. When it is time to start the conference, a controller site can be selected and provided with the description information that has already been constructed. (If by accident two or more users start conferences with the same description information, these conferences can be "combined" as described in Section 5.2.6.)

5.2.7.4 Recording

A recording server maintains histories of objects, such as conferences, and makes these available for lookup by users, perhaps according to specified access control restrictions. A recording server is sent update messages but stores them in the history instead of processing them against a current copy of the object. In some cases, a workstation, or the conference controller, may itself act as recorder; these sites will both record the updates and update their current copies.

5.2.7.5 Code Servers

A code server supplies, on request, code needed by controller and workstation sites for different conference types and different object types within a conference. How code modules are named and versions tracked will depend on the particular system. It is assumed that code servers have sufficient intelligence to send the appropriate compiled object code for different processor types. (If not, the users may have to do this manually.)

5.2.7.6 File Servers

Finally, a file server provides permanent storage facilities; these are used from a conference to read and write permanent versions of objects such as documents. A file server may present a very low-level interface, e.g., read and write entire files having unique "filenames", or may provide sophisticated database retrieval and update facilities. In the

latter case, the "file" server may also be able to perform planning, recording, code, and lookup services as described above. Or, in a system without such services, a file server could be used to "simulate" them by defining file name conventions for the different services and including sufficient intelligence in the controller and workstation code to find, update, and interpret the relevant files.

The file server for a given object is the site that carries access control information for the object, and it alone grants and takes away "reservations" on the document. The conference controller must therefore interact with the file server(s) "on behalf" of the participants in order to implement the desired access control policy. In JEDI, for example, the policy described in Chapter 2 is to require at least one participant with read access in order to read a document, and at least one participant with update access to set a reservation and write new versions. In addition, a given participant can edit the controller's volatile copy of a document only if he has update access to the document itself. As each participant joins the conference, and each document is added to the working-set, the controller will therefore determine from the file servers which participants have read and update access to which documents. In addition, the file server undertakes to inform the controller whenever any of this information changes as a result of actions by a document's owner. The file server may also bias its granting of reservations in favor of a conference, as described in Section 2.3.3.

In order for the above to work correctly, a file server must trust the conference controller to properly "authenticate" the conference participants, which requires that the controller be able to authenticate itself to the file server. The authentication problem is simpler if such sophisticated access control policies are not being used. For example, if all file reading and writing is done with the "chairperson's" privileges, and the chairperson's workstation is the controller of the conference, then the only authentication required is between the file server and the chairperson's workstation.

5.3 Decentralized Control

The implementation architecture and techniques we have presented are based on the assumption of a centralized controller site for each conference; this choice was made because of the simplicity with which a consistent ordering is established for all participant's actions on the shared space. An alternative organization that is used in many distributed

systems is *decentralized* control, where no single site is necessarily in charge. (We do not consider many "distributed" techniques that in fact rely on a "primary copy" for each object; with the ability of the primary copy to move, and of backups to take over on a crash, these are effectively indistinguishable from our centralized organization.) We examine decentralized methods in this section, and observe that they incur significantly greater communication overhead than centralized control, and are therefore probably not a good choice for real-time conferences. However, for the special case of two-person conferences this observation is found not to hold, and decentralized control does appear to be a viable option for this common special case.

Decentralized control requires that each site involved be capable of executing operations on the shared space without relying on some other site to supply the "results". We therefore assume full replication of the shared space at all workstations in this section; decentralized control cannot be used when workstations do not support full replication.[25] Decentralized control is also not appropriate when the access control requirements of a conference require a fixed controller site, e.g., the "chairperson's" workstation.

5.3.1 Locking and Timestamps

As described by Bernstein and Goodman [7], decentralized methods fall into two main classes: *locking*, and *timestamps*. With either method, a workstation wishing to perform a command issued by its participant must first obtain the approval of every other site; some methods [114, 36] relax this by requiring only a majority of "votes". Any of these methods could be used in a real-time conference as follows, assuming multicast communication:

1. The originating workstation sends the operation to all workstations.
2. Each workstation decides whether or not to approve the operation, and multicasts its vote to all workstations. Depending on the specific method used, the approval is based on whether or not some other uncommitted operation holds a conflicting lock, or a conflicting operation with higher timestamp has already been approved.
3. Each workstation on receiving every workstation's vote decides whether or not a sufficient number (all, or the required majority) approved, and executes or aborts the operation accordingly.

---

[25] It should be possible to construct a "hybrid" organization in which decentralized control is used among some small group of workstations that support full replication, while the remaining workstations transmit commands to and receive results from the workstations in this group. We have not worked out the details of such a scheme.

Two rounds of message transmission are involved, which is about the same delay as with centralized control. (Response time can be reduced to one message delay, just as with centralized control as described in Section 4.7, by immediate execution of a received operation, and later undoing if necessary.) The *number* of messages sent, however, is greater here than with centralized control. Every workstation sends one multicast message in the above, requiring N multicast messages to execute an operation when there are N workstations. Centralized control on the other hand requires only *two* multicast messages: one from the originating workstation, and one from the controller. Without multicast, the situation is even worse; centralized control requires N messages, while decentralized control requires N*(N-1). Except for the special case where N=2, which is discussed below in Section 5.3.3, decentralized control imposes an unacceptable communication overhead; the additional network load will degrade response time because of congestion or packet collisions, especially for large N. Decentralized locking and timestamping techniques are best suited for distributed databases in which most transactions affect only a small number of sites; in a real-time conference, this is not the case because every operation on the shared space must usually be seen by every workstation.

5.3.2 Token-Passing

Token-passing is a hybrid of decentralized and centralized control, in that at any given time one site is the "controller", but the controller site changes rapidly. Token-passing may be used in a real-time conference, assuming full replication at all workstations, as follows. The workstations in a conference are arranged in a virtual *ring*, such that each workstation communicates only with the next workstation in the ring. At any given time, one workstation holds the *token* and is the current controller. A workstation can issue a message only when it holds the token, and when it does it sends the message to the next workstation in the ring before sending the token. Every message received is processed and forwarded to the next workstation in the ring, until it reaches the originating workstation where it is removed and discarded. The ring structure and the circulating token ensure that every workstation sees all messages in the same order.

This conceptually simple method has an obvious problem when applied to real-time conferences, namely *delay*. A workstation wishing to perform an operation (a participant's command) must wait an average of half a trip around the ring before it receives the token, i.e.,

N/2 message transmission delays when there are N workstations. After that, each successive workstation around the ring sees the operation one additional message transmission delay later. Rapid circulation of the token may overload the communication network, whereas less frequent circulation (i.e., passing the token only after a brief wait interval) further increases delay.

We note that token-passing has been a very successful synchronization technique in local area networks [19]. This is because delay is not a factor when the token-passing algorithm is implemented in hardware; the question of "overloading" the network also does not arise because the token does not compete with other signals for use of the links between sites. When implemented in software, however, token-passing involves too much communication overhead and delay to be practical for real-time conferences. (The same conclusion was reached by Bernstein and Goodman for distributed databases [7].)

5.3.3 Two-Person Conferences

As described in Section 3.5, a two-person "conference" is different in some ways to a user than a conference with three or more participants. This section explores possible differences in implementation. We assume here that the services of a separate controller site are not needed in order to conduct the conference; only the workstations of the two participants need be involved. We also assume that the conference is intended to have only the same two participants for its duration; other users are not invited, nor is the conference "announced" at a lookup server for others to try and join. A conference that is announced to unnamed users, or to which more than two participants are invited, is not considered to be "two-person" just because it happens to have only two participants at some given time; the following will not apply.

With just two workstations interacting, only one point-to-point communication channel is needed, and many of the communication issues discussed in Chapter 4 become simpler. Multicast, for example, is no longer interesting or useful when there is only one other site to communicate with. A crash of either workstation, or a partition that prevents them from communicating, simply aborts the conference; the issue of "removing" a workstation and continuing the conference with a subset does not arise. (After aborting the conference, a workstation may still use its copy of the shared space for private interactions or to start a new conference with other participants.)

The above simplifications, and a different user interface, can be implemented for two-person conferences using our centralized organization: One workstation, presumably the one initiating the conference, plays the role of controller and is responsible for updating the shared space in response to its own and the other workstation's commands. This, however, imposes an asymmetry that seems unnatural for two-party communication. Most two-party communication protocols, as well as two-party telephone connections, are *symmetric* in that neither party has precedence over the other. We therefore consider decentralized methods for two-person conferences, for which the criticisms of Section 5.3.1 no longer apply. (When N = 2, the number of messages sent per operation is the same with decentralized and with centralized control.) We will assume below that decentralized control is done using locking; timestamps could be used with similar effect. Token-passing still does not seem appropriate, because a workstation issuing an operation must wait for the token if it does not already have it.

A workstation, say W1, wishing to perform an operation on the shared space needs to obtain a lock from only one site other than itself. W1 sets the needed lock(s) on its own copy of the shared space, and sends an input message describing its operation to the other workstation W2. The receiving workstation W2 knows that if it can grant the lock(s) needed, i.e., W2 did not issue a concurrent conflicting operation, then the operation must succeed. Therefore, W2 can execute the operation immediately and inform W1. A total of two messages are sent, as with a central controller. Response time for the originating workstation (W1 above) can be improved by immediately executing the operation reversibly, needing to undo and redo it only if a concurrent conflicting operation is received from W2 before the acknowledgment of this operation. In the case of conflicting operations issued concurrently by the two workstations, a "priority" mechanism can be used to determine which one is executed first; only one workstation will have to undo and redo its operation.[26]

With the above decentralized method, neither workstation holds the "primary" copy of the shared space. Consistency of the two copies not only requires that update operations be executed in a consistent order (ensured by the above), but also that the initial copies be

[26]This priority mechanism is the only source of asymmetry in an otherwise symmetric protocol. The priorities do not have to be permanently biased in favor of one site, e.g., each site could assign priorities to its operations randomly or from a clock, with only the least significant bit of the priorities biased to ensure uniqueness and prevent a tie.

consistent. This can be ensured by having each workstation start with its copy of the shared space in a standard state that is statically determined by the software, e.g., an empty "working set". (This is the approach taken in the TELNET virtual terminal protocol [25]; the "shared space" is however only a small set of option flags, not a complex application database.) Information is then added to the shared space, by either or both workstations, using update operations, and if concurrent conflicting operations are issued they are resolved by the above mechanism. Some optimizations are possible, in that if both workstations concurrently issue identical update operations each can acknowledge the other and the operation is executed only once.

With neither workstation taking primary responsibility for the shared space, more flexibility is possible in how information is replicated. Either workstation can add information to the shared space, e.g., submit a document from the participant's private database, and can "replicate" some or all of it at the other workstation, e.g., the other workstation need hold copies of only those document lines actually displayed. This applies equally in both directions, which means that each workstation may hold a subset of the information added by the other. With a central controller, one workstation holds "the" entire shared space and the other holds a subset; this is unnecessarily restrictive. (The same reasoning could be extended to groups of more than two workstations, but then the problem of workstations concurrently adding information to the shared space will involve more message overhead, as already described.)

The asymmetry of centralized control also introduces extra delay when conducting a protocol *negotiation*. When the controlling workstation proposes a change, the other workstation returns its capabilities and must wait for the controller's final decision. With decentralized control, a workstation can propose a change and specify its capabilities in a single message, and the other workstation on receiving this proposal can make a decision immediately based on its own and the first workstation's capabilities.

While we have not worked out the above in full detail, it appears that decentralized control is a reasonable method for conferences that are intended to have two participants only. Performance, in terms of message overhead and delay, is for most operations about the same for decentralized and for centralized control, but the symmetry of decentralized control allows faster negotiations and more flexibility in replicating data. Since decentralized techniques are not appropriate for larger conferences, and would in any case require more complex

programming if they were used, a decentralized two-person conference cannot "grow". If the participants in a two-person conference change their minds and wish to invite or add more participants, a new conference can be started using our centralized implementation techniques, and the state of the first conference copied into the new one in the same way as when "combining" conferences in Section 5.2.6.

Chapter Six

Prototype Implementations

This chapter describes two prototype real-time conferencing systems that we have implemented. Our purpose in building these prototypes was to demonstrate the feasibility of implementing real-time conferences using the techniques we have described, and to provide some insight into the difficult problem of selecting a reasonable set of conferencing functions and user interface.

The first prototype, RTCAL, supports the scheduling of a meeting by the exchange of information from participants' on-line calendars. This system was implemented in 1982 and, although limited in functionality and architecture, was a major influence in the development of the ideas presented in this thesis. The second prototype, MBlink, has just been completed (June 1984), and supports real-time conferencing on more advanced workstations over a network. Sections 6.1 and 6.2 describe RTCAL and MBlink, respectively, and Section 6.3 describes an extended version of MBlink that we designed but did not implement.

6.1 Meeting Scheduling in a Real-Time Conference

The prototype system RTCAL (for *Real-Time CALendar*) allows a group of users to exchange information from their calendar databases in order to select a suitable time for a future meeting. The focus of RTCAL is to provide users with information and tools that they may use to arrive at a decision in whatever manner they wish; it does not attempt to automate the selection of a meeting time, which is an interesting but separate problem [40]. Our selection of meeting scheduling as an application area was partly motivated by the availability of Greif's personal calendar system PCAL [45, 46]. RTCAL reads information from, and writes meeting times into, the personal calendar databases that users create and maintain using PCAL.

RTCAL was implemented in CLU [78] on a DecSystem-20™ mainframe. Although participants' "workstation" processes run on a single time-shared machine, we used

message-passing to simulate a distributed environment. The message-passing extension to CLU was implemented by Maurice Herlihy using algorithms described in [54, 53].[27]

6.1.1 RTCAL Functions

A user who wishes to schedule a future meeting using a real-time conference takes on the role of conference *chairperson*. The chairperson types in the names of the desired meeting participants, the range of acceptable dates for the meeting, the desired meeting duration, and a short description of the purpose of the meeting. A conference "invitation" message is then sent to those participants who are currently on line and running their calendar programs. (These participants could have been contacted by phone, and instructed to run their programs, just prior to starting the conference, or a phone link could be established after initial rendezvous has been made over the system with those participants who happen to be available.) A participant who receives such an invitation is asked by his program whether or not he wishes to join the conference. If the participant agrees, a message is returned to the chairperson carrying an "outline" of the relevant range of dates extracted from the participant's personal calendar; this outline indicates which time slots are free and which are not, without disclosing any additional information such as why the participant might be unavailable at a given time. The participant may alternatively decline to join the scheduling conference (which does not mean that he cannot come to the planned meeting, whenever it may be held), and a message to that effect is accordingly sent to the chairperson's program.

The real-time conference begins in earnest when the chairperson's program receives replies from all invited participants, or when the chairperson instructs his program to stop waiting for replies and commence the conference, which might be necessary if some replies are not received after a long wait. At this point, the calendar outlines received from the participants who joined the conference are "merged" to identify time slots for which all participants are available. The displays of the conference participants are initialized to show a part of this aggregate view in a *shared window*. (An example of a participant's display is shown in Figure 6-1.) The shared window can be manipulated by commands to "scroll" over

[27] The existing message-passing implementation used the operating system's "inter-process communication facility", which was found to be too slow for the large number of messages that are generated in a real-time conference. Performance was improved by using the system's facilities for "sharing" memory pages among files and process address spaces [88], and reimplementing Herlihy's message encoding and decoding algorithms using shared pages. Shared pages would have provided an ideal mechanism for simulating "multicast", but that was not attempted.

the range of dates, and by commands that select different combinations of participant calendars for merging (which is useful if there are scheduling conflicts that cannot otherwise be resolved). The effects of each such command are immediately displayed on every participant's screen.

```
+------------------------------------------------------------+
|RTCAL 3.2 use control-↑  12-4-82 11:52:07 Load=8.7    SARIN|
+------------------------------------------------------------+
|scheduling "thesis"  uncommitted (2hrs, 12-25 to 12-31-82)|
| With GREIF         LICKLIDER     SARIN         HAMMER      |
|      IN-Session    IN-Session    IN-Session    Absent      |
| session Running   chairperson: GREIF  controller: SARIN    |
+------------------------------------------------------------+
|LICKLIDER joined - all replies received                     |
+--------------------------------+---------------------------+
|Monday 27 December 1982         |Private calendar           |
|Merge of GREIF LICKLIDER SARIN  | Joe's birthday            |
| 9:00   XXX                     | 9:00                      |
| 9:30   XXX                     | 9:30                      |
|10:00                           |10:00                      |
|10:30                           |10:30                      |
|11:00                           |11:00                      |
|11:30                           |11:30                      |
|12:00                           |12:00                      |
|12:30   XXX                     |12:30    lunch             |
|13:00                           |13:00                      |
|13:30                           |13:30                      |
|14:00   XXX                     |14:00    Arpa meeting      |
|14:30   XXX                     |14:30     xx               |
|15:00   XXX                     |15:00                      |
+--------------------------------+---------------------------+
| COMMAND> propose 10:30_                                    |
+------------------------------------------------------------+
```

Figure 6-1: Example RTCAL Display

At any given time, only one of the conference participants is allowed to enter commands to manipulate the shared window; this participant, initially the chairperson, is said to be *in control*.[28] Special commands are provided to allow the participant currently in control to "give the floor" by passing control to another participant (or back to the chairperson). A "request

[28] RTCAL uses some terminology that is now obsolete. The "controller" in Figure 6-1 is a person, not a site or program as in Chapter 4, and what RTCAL calls a "session" we now call a "conference".

control" command is also provided; the system queues all participant requests, and a list of requests that have not yet been granted can be displayed on command. The chairperson is given special powers in that he can at any time *preempt* control from the participant who currently has it. These *conference control commands*, entered by typing a special control character, have no effect on the shared window; they only determine who has the ability to manipulate this window.

Each participant in a conference sees not only the shared window but also a *private window* that shows more detailed information, such as descriptions of or comments about this participant's appointments, that is truly private and is not visible to the other participants. The purpose of showing this private data is to aid decision-making; seeing details of his private appointments can help the participant decide whether or not he can agree to a proposed meeting time (see below). The private window is displayed alongside the shared window and is always kept current (with respect to the shared window) by automatically scrolling the private window whenever the shared window is updated to show a different date and time interval.

Decision-making in a conference is supported by allowing participants to *vote* on a *proposal*. The participant currently in control can propose a specific date and time for the meeting, and each participant enters a Yes or No vote depending on whether or not he finds the proposed meeting time agreeable. (A participant is not constrained to base his vote on whether or not his personal calendar shows him as being available.) The votes are tabulated and displayed to all participants as they arrive. A proposed meeting time can be *committed* by the participant currently in control (whether or not all participants voted Yes); the committed time is written into every participant's private calendar. A committed time can be "undone", and erased from the participants' calendars, to permit recovery from mistakes. The chairperson may terminate the conference at any time, whether or not a final meeting time has been committed.

A participant may leave a conference permanently at any time (using a control command). He may instead "escape" from the conference temporarily to perform some unrelated activity at his terminal (such as read some newly-arrived mail). When he "returns" to the conference (by resuming his conference program), his display is brought up to date to reflect the current state of the conference; he is also informed of any important events that may have occurred in his absence, namely votes on proposed times or the commitment of a final meeting time.

No facility was provided for allowing a participant to modify his personal calendar database while in a conference. Instead, the participant may "escape" from the conference and run the existing PCAL program to do this. If he updates his personal calendar in a way that affects the range of dates being considered for the meeting in question, an up-to-date version of his calendar outline will be automatically submitted to the conference when he returns. Thus, a participant might decide to free up some time for the meeting by canceling or rescheduling some less important entry in his personal calendar, and the effect of this will be made visible to the other conference participants when this participant's new calendar outline is merged with the others'.

6.1.2 Experience with RTCAL

Several trial real-time conferences were conducted using RTCAL, in groups of up to four users at a time. (Because all participants' "workstations" run on the same time-shared host, it was difficult to support larger conferences.) While we did not conduct a scientific study, the overall impression of our subjects was positive, that real-time conferencing was an interesting new mode of interaction. There were several suggestions for improvement such as more interesting and useful display and processing of calendar information. Many such features were planned can could have been easily added with little change to the structure of the implementation, but were not because it was more important to move on to other applications and develop the generic real-time conferencing ideas presented in this thesis.

By implementing RTCAL, we were able to take a first cut at implementation techniques, in particular replication of data, transfer and sharing of information between shared and private spaces, initiating a conference by "inviting" participants to "join", and response-gathering for negotiations. The techniques used have since been generalized, as described in Chapters 4 and 5.

The most important lesson learned from our experience with RTCAL is that designing the user interface to a real-time conferencing system is a difficult task. A participant's display screen presents many different kinds of information, including status information about the conference, shared and private application information, and command echo and feedback. On the 24x80-character alphanumeric displays that we used, there is no room for the "borders" between windows that we showed in Figure 6-1, making it very hard for a participant to sort out and understand the information. This would be less of a problem on

higher-resolution screens that would allow us to separate windows with borders, or if different windows could be shown in different colors. The display would still be somewhat cluttered, however, and it seems desirable to allow the users to suppress some of the displayed information and call it up only when they want it. It was also found especially important to notify participants (e.g., by ringing the terminal bell and displaying a message) of significant asynchronous changes that they might otherwise not notice, such as a participant leaving or joining or a passing of permission to enter commands.

6.2 Shared Bitmap System

The shared bitmap system that we have implemented is an extension of the concept of "virtual terminal" that allows multiple workstations to share the same virtual terminal. It provides input and display functions not available in most traditional virtual terminal protocols, namely a bitmapped rather than an alphanumeric virtual screen, and pointing device or "mouse" input. It has the disadvantages of virtual terminals discussed in Section 1.6.3, especially the inability to transfer application data. For this prototype, however, we decided that speed of implementation was more important than demonstrating all of the possible conferencing functions described. The general-purpose shared bitmap facility allows easier development of a centralized rather than a distributed application program, and can be reused for other prototype applications in the future.

Our choice of a shared bitmap was also influenced by the availability of an experimental protocol named *Blink*, developed by David Reed for coordinating a copy of a bitmap between a single workstation and a remote host. Blink uses a "non-sequential" protocol of the kind described in Section 4.6.4, and we retained the basic structure of the protocol while making several extensions in order to support multiple workstations holding copies of the same bitmap. The resulting protocol, and the shared bitmap system itself, we have named *MBlink*. Because non-sequential protocols are not commonly found in practice (except in specialized applications such as voice), MBlink is useful in providing a practical manifestation of the idea.

MBlink will support any application that uses a bitmap for display and processes input from one or more mice. The architecture used is shown in Figure 6-2. The application program generates updates to the shared bitmap, which the MBlink manager sends to the workstations for display. Each workstation reports the position of its mouse and the status

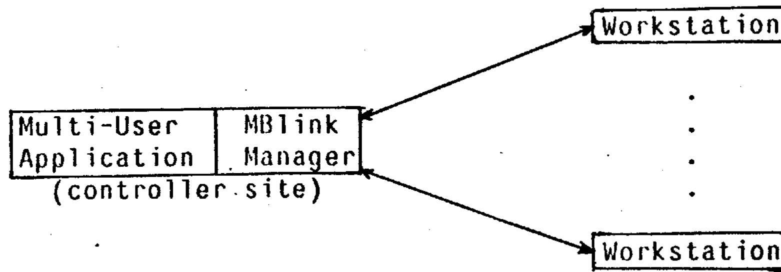

Figure 6-2: Architecture of Shared Bitmap System

(up or down) of the mouse buttons to the manager, which reports it to the application for processing. The MBlink manager displays a "pointer" pattern on the bitmap that follows each workstation mouse as it moves. The interface between application and manager also allows for adding and removing workstations; details are discussed presently.

MBlink, like its predecessor Blink, is implemented using DEC VAX™ machines as controller sites and Xerox Alto personal computers [113] as workstations. The machines are connected by 10Mb/s Proteon Pronet™ ring network and an 3Mb/s Xerox experimental Ethernet. The underlying communication protocol is the "User Datagram Protocol" extension [96] to the Darpa Internet Protocol [98].

6.2.1 The Blink Protocol

In order to provide a context for understanding MBlink, we first describe Reed's original Blink protocol for communication between a remote server and a single workstation. This description is adapted from internal Blink documentation written by David Reed, and from the actual code.[29]

Blink allows an application program on a host machine to treat a block of memory as a directly manipulable screen bitmap, whereas in actual fact the screen is on a workstation machine connected to the host via a network or internet; Blink provides the mechanism for ensuring that the workstation's screen is consistent with the host's bitmap. Blink also allows the position of the workstation's mouse, and the state of the buttons on the mouse, to be seen

[29] A newer version of the Blink manager code that uses asynchronous "tasks" for the manager and the application was implemented by Michael Greenwald for Berkeley Unix(TM) version 4.1, but was not used for our implementation of MBlink because the multi-tasking extension to CLU has not been ported to Berkeley Unix version 4.2.

by the application program at the host.

The screen bitmap (at both the workstation and the host) is treated as a collection of *ublocks* ("unit blocks"), each of size 32 by 32 pixels. This size was chosen by Reed to allow as many as four ublock updates, sufficient for most small changes to the bitmap, to be transmitted in a single internet user datagram. The choice of ublock size also implies that a complete array of ublock version numbers, described below, from an Alto screen of size 640x800 pixels, will occupy 20x25 = 500 bytes and will also fit into a single datagram.

Each ublock of the bitmap has an 8-bit *version number* that is incremented by one (modulo 256) whenever the application modifies the contents of one or more pixels in the ublock.[30] It is possible for a given ublock to be updated repeatedly and to thus cycle through the finite version number space; if so, delayed duplicate packets still lurking in the network may be mistaken for current ones. For an assumed "maximum packet lifetime" of one minute, this will not happen if each individual ublock is updated no more frequently than every half second. (This maximum frequency is averaged over a minute; a ublock may be updated more frequently in bursts.) No special care is taken to protect against more frequent wrap-around of a ublock's version number space; it is simply assumed, with extremely high probability of being correct, that this will not happen.

All bitmap updates are sent by the manager to the workstation in *absolute* form: a ublock is identified by its X and Y coordinates, and a new version number and the new contents of all 32x32 = 1024 bits are included. Since such updates do not depend on previous ones, the protocol is robust against the loss of old updates to a ublock because a new update supplies the entire new contents. (If a given ublock update is lost and the same ublock is not subsequently updated, the update can be retransmitted as described below.) The protocol is also robust against packet duplication and reordering because the workstation will not install a received ublock update unless the version number it carries is greater (modulo 256) than the corresponding version number in the workstation's copy of the bitmap. Thus, updates are sent as unreliable datagrams without the overhead of a "virtual circuit" or TCP connection.

---

[30] A set of subroutines, such as "RasterOp" or "BitBlt" and writing characters from fonts, are provided to the application for modifying the bitmap. This set can be arbitrarily extended, e.g., to draw arcs, curves, and polygons, with no change to the Blink protocol or to the manager or workstation software. The only requirement is that each subroutine correctly report to the Blink manager which ublocks it modified.

To bring the workstation's bitmap up-to-date in the face of lost update packets, the manager periodically sends the workstation a *poll* message requesting the workstation to return a packet carrying the version number of every ublock in the workstation's bitmap copy. (The workstation's response also carries its mouse coordinates and button positions, which are made available to the application.) Each poll message from the manager carries a *timestamp*, drawn from a 16-bit one-millisecond clock (which wraps around every 65 seconds, greater than the expected maximum packet lifetime). The timestamp is repeated in the response sent by the workstation, and the manager ignores a response that is not "current", i.e., does not carry the timestamp of the last poll message sent. This ensures that the manager does not perform unnecessary retransmissions (below) on receiving an obsolete response that may have been delayed in the network.

When a current response is received from the workstation, the manager compares the ublock version numbers in the response with the actual version numbers in the manager's copy. The manager retransmits updates (coordinates and version number and entire contents) for each ublock for which the workstation's version number does not match. After processing the workstation's response and performing necessary retransmissions, the manager also sends updates for ublocks that have been modified by the application, followed by a new poll message. If no current poll response is received from the workstation within some conservative estimate of the round-trip delay, the manager sends new updates only and a new poll message.

The "shape" of the workstation's hardware cursor, which follows the movements of the mouse, is a programmable 16x16-pixel pattern. The application can modify the cursor pattern, and changes to the cursor pattern are transmitted to the workstation in the same way as with ublock updates, i.e., complete new contents together with a version number. A cursor update is retransmitted if the cursor version number returned by the workstation in its poll response is out of date.

The manager also maintains a "flag" indicating whether or not the workstation's entire copy of the bitmap is known to be up-to-date. This flag is reset whenever the bitmap is updated and is set whenever all ublock version numbers in a received response match the manager's ublock version numbers. If the application wishes to ensure that the workstation's screen displays the current state of the bitmap, it can halt further updates to the bitmap until the flag is set.

A Blink "connection" between a given host and workstation is initiated by the manager sending poll messages to the workstation until a response is received, at which point a bitmap is created with the size and initial ublock version numbers specified in the workstation's response.

6.2.2 Handling Multiple Workstations

Our implementation of MBlink is a compatible extension of the above Blink implementation, with the following properties:

- No change to the workstation software is needed in order to support MBlink; each workstation in a conference interacts with the MBlink manager using the original Blink protocol and is unaware that there may be other workstations at which the same bitmap is replicated. (The MBlink manager is of course aware of the multiple workstations, and a user at a workstation may also be thus aware.) This was done partly to save programming time, the workstation software having been written in a different language and operating system than the Blink manager. In addition, it provided a useful demonstration of how much can be implemented with minimal change to existing software. Even the "echoing" of mouse positions on all workstation screens can be implemented without the workstation software being aware of it, albeit using a rather convoluted method described below. Functions that could not be implemented within this constraint are described in Section 6.3; for example, Blink does not support keyboard input from the workstation and neither does MBlink.

- The same set of functions available to the application in the original Blink application are still available in MBlink. MBlink only adds new functions without changing the semantics (including the procedure call and return interface) of any existing ones. Thus, an existing Blink application that interacts with a single workstation can be run without any change if MBlink is used instead. (It is only necessary to relink the object code modules.) The output from an existing Blink application can also be displayed on multiple workstations with minimal change to the application software by including a few commands for adding and possibly removing workstations.

In order to describe how the above compatibility was achieved, we briefly examine the application interface to the existing Blink implementation. The procedures that are relevant here are:

remotescreen$sync(rs:remotescreen):
Any changes made to the bitmap by the application are sent to the workstation, and polling and retransmission is performed until the workstation's copy is known to be up-to-date.

remotescreen$get-mouse(rs:remotescreen) returns(int,int,buttons):
The workstation is polled until it returns a response; the x- and y-positions of the workstation mouse and the state of the mouse buttons are extracted from the response and returned to the application.

To provide a reasonable semantics for the above in MBlink, when there may be multiple workstations showing the bitmap and sending mouse input, we introduced the concept of a *primary* workstation. The primary workstation is the first one to reply when an MBlink connection is initiated. (While an existing Blink application will specify only one workstation address when initiating the connection, MBlink provides an additional procedure that allows the application to supply more than one workstation address.) Subsequent calls by the application to the above procedures apply to the primary workstation only, i.e., **remotescreen$sync** returns when the primary workstation is up-to-date regardless of the state of other workstations, while **remotescreen$get-mouse** returns the coordinates and buttons of the primary workstation's mouse. The application can designate a different primary workstation at any time, using the procedure **remotescreen$set-primary-ws**.

The notion of primary workstation was introduced solely for compatibility reasons, to support existing Blink applications. An MBlink application does not have to use the above procedures, and can synchronize with and read mouse input from all workstations using the following new procedures:

remotescreen$maybe-sync(rs:remotescreen,seconds:int):
Returns when all workstations are known to have up-to-date copies of the bitmap, or when the given timeout period expires, whichever happens first. When this procedure returns, the application may examine the state of **rs** to determine which workstations, if any, are behind, and may take any action it wishes such as removing these workstations from the conference.

remotescreen$read-ws-mouse(rs:remotescreen,ws:address)
returns(int,int,buttons):
Returns the mouse report last received from the given workstation. If the application wishes to read the mouse position and buttons for every workstation, it must call **remotescreen$read-ws-mouse** repeatedly. This procedure does *not* poll the workstation and wait for a response because of the delay that would be incurred if invoked repeatedly for each different workstation. If the application wishes to read current mouse

positions rather than the last position received (which might be quite old), it first calls `remotescreen$maybe-sync` once to request a response from all workstations.

The *retransmission* procedure for MBlink differs somewhat from Blink because of the presence of multiple workstations. "New" ublock updates caused by the application are periodically sent to all workstations. (A separate, but identical, packet is needed for each workstation, because broadcast and multicast are not currently supported by the IP datagram protocol.) When a given workstation's response arrives, however, ublocks for which the workstation is behind are retransmitted only to that workstation. Unlike Blink, the MBlink manager does not send a new poll message immediately after processing a workstation's response, because other workstations may not yet have responded to the previous poll message. A new poll message is therefore sent only after all workstations respond, or after a suitable time interval (one second in the current implementation).

6.2.3 Remote Mouse-Tracking

The position of each workstation's mouse can be shown on all workstation's screens, without the application or the workstation software being aware of it, as follows. Whenever the MBlink manager has control and is about to send a new set of updates to the workstations, it first superimposes each workstation's pointer shape on the bitmap at the most recent mouse position received from the workstation, using an exclusive-or (XOR) operation. Ublock updates sent to the workstations will therefore show the pointers at the appropriate positions on the bitmap. Whenever the MBlink manager returns control to the application, it repeats the XOR operation to restore the modified pixels of the bitmap to their original state. This works because XOR is a reversible operation that is its own inverse, and is necessary because the application may be using the contents of the bitmap, e.g., in a "BitBlt" operation, and must see the correct bitmap contents without the pointers superimposed.

A user at a workstation will see not only the mouse pointers "echoed" by the manager, but also a locally-tracked hardware cursor. When the user moves his mouse, the pointer echoed by the manager will lag behind the local cursor somewhat, as shown in Figure 6-3 for participants 1 and 2. The echoed pointer will ultimately catch up with the local cursor if and when the participant stops moving his mouse; the two patterns will then overlap, as shown for participant 3. Seeing both a locally tracked and a remotely echoed cursor does have its uses,

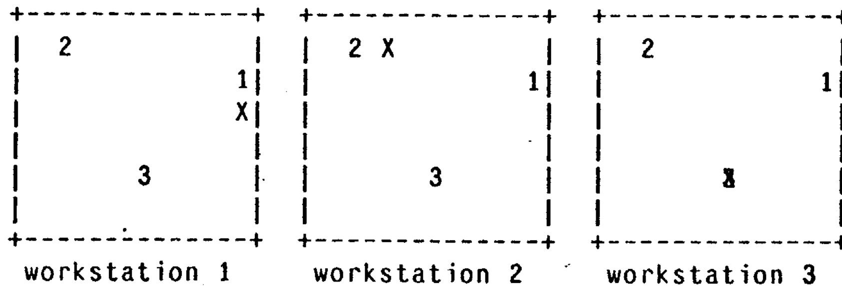

Figure 6-3: Tracking of Workstation Pointers

in that a participant can roughly judge the response time of the system and can tell approximately when the other participants can see his updated mouse position (at which point the user may wish to say something over the voice channel in reference to the information displayed in the region of the bitmap where he is pointing). On the other hand, when the locally and remotely tracked cursors do coincide, the pattern seen by the user will be somewhat fuzzy because of the superimposition. Given the constraint of no change to the existing workstation software, there is no way to avoid this problem; it can be resolved only by modifying the workstation software to do more local processing as described below in Section 6.3.

6.2.4 Experience with MBlink

At the time of completing this thesis, the only application that we have run using MBlink is a network simulator written by Lixia Zhang for experimenting with congestion control algorithms. This application was originally written by Zhang using Blink to display the status of the network being simulated on a workstation screen, and this author was able to convert the application to use MBlink by modifying just a few lines of code to allow workstations to be added and removed. Thus, a group of network researchers can hold a "design meeting" in which they initiate and observe the simulation of one or more congestion control algorithms using different network topologies and parameters. A user can also work alone from a single workstation using the same programming, with no change in interface from the original program that ran with Blink instead of MBlink.

6.3 Extended Bitmap System

The implementation of MBlink described above is adequate for some purposes, but it has some limitations that arise from our unwillingness to modify the existing workstation software used for Blink:

- Because the workstation in Blink does not send keyboard input to the manager, the application cannot process keyboard input from any of the workstations. (The application may read keyboard input from the one terminal under which it is logged in, which may or may not be one of the workstations in the conference, and which cannot be changed even if MBlink's "primary" workstation is changed.)

- Response to mouse movements and button commands is somewhat sluggish (about a second) because a workstation sends mouse information only when "polled" by the manager. This could be improved by having the workstation unilaterally send its mouse state to the manager at some reasonable frequency. (Care must be taken in this case not to overload the manager or the network and cause response to degrade instead.)

- The application cannot send "participant-specific" feedback that it wishes to display on one workstation screen only and not the others. (At best, it may use the following convoluted method: Remove all workstations except one, update the bitmap to display the necessary feedback, wait until synchronization is achieved and the one workstation displays the up-to-date bitmap, restore the bitmap to its original state, and add back the workstations that were previously removed.)

- The bitmap size is determined solely by the first response that the MBlink manager receives; there is no possibility of negotiating a suitable bitmap size among a group of workstations. (This is currently not a problem because the only workstations supporting the protocol are Altos, which all have the same screen size.)

- The method described above for tracking workstation mice may be a clever programming trick, but it is inflexible and inefficient. It imposes considerable overhead on the MBlink manager, which must repeatedly superimpose the pointer patterns and then turn them off again before letting the application manipulate the bitmap. The communication bandwidth used is also greater than it would be if the manager simply sent all mouse coordinates to the workstations and let the workstation software locally perform the superimposition of pointer patterns. There is also the problem described above of a

workstation's local and remotely-tracked cursors overlapping. If the superimposition were done locally, it would be very easy for the workstation software to suppress the remotely tracked cursor whenever it overlaps the local cursor, or even to disable the workstation's remotely tracked cursor (while still showing the other workstation's cursors) if the user finds it distracting to see two cursors following his mouse.

This section presents a design for an extended version of MBlink, which we shall call *XMBlink*, that overcomes the above limitations at the cost of having to modify the existing workstation software. The design is presented below in the form that we originally intended to implement it but were unable to for lack of time.

6.3.1 XMBlink Functions

XMBlink supports "conference objects" of the type XMBLINK-CONF shown in Figure 6-4. The component objects are described below.

- The **height** and **width** of the shared bitmap are specified as the number of "ublocks" in the vertical and horizontal directions, respectively. (The actual height and width of the bitmap in pixels are therefore **height*M** and **width*N**, respectively.) The height and width are included in the conference "description" that is transmitted initially to workstations and lookup servers as described in Section 5.1.2. Prior to initialization of the bitmap (below), the height and width are both zero.

- The component **ptr-frequency** specifies the maximum number of times per second a workstation should send a pointer position report. This is included in the conference description.

- The **title** is a descriptive text string supplied by the application. This is included in the conference "description"; aside from the parameters listed above, the title is the only information released outside the conference to prospective participants. Information about scheduled times, user names, access rights, and so on are implemented by the application in whatever way it chooses. (Such information typically does not belong at the level of a virtual terminal protocol. The crossword application at the end of this chapter provides an example of how these functions may be implemented.)

- The addresses and current status of the workstations in the conference are specified by the arrays **ws-address**, **ws-invited**, **ws-requested**, and **ws-active**, each of which is

```
type UBLOCK = array[1..M,1..N] of (on,off)
                                        M = N = 32 in the implementation

type XMBLINK-CONF = record

    conference description
    controller: site-address
    title: string
    height: int                         number of ublocks
    width: int
    ptr-frequency: int                  per second

    workstation information
    nworks: int
    ws-addresses: array[1..nworks] of site-address
    ws-invited: array[1..nworks] of
            (Invited,Removed,Undefined)
                                        invited by application?
    ws-requested: array[1..nworks] of
            (Requested,Declined,Undefined)
                                        workstation requested to join?
    ws-active: array[1..nworks] of (yes,no)
                                        active or not

    information replicated at active workstations only
    bitmap: array[1..height,1..width] of ublock
    ypositions: array[1..nworks] of int
    xpositions: array[1..nworks] of int
    ptr-shapes: array[1..nworks] of ublock
    own-ptr-shape: ublock
    messages: array[1..nworks] of string

    information from workstations,
    retained by controller only
    streams: array[1..nworks] of queue of record
            event: key-or-button
            mousey: int
            mousex: int
            timestamp: int
    ws-heights: array[1..nworks] of int
    ws-widths: array[1..nworks] of int

    continued
```

Figure 6-4:XMBlink Conference Object Specification

```
version numbers
description-v#: int
ws-status-v#: array[1..nworks] of int
ws-requested-v#: array[1..nworks] of int
message-v#: array[1..nworks] of int
own-pointer-v#: int
ublock-v#: array[1..height,1..width] of int
ptr-timestamps: array[1..nworks] of int
poll-timestamp: int            last poll message sent

controller bookkeeping information
ws-description-ackd: array[1..nworks] of (yes,no)
ws-message-ackd: array[1..nworks] of (yes,no)
ws-pointer-ackd: array[1..nworks] of (yes,no)
ws-bitmap-ackd: array[1..nworks] of (yes,no)
ws-response-timestamp: array[1..nworks] of int
                               timestamp of last poll
                               to which workstation responded
ublock-dirty: array[1..height,1..width] of (yes,no)
                               ublock has changed
ublock-recent: array[1..height,1..width] of (yes,no)
                               ublock has been transmitted
                               since last poll
```

Figure 6-4, continued

of size **nworks**. The arrays include all workstations with whom any communication about the conference occurred, whether or not they currently are or ever were active; a given workstation therefore has a fixed "index" into these arrays for the duration of a conference.

- The **bitmap** itself is conceptually viewed as a rectangular array of the specified number of ublocks. It is, however, implemented in the usual way as a single monolithic bit-array in "row-major" order. The bitmap is replicated at "active" workstations only.
- Each element of **ypositions** and **xpositions** is the latest received pointer position report, in pixel coordinates relative to the top left corner of the bitmap, from a given active workstation. (The values are undefined, and the pointers not displayed, for workstations that are not active, or whose mouse positions lie outside the shared bitmap.)
- Each element of **streams** is a queue of "input events" from a given active workstation. Each input event is either a keystroke or a mouse button, and carries the position of the mouse (in pixel coordinates) at the time the key or button was pressed by the participant.
- Each element of **ptr-shapes** is the bit-pattern to be used for displaying the corresponding workstation's pointer.
- The pattern **own-ptr-shape** is the bit-pattern that each workstation should use, instead of the pattern seen by the others, for displaying its own pointer.
- Each element of **messages** is a text string that the corresponding workstation should display to its participant in a private area of the physical screen. This is used by the application to show participant-specific error reports or text messages, which cannot be displayed in the shared bitmap because the latter is seen by all participants.
- The arrays **ws-heights** and **ws-widths** specify what height and width each workstation is willing to allocate for displaying the shared bitmap. The application will usually (but is not forced to) take this into account when selecting a bitmap size.

The remaining components of XMBLINK-CONF are version numbers for the above, and bookkeeping information that the controller maintains about responses from the workstations. We next describe how the application and manager interact in a conference.

6.3.2 Application Program Interface

The application program and XMBlink manager run as asynchronous (but not concurrent) "tasks" in a single address space. Thus, the application can at any time examine the components of the conference object created by the manager. The application may modify some of the components of the conference object, using the subroutines described below; other components are updated only by the manager. The application and manager pass control to each other by means of *signals* that request specific actions (when the application signals the manager) or report specific events (when the manager signals the application). This passing of control is synchronous and symmetric (as with "coroutines") in that the task issuing the signal is suspended until the other task issues a signal in return.

Manager Signals from Application:

```
Invite-Workstation(conf,workstation#)
Remove-Workstation(conf,workstation#)
Accept-Workstation(conf,workstation#)
Enable-Input(conf,workstation#)
Disable-Input(conf,workstation#)
Send-Message(conf,workstation#,string)
Terminate-Conference(conf)
```

Application Signals from Manager:

```
Workstation-Requests-Join(conf,workstation#)
Workstation-Declined(conf,workstation#)
Workstation-Size-Changed(conf,workstation#)
Workstation-Input-Received(conf,workstation#)
```

Figure 6-5:MBlink Signals

The signals between the application and manager are listed in Figure 6-5. In addition, the special signal *Resume* can be issued by either task when it wishes to yield control without requesting a specific service or reporting a specific event. This is done when a task has nothing further to do for the moment, or when it has a lengthy computation to perform which it breaks up into shorter steps so as to give the other task a chance. (It is especially important that the manager be allowed to run frequently, so that it can remain responsive to incoming messages.) Each task on yielding control maintains its own state information describing unfinished work that is to be performed later. A Resume signal carries an argument which is the approximate amount of time the other task should wait, if it has no other signals to issue, before signalling a Resume in return; this forestalls a useless exchange of Resume signals

when there is nothing going on in either task.

The application obtains an object of type XMBLINK-CONF, i.e., a conference, by calling the subroutine Create-Conf with the desired conference title as argument. (This is a simple procedure call, not a task signal.) The conference object returned has no workstations and has an empty bitmap. The subroutine Add-Workstation registers a new workstation address and returns its "index" into the workstation arrays; no actual communication with the workstation occurs until a signal (e.g., Invite-Workstation or Accept-Workstation, below) is issued. The subroutine Set-Bitmap-Size, which can be called just once for a given conference, sets the size of the shared bitmap and initializes its contents to all zeroes.[31] (Active workstations will be informed of the bitmap size when the manager next receives control.) Subroutines are provided to allow the application to modify the bitmap contents, by writing text characters from a font, or performing Boolean operations on the contents of rectangular regions ("RasterOp" [90] or "BitBlt" [58]), or drawing lines, and so on. Each such subroutine marks any "ublock" that it modifies as being "dirty", so that the manager will send update messages when it next received control (as described in the next section).

A typical use of the signals provided, but not the only possible use, is the following. We assume that the application has a list of addresses of the workstations it wishes to include in the conference, obtained by some unspecified outside means, and that it wishes to choose a bitmap size that all workstations can accommodate.

- The application calls Create-Conf and Add-Workstation to include all the desired workstations' addresses in the conference. It then issues an Invite-Workstation signal for each workstation, which causes the manager to send a conference "description" to each.

- Whenever the application receives control (a Workstation-Requested or Workstation-Declined signal, or a simple Resume after a timeout), it checks the array **ws-requested** to see if all workstations have requested to join or have declined. If there are no more workstations from whom a reply is still awaited, or if the application decides it has waited long enough, it selects a suitable size and calls Set-Bitmap-Size. If the application wishes to continue waiting, it Resumes the manager, with an argument equal to the amount of additional time the application is willing to wait if nothing new happens.

---

[31] The restriction on not allowing the bitmap size to change avoids problems of reallocating storage and moving bits around; there is no conceptual reason why the bitmap size cannot change in a conference.

- Having set the bitmap size, the application issues an Accept-Workstation signal for those workstations with which it wishes to share the bitmap. (The application will typically accept all workstations that wish to join, but it can use other arbitrary access criteria.) The manager informs the accepted workstations thereof. Each workstation on being informed allocates and initializes a bit-array for the shared bitmap and displays it to the participant; it also begins sending mouse position reports at the specified frequency. (The frequency is initialized to some standard value; the manager automatically reduces the frequency as the number of active workstations grows large.) The application may also issue an Enable-Input signal for one or more workstations; unless it wishes to accept input from only one participant at a time, the application will typically enable input for every active workstation.

- Keystrokes and mouse button commands arriving from active workstations cause the manager to issue Input-Received signals to the application. The application processes input according to its own syntax and semantics; it may even let input received from a workstation remain on its queue (the corresponding element of `streams`) and process it later. The manager issues an Input-Received signal for a given workstation only if there is no input already in the workstation's queue when new input arrives.

- The application, in response to input or at any time when it has control, may update the shared bitmap to display some information. When it has completed a batch of changes and wishes to have them sent to the manager, the application Resumes the manager.

- The manager on being Resumed sends update messages for all "dirty" ublocks to all active workstations, as described below.

- If the application issues an Accept-Workstation signal later in the conference, the manager sends the new workstation an update message for every ublock in the bitmap.

- If the application wishes to be certain that all workstations (at a "synchronization point"), or a particular one, or some subset, have received all updates so far, it refrains from updating the bitmap until the desired condition is met. The condition is tested by examining the fields of the array `ws-bitmap-ackd` for the workstation(s) of interest. (The fields are automatically maintained by the manager as described below.) The application may change its mind and stop waiting after a timeout interval, just as at the beginning of the conference.

6.3.3 Transport Protocol

XMBlink is uses a "non-sequential protocol" that is a modification and extension of that used in Blink. Components of the conference object therefore have associated *version numbers* which are transmitted in all update messages. These version numbers, together with acknowledgment information that the manager collects from the workstations, have been included in Figure 6-4 for completeness.

The conference *description* consists of the `controller` address, conference `title`, bitmap `height` and `width`, and `ptr-frequency`. The description is sent by the manager whenever "inviting" a workstation. An update to any of the components in the description causes the `description-v#` to be incremented, and a description update sent to all active and invited workstations.

A workstation being invited is also sent its corresponding elements of the workstation "status" arrays, together with a version number that is incremented when any of these change for that workstation (by virtue of the workstation being invited, accepted, or removed by the application). A workstation receiving the conference description or its own status, or both, returns an acknowledgment carrying the version numbers on its copy of the description and of its own status. The manager keeps track of whether the workstation has acknowledged the most recent version of each, and retransmits either or both until acknowledgment is received. (On removal of a workstation, the new status of that workstation is sent only once; it is not retransmitted unless the workstation attempts to communicate further.) Workstations are not informed of other workstations' status, to reduce communication traffic and manager bookkeeping overhead.[32] The application is expected to display status information about all workstations if it needs to in the shared bitmap. (The application may display the status information maintained by the manager, but will also usually include higher-level information such as user names, maintained by the application in "parallel" workstation arrays.)

The "message" object for a given workstation also carries a version number that is incremented when the application wishes to send a new message. As with the above, the manager keeps track of which message version number the workstation has acknowledged,

---

[32] If multicast were available, it would be reasonable and efficient to send every workstations' status to all of them at once.

and retransmits the message until acknowledged by the workstation or replaced by a newer message from the application. (If the application wishes to ensure that the workstation sees every message, it can refrain from issuing a new message until it determines from the `ws-message-ackd` array that the previous message has been acknowledged.) Updates to, and acknowledgments of, `own-pointer-pattern` are handled similarly.

Updates to "ublocks" in the bitmap are handled in essentially the same way as in Blink and MBlink. The bitmap size in XMBlink is selected by the application, perhaps after a "negotiation" in that considers the workstations' screen size, rather than being automatically determined by the first workstation response received by the manager. Version number "zero" is never used by the manager, but may be returned in a response by a workstation that has no knowledge of a ublock's contents; this will always trigger a retransmission of the ublock regardless of its current version number.

Mouse input in XMBlink is transmitted unilaterally by the workstations, rather than only in response to a poll, to speed up response. Every pointer position report from a workstation is "timestamped" to ensure that the manager retains, and forwards to the other workstations, only the most recent one that it receives. These position reports are never retransmitted because new ones are generated by the workstation at the specified `ptr-frequency`. To save communication bandwidth, the manager does not report a workstation's mouse position to all other workstations immediately on receipt of a report. Instead, the manager periodically (at about the specified `ptr-frequency`) sends every workstation's most recent mouse position in a single packet.

"Stream" input, i.e., keystrokes and button clicks, is inherently sequential, and transmitting it using a non-sequential protocol requires some extra work. Each input event from a workstation carries a timestamp, and the manager returns acknowledgments to the workstation specifying the timestamp of the last received input event. The workstation keeps a buffer of the stream input events not yet acknowledged, up to some maximum number K. Whenever the workstation has new input to send, it sends not just the new input event but every unacknowledged one. Thus, if the manager did not receive the previous packet, the missing input event will appear in the new packet. (The manager of course discards duplicate events with timestamp less than or equal to the most recent one already received.) If the workstation has no new input events to generate and the manager has not acknowledged the most recent one, the workstation retransmits the input event buffer as often as necessary.

The manager will not receive some input if every one of K successive input packets from a workstation is lost; special action may be taken to protect against this rare possibility. That is, if a workstation finds that it has new stream input to send and the manager has not yet acknowledged any of the last K events, the new input is saved locally until the manager returns an acknowledgment. (This is similar to the "window" flow control strategy of virtual circuits, except that the window size is now fixed.) The participant is informed by his workstation (in the private messages area) that his input is not being processed (which he might also notice when the application fails to respond), and may choose to leave the conference if input remains blocked for a long time. The participant may instead instruct his workstation to discard all buffered input and only send newly-entered input. There is no guarantee that the manager will get the new input quickly, nor is it known how much of the previous input the manager did receive because it may have been the manager's acknowledgment, not the workstation's input message, that was lost.

Because each input event also carries the position of the mouse at the time of the keystroke or button click, some optimizations are possible. The timestamps on input events and on mouse position reports are drawn from the same clock, allowing the manager to use the mouse position from an input event if the event's timestamp is later than the most recently received position report. This also means that the workstation need not generate a separate mouse position report if there is a new stream input event within the time interval specified by the reciprocal of `ptr-frequency`. If there is no new stream input and the time to send a new mouse position report arrives, the workstation includes all unacknowledged input events in the same packet as the position report.

6.3.4 Sample Application

We have designed a first application for XMBlink, a game in which a group of users can cooperate in solving a crossword puzzle. We chose a game because it has a predefined and well-understood objective and allows users to try out the system with a minimum of training overhead. (Our experience with RTCAL was that contrived problem situations had to be constructed in order to test the system, and conferences tended to be relatively inactive because users had to learn commands and often received instruction over the phone.) Note that the crossword game as we describe it is not a competitive game, but a cooperative one in the same vein as the real-time conferences addressed in this thesis.

```
1  ALTO-504         SMITH      253-5845  Req#1
2  ALTO-220         -- WAITING --
4  18.10.0.79       WESSON     555-1212
5  ALTO-509         JONES      --        Reserved
6  <enter new address here>
```

```
+1-+2-+--+--+3-+--+--+4-+     "Animal House" by Sunil Sarin
| C| A| T|##| F| R| O| G|
+--+--+--+5-+--+--+--+--+     19-Apr-1984 11:45am
|##| C|##|  | I|##|##|  |     played 11m15s
+6-+--+7-+--+8-+9-+10+--+     12/18chars 4/8words 0peeks
| A| T| E|##| G| O| A| T|
+11+--+--+--+--+12+--+--+
|  |  |  |  |##| I| F|##|
+13+--+--+--+14+--+--+--+
| O| R| E|##|  | L| T|  |
+--+--+--+--+--+--+--+--+
```

```
        ACROSS                   DOWN

   1. Grown kitten         * 2. Pretender
   3. Grown tadpole          3. "... Leaf", Eden
 * 5. Hello                     clothing
   6. Finished eating      * 4. Obtained, colloq.
   8. Grown kid            * 6. "Much ... about
 *11. Not across                 nothing"
  12. Suppose              * 7. Grown female lamb
  13. Mine product           9. Black gold
 *14. Vocal part            10. Rear
```

Figure 6-6: Example Crossword Display

The crossword application uses the shared bitmap provided by XMBlink as shown in Figure 6-6 (which is only a crude approximation of what would be shown on a bitmapped screen; the crossword itself would also be larger and more interesting). The top region of the bitmap shows information, in tabular form, about each participant:

- The participant's workstation-address (in symbolic form if it appears in a host name table).
- Name and phone number. These are entered manually by the participant himself after joining, by selecting the region of the bitmap using his mouse and then typing in the information. This information is purely for human use (e.g., to establish a phone connection if it has not already been made); the crossword system does not force any participant to enter this information, and does not use it in any way except to display it.
- Which participant, if any, has the crossword board "reserved", and the ranks of requests, if any, in the queue. Requests are made by pointing at the region where this is displayed and clicking a mouse button; releasing the reservation or withdrawing a request is done similarly. The reservation is granted automatically to the first request on the queue whenever no "words" are reserved (below).
- The flag "Waiting" indicates a workstation that was "invited" but from which no response has yet been received.
- The numbers on the left hand side are indexes into the workstation array. Workstations of participants who left the conference, or declined to join, are not shown (e.g., number 3 in the figure).

An extra line (number 6 in the figure) is always included at the bottom of the table, which any participant can select by pointing and then type in the address of a new workstation to invite. (Only one participant can do this at a time; an attempt to select this line while another participant is using it will fail.)

The middle region of the bitmap displays a crossword *board*, with some of its cells filled with characters typed by the users. Alongside the board is shown a caption for the crossword and summary information about the current state of the game: current and elapsed time, and fraction of cells and words filled in.

Each active participant has a *cursor* (not shown in the figure) which is located in some cell and is oriented in some "direction", across or down. A character typed by the participant will be inserted into the current cell of his cursor, and the cursor advanced to the next cell in its given direction. The cursor will not advance past the end of a word in this way, but can be moved over the board using commands to move to the next or previous word, or to move to a specified "clue" number, or to move to the cell in which the participant's mouse pointer is

currently located.

A character cannot be inserted into a cell that has already been filled, unless the cell is first *erased* by an explicit command. This provides some protection against accidental interference of concurrent commands. (It also closely parallels what happens when solving a crossword on paper.) A different kind of protection is available, by a participant "reserving" the current word of the cursor, i.e., the sequence of cells that includes the current cell in the current direction of the cursor. While it is not necessary for a participant to hold a reservation in order to modify (insert or erase) a given cell, holding a reservation on a word prevents other participants from modifying any cell in the word. If no participant has any word reserved, a participant may reserve the entire board, in which case only that participant can modify the board. Timeouts are set on all reservations, similar to those described for the joint document editor in Chapter 2.

The bottom region of the bitmap shows the *clues* for the given crosswords. Every clue for which not all of the corresponding cells on the board have been filled is flagged with an asterisk, to help the participants easily locate the clues that need work. (The absence of a flag does not mean that the cells were filled correctly; this can only be checked using the "peek" and "score" commands below.) The flags are on for all clues at the start of the game, and are set and reset automatically by the system as characters are inserted and erased, an amenity not available when solving a crossword on paper. Other amenities not available with paper crossword puzzles are a *Peek* command that causes the system to insert the correct solution character in a given cell, perhaps allowing the participants to continue working if they are stuck, and a *Score* command issued at the end of a game for checking the participants' entries and filling in the solution for the entire board.

Chapter Seven

Summary and Conclusion

This thesis has addressed the problem of supporting a "real-time conference" in which a group of users may view and manipulate a "shared space" of problem information from their own workstations, possibly conducting a voice conversation at the same time. Starting with this concept, we have examined in detail the issues that must be addressed in both designing and implementing such conferences. Our objective has been to uncover the options available and to determine their effect on the user interface and on performance, so that a system designer may make reasonable choices in developing a real-time conferencing system for his particular application and user community. While our intention has been to develop concepts and principles that are as generally applicable as possible, we used a particular example application and developed it in detail in order to illustrate and justify the general concepts and principles.

The problems we addressed in designing real-time conferences include the specification of shared and private "spaces", methods for controlling concurrent participant actions, and overall conference management issues such as initiating and terminating a conference and participants joining and leaving. Designing an appropriate user interface that includes all of these functions is not a trivial task. There are many more "degrees of freedom" available to the designer than when designing a single-user interface, e.g., whether and how much concurrent activity to allow, or which activities should be private and not visible to the others versus which should be shared. Consequently, there is also a greater danger of a user getting confused and making errors, e.g., inadvertently performing an action that all other participants see when in fact he meant to do it privately; such errors may have serious social consequences that do not arise when a user is working alone.[33] It is therefore critical that sufficient feedback be presented to the user to indicate what information is shared and what

[33] A user working alone can update shared data that others might then see; shared data, however, is usually updated only at discrete points after an explicit command, and the user is often able to undo his changes before anybody sees them.

information only he can see (e.g., by clean separation of shared and private spaces on the display) and to indicate (perhaps by highlighting) which "mode", shared or private, he is currently working in.

In designing a particular real-time conferencing system, the designer may wish to restrict some of the available degrees of freedom so that the complexity of the user interface is manageable rather than overwhelming and the probability of user error is reduced. We have tried to illustrate this design process by developing an example in considerable detail and explaining our design decisions on the basis of common-sense observations about how people interact in meetings. It is hoped that by systematically addressing the design issues that we have listed, and making a reasoned choice in each case from the options presented, a designer will be able to select a set of functions and an interface that is well-suited to his particular application and user community. The designer might also wish to allow the users themselves to select from a set of options based on their personal preferences. In this case, it is desirable to provide a default selection of options so that users who are unaware of the distinctions are not burdened with having to make a choice. Or, one knowledgeable user could make a selection of options for a given conference, freeing the others from this burden; this would be particularly useful in an experimental system for "human factors" testing (described below) of different options.

We have also examined how the real-time conference functions described might be implemented. Various protocols for replicating and coordinating data at participants' workstations were described, and compared on the basis of the response time perceived by the participants to different operations. In particular, since consistency among a collection of sites requires some synchronization and consequent delay (in the form of either a central controller or a distributed locking or timestamp scheme), response time can be improved if inconsistent data (updates that "never happened" and must be done over) is occasionally tolerated at the user interface. Effective use of this method requires that such inconsistency in fact happen only infrequently; this can be ensured using the interface design techniques described ("reservations") for minimizing concurrent conflict among participants' commands.

Using the given implementation techniques effectively, when developing a particular conferencing system for a particular computing and communication environment, requires trading off user interface functionality (richness of the data displayed and operations on it)

against response time. In a heterogeneous environment, with networks and processors and displays of varying capability, it is best to make this tradeoff dynamically in order to obtain the most effective use of the resources (workstations and communication links) involved in a particular conference. We devised an architecture, named *Ensemble*, that permits the exchange of as many levels of "meta-information", describing run-time parameters, as are needed in order to arrive at a reasonable choice of functions and implementation. Such negotiations can be repeated as often as desired during a conference, as participants leave and join and as network conditions change.

7.1 Future Directions

The main direction in which we see this research continuing is more extensive implementation and testing of the ideas presented. We list the important research questions that we think will need to be addressed.

First, the various design options and conference management policies we have presented need to be evaluated in terms of their usefulness in practice. Short of prolonged experimenting with a real-life implementation, such "human factors" testing would best be done in a laboratory setting with high-bandwidth and low-delay connections among high-performance workstations; a local area network or even dedicated wires would be suitable. This would provide the ideal environment for evaluating design options based solely on their perceived usefulness, without the complicating factors introduced by lengthy or variable communication delays. This use of an "infinite bandwidth" and "zero delay" environment has been applied by Chapanis [16] and others since for evaluating different communication media. The effect of communication delay on user satisfaction and performance can be separately tested by introducing artificial delays into the ideal system. User satisfaction and anxiety can be estimated by questionnaire techniques, such as used by Good [41, 42], and users' comprehension of conference functions can be judged by observing whether and how often they use various commands, how often they make errors, and how well and how quickly they respond to unexpected situations such as concurrent interfering commands by other users. It may also be useful to measure "social" effects, such as how evenly or unevenly the use of the system is distributed among the different participants (as has been done for face-to-face meetings by Brecht [11]). Or, the length of time to reach a problem solution, and the

degree to which individual members are satisfied with that solution, may be measured in comparison with other communication modes such as face-to-face, voice-only conferencing, and video conferencing. Such comparison of communication media has been perförmed in the past [16], but the only form of computerized communication included in such studies is teletype-based, not interactive manipulation of a "shared space" as we have described it. It is important to use a well-understood problem or game, such as the crossword puzzle we chose for our prototype, so that the measurements are not skewed by the subjects' need to learn and understand a complex application interface.

At the implementation level, more performance analysis and comparison of the various available protocols is needed, in different implementation environments. Given an application and an anticipated "load model", i.e., pattern of usage of different commands by the participants, it should be possible to compute, analytically or by simulation, approximate relationships between bandwidth, delay, and number of participants. This can be used to determine how much bandwidth is needed to realize a given maximum delay with a given number of participants, or how the delay increases with the number of participants given a fixed available bandwidth. A major problem here is obtaining a reasonable load model for the analysis; no such data is currently available because real-time conferencing is relatively new. If a laboratory were set up as described above, it would be an ideal source of data for a load model. Such approximate analyses will not be sufficient to fully anticipate how a conferencing system performs in an environment where a conference must compete for resources with other applications and possibly other conferences; that will only be learned through experience with an actual implementation.

7.2 Relevance to Distributed Computing

We have identified many services that a system should provide in order to make real-time conferences a convenient and useful tool. These services are interesting and useful in their own right, even in the absence of real-time conferences. While we have not made direct contributions in these areas, we list them here in the hope that their usefulness for real-time conferencing will provide further impetus for more widespread development of such services:

- Lookup services, for finding resources, users, and objects.
- Recording the history of objects, using a combination of complete copies

("checkpoints") and incremental or "relative" changes.

- Access control and authentication.
- Integrated systems providing uniform access to multiple applications and object types.
- Software version control and dynamic linking of code.
- "Non-sequential" communication protocols based on unreliable datagrams.
- Multicast communication, both at the network level and at higher levels ("reliable multicast").

The problem of distributing an application program between a workstation and a remote host has been studied in contexts such as "satellite graphics" [47], "distributed editing" [39, 112], and general-purpose programming languages [4]. The thrust of these systems has been to improve performance and response time by reducing communication between host and workstation and by local execution of operations at the workstation. Our concept of "partial replication" of the shared space of a conference is derived in part from this work. There is one critical difference, in that past systems have addressed the problem in the context of a single user's workstation and a remote host whereas a real-time conference involves multiple workstations. This limits the freedom of individual workstations to perform actions on their own before informing other sites, because consistency among the different workstations requires a synchronization mechanism such as approval from a central controller site. We have considered local execution of operations at a workstation in order to improve response time, but unlike single-user distributed editing or graphics it is not acceptable for a workstation in a real-time conference to "batch" several locally-executed operations before informing the other sites; all other workstations must be informed quickly of each operation (either directly or via the controller) so that they may update their displays. Batching of operations may be necessary in cases of extremely limited bandwidth, but then the conference would be somewhat less than "real-time".

A slightly different approach to distributed application programming is the development of "remote procedure call" techniques for programming languages [8, 77]. Many of these have been found convenient and useful in practice. However, because of the inability to meaningfully pass "pointers" between different address spaces, remote procedure call mechanisms do not consider the problem of replicating data objects at two or more sites. Either the same data must be transmitted repeatedly with each remote call, or the application programmer must devise a "naming" mechanism so that data can be sent once and then

referred to in later calls using its name or "id". This problem, the naming of application objects and the tracking of changes to and versions of such objects, has been a major topic of this thesis.

Given some naming and replication scheme for a particular conferencing application, remote procedure call might be considered to be a convenient tool for implementing it. There are some problems, however, that will limit the usefulness of remote procedure calls for real-time conferences:

- The inability to perform "multicast", i.e., one identical remote call to multiple sites. Separate remote calls must be made to each remote site, which may be acceptable for small conferences (e.g., two or three participants) but will degrade performance for larger conferences. Part of the problem here is defining an appropriate semantics for a call that expects several returns, some of which might never happen or be delayed for much longer than the others.
- The inability to "pipeline" remote calls, i.e., issue a new call before the previous one has returned, except by convoluted methods involving a separate process for each pipelined call. This restricts the ability of participants to "type ahead". (This problem is in fact not peculiar to real-time conferencing, but also applies to single-user interactive communication such as remote login. For such applications, lower-level protocols that do not enforce a strict call/return discipline are typically used instead.)
- Remote procedure calls impose an inherent asymmetry in that a programmer must decide which message is the "call" and which is the "return"; in many cases, a given message may have characteristics of both a call and a return.

We have preferred to avoid the above problems altogether by dealing with communication at a lower level than procedure calls, namely asynchronous messages where the sender of a message does not block until a reply is received. (Where there is a direct relationship between a given message and some previous one, in either direction of communication, it is still possible for the message to indicate that it constitutes some form of "reply" to the previous message. A given message can thus both provide a reply and ask for a reply, which is not possible with remote procedure calls.) The problem with asynchronous message communication, compared with remote procedure calls, is that it is too unstructured and increases programming complexity. More research and practical experience is needed in designing primitives that balance the programming ease of remote procedure calls against the power and flexibility of asynchronous message communication, and that also allow for

multicast communication.[34]

In the past, two main classes of distributed applications have been developed that can be characterized as "multicast", i.e., involving communication among more than two sites:[35] network voice conferencing [20, 49], and distributed databases [7]. These represent opposite extremes of the tradeoff between consistency (or reliability) and delay (or response time). Voice conferencing is characterized by strict real-time delay requirements, which can only be met by sacrificing some consistency, i.e., a participant may not receive a given voice packet in time, with some resulting loss of quality in the voice output that he hears. Except for a small amount of "conference control" information (addresses of participants in the conference, who holds the "floor", and so on), there is little lasting state information in a voice conference.

A distributed database (or distributed file system), on the other hand, has a large amount of state information with strong consistency requirements; considerable delay (as with "two-phase commit" [44, 71]) may be involved in achieving the desired consistency. Real-time conferencing lies somewhere in between these two extremes; while consistency of shared information is necessary, low delay is also important in order for a conference to be interactive. We identified a particular method of trading one for the other, by allowing occasional temporary inconsistency of displayed information while at the same time preserving eventual consistency by not allowing such inconsistencies to persist.

Distributed databases also place particular emphasis on crash recovery, which is less of a problem for the brief duration of a real-time conference. The same crash recovery techniques used in distributed databases are applicable to real-time conferences, but again response time requirements dictate that users see each others' updates before the system is certain that they will survive a crash. When crash recovery is invoked, therefore, some updates that users saw may be lost.

The "database" in a real-time conference is characterized by the replication of identical

---

[34] The distributed "V" kernel supports a form of high-level multicast communication [17] in which the sender of a multicast message blocks until one reply is received; the sender can subsequently check, as often as it wishes, to see if additional replies have arrived. This facility is a mixture of synchronous and asynchronous message-passing, and experience with it is still limited.

[35] Multi-site communication is also used within computer networks for routing, locating servers, and so on [10]. These applications, however, are mainly broadcast, i.e., involve an unknown set of sites, rather than multicast.

data at multiple sites. For this particular kind of distributed database, it appears that the problems of initiation and negotiation and of ensuring consistency are solved with less communication overhead and delay using centralized rather than decentralized control. This result does not hold for the important special case of a conference or distributed database with only two sites. (Nor does it apply to more conventional distributed databases where different data is replicated at different sites.) For two-person conferences, decentralized control performs slightly better but centralized control retains the virtue of simplicity; this reflects the current status of two-party network protocols, most of which are decentralized but many of which exhibit a "master-slave" relationship, i.e., use centralized control.

The database in a real-time conference is also highly dynamic in that information and sites may be added and removed. Some distributed database systems support dynamic exchange of "schema" [50, 79] or "catalog" [75] information describing what data is available; this is similar to our notion of a conference "description" that is used for getting sites started in a conference. However, sharing of data in the above distributed database systems is done by *pairwise* negotiations between the site holding the data and the site wishing to access it, on the assumption that not all data will necessarily be of interest to all sites. The opposite is in fact true for real-time conferences, and our conference architecture therefore presents methods for *groups* of sites to negotiate a data sharing protocol without requiring a separate negotiation for each pair of sites.

The "concurrency control" problem that we have studied for real-time conferences differs somewhat from the problem that most distributed database systems have addressed. The objective in a database system is usually to isolate different users' "transactions" from each other, while in a real-time conference each user's actions are intended to be immediately visible to all participants. For real-time conferences we have therefore assumed that each individual operation by a user is itself atomic, and have concentrated on the problem of longer-term concurrency control across a series of operations, i.e., ensuring that each operation has the effect that the user intended based on what he saw at the time he entered the operation. Similar problems are likely to arise with asynchronous communication as well, considering that a user's involvement with a given object (e.g., a document or a circuit design) may last for days or weeks during which time it may not be desirable to prevent other users from working on the same object. Previous attempts at addressing this problem (e.g., Violet [37]) have been unduly restrictive in that either one user locks out all others for a long

period of time or a user loses a lot of work when his locks are "broken" by another user's transaction. The methods we have presented for concurrency control, namely reservations and validation, are extensions of locking [44] and "optimistic" methods [68], respectively, used in database systems. Our extensions allow more flexible policies for dealing with concurrent conflict, including time limits on reservations, saving "alternatives" rather than losing work when there is a conflict, and allowing users to decide how to deal with concurrent conflict. We have observed that validation, which detects rather than prevents conflicts, is practical only when the probability of concurrent conflict is low, which might often be the case when users are acting asynchronously. Various combinations of reservations and validation are likely to be useful for handling longer-term interactions among users of a large distributed system, and the implementation and evaluation of these appears to be an interesting avenue for further research.

References

[1]

Proc. Human Factors in Computing Systems.
National Bureau of Standards Institute for Computer Sciences and Technology, Gaithersburg, MD., March 1982.

[2]

Proc. CHI'83 Human Factors in Computing Systems.
Boston, MA., December 1983.

[3]

Proc. ACM SIGPLAN/SIGOA Symposium on Text Manipulation.
Portland, Oregon, June 1981.

[4] Balzer, R., Cooperband, A., Feather, M., London, P., and Wile, D.
Application Downloading.
In *Proc. 5th International Conference on Software Engineering*, pages 450-459. IEEE, March, 1981.

[5] Bayer, R., and McCreight, E.
Organization and Maintenance of Large Ordered Indexes.
Acta Informatica 1(3):173-189, 1972.

[6] Berglund, Eric J., and Cheriton, David R.
Amaze: A Distributed Multi-Player Game Program using the Distributed V Kernel.
In *Proc. 4th International Conference on Distributed Computing Systems*. IEEE, May, 1984.

[7] Bernstein, Philip A., and Goodman, Nathan.
Concurrency Control in Distributed Database Systems.
Computing Surveys 13(2):185-221, June, 1981.

[8] Birrell, Andrew D., and Nelson, Bruce Jay.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[9] Birrell, Andrew D., Levin, Roy, Needham, Roger M., and Schroeder, Michael D.
Grapevine: An Exercise in Distributed Computing.
Communications of the ACM 25(4):260-274, April, 1982.

[10] Boggs, David R.
Internet Broadcasting.
Technical Report CSL-83-3, Xerox Palo Alto Research Center, October, 1983.
(Ph.D. thesis, Stanford University, 1982.)

[11] Brecht, Mark Allen.
A Study of Meeting and Conference Behavior.
Technical Report, Johns Hopkins University, Dept. of Psychology, July, 1979.
(Ph.D. thesis.)

[12] Bury, K., Boyle, J., Evey, R.J., and Neal, A.
Windowing vs. Scrolling on a Visual Display Terminal.
In *Proc. Human Factors in Computing Systems*, pages 41-44. Gaithersburg, MD., March, 1982.

[13] CCITT.
Interface Between Data Terminal and Data Circuit-Terminating Equipment for Terminals Operating in the Packet Mode on Public Data Networks.
Recommendation X.25, CCITT, November, 1978.

[14] Cerf, V., and Kahn, R.
A Protocol for Packet Network Interconnection.
IEEE Transactions on Communication COM-22(5):637-648, May, 1974.

[15] Chang, Jo-Mei, and Maxemchuk, N.F.
Reliable Broadcast Protocols.
Technical Note 4, Bell Laboratories Computer Technology Research Laboratory, January, 1983.

[16] Chapanis, Alphonse, Ochsman, Robert B., Parrish, Robert N., and Weeks, Gerald D.
Studies in Interactive Communication: I. The Effects of Four Communication Modes On the Behavior of Teams During Cooperative Problem-Solving.
Human Factors 14(6):487-509, 1972.

[17] Cheriton, D.R., and Zwaenepoel, W.
One-to-many Interprocess Communication in the V-system.
In *Proc. Symposium on Communication Architectures and Protocols*. ACM SIGCOMM, June, 1984.

[18] Cimral, John J.
Integrating Coordination Support into Automated Information Systems.
Master's thesis, Massachusetts Institute of Technology Dept. Electrical Engineering and Computer Science, May, 1983.

[19] Clark, David D., Pogran, Kenneth T., and Reed, David P.
An Introduction to Local Area Networks.
Proceedings of the IEEE 66(11):1497-1517, November, 1978.

[20] Cohen, Dan.
A Protocol for Packet-Switching Voice Communication.
Computer Networks 2(4/5):320-331, September/October, 1978.

[21] Comer, Douglas.
The Ubiquitous B-Tree.
ACM Computing Surveys 11(2):121-137, June, 1979.

[22] Cooper, Geoffrey H.
An Argument for Soft Layering of Protocols.
Technical Report 300, Massachusetts Institute of Technology Laboratory for Computer Science, May, 1983.
(S.M. thesis.)

[23] Dalal, Yogen K.
Use of Multiple Networks in the Xerox Network System.
IEEE Computer 15(10):82-92, October, 1982.

[24] Dalal, Yogen K., and Metcalfe, Robert M.
Reverse Path Forwarding of Broadcast Packets.
Communications of the ACM 21(12):1040-1048, December, 1978.

[25] Davidson, J., Hathaway, W., Postel, J., Mimno, N., Thomas, R., and Walden, D.
The Arpanet TELNET Protocol: Its Purpose, Principles, Implementation, and Impact on Host Operating System Design.
In *Proc. Fifth Data Communications Symposium*, pages 4-10 to 4-18. September, 1977.

[26] Ellis, Clarence A.
A Robust Algorithm for Updating Duplicate Databases.
In *Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 146-158. Lawrence Berkeley Labs, CA., May, 1977.

[27] Engelbart, Douglas C.
NLS Teleconferencing Features: the Journal, and Shared-Screen Telephoning.
In *Fall COMPCON 75 Digest of Papers*, pages 173-177. September, 1975.

[28] Engelbart, Douglas C.
Toward High-Performance Knowledge Workers.
In *Office Automation Conference Digest*, pages 279-290. AFIPS, April, 1982.

[29] Engelbart, Douglas C., and English, William K.
A Research Center for Augmenting Human Intellect.
In *Proc. Fall Joint Computing Conference*, pages 395-410. AFIPS Conference Proceedings Vol. 33, December, 1968.

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

[31] Foley, James D. and van Dam, Andries.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley, 1982.

[32] Forgie, James W.
ST - A Proposed Internet Stream Protocol.
Arpa Internet Experimental Note IEN-119, M.I.T. Lincoln Laboratory, September, 1979.

[33] Furuta, Richard, Scofield, Jeffrey, and Shaw, Alan.
Document Formatting Systems: Survey, Concepts, and Issues.
ACM Computing Surveys 14(3):417-472, September, 1982.

[34] Garcia-Molina, Hector.
Elections in a Distributed Computer System.
IEEE Transactions on Computers C-31(1):48-59, January, 1982.

[35] Gerla, Mario, and Kleinrock, Leonard.
Flow Control Protocols.
In P.E. Green, editor, *Computer Network Architectures and Protocols*, pages 361ff.. 1982.

[36] Gifford, David K.
Weighted Voting for Replicated Data.
In *Proc. Seventh Symposium on Operating Systems Principles*, pages 150-162. ACM, November, 1979.

[37] Gifford, David K.
Violet, an Experimental Decentralized System.
Computer Networks 5(6):423-433, December, 1981.

[38] Gifford, David K.
Information Storage in a Decentralized Computer System.
Technical Report CSL-81-8, Xerox Palo Alto Research Center, March, 1982.
(Ph.D. thesis, Stanford University, 1981.)

[39] Goldberg, Robert N.
Software Design Issues in the Architecture and Implementation of Distributed Text Editors.
Technical Report DCS-TR-110, Rutgers University Dept. Computer Science, 1981.
(Ph.D. thesis.)

[40] Goldstein, Ira P., and Roberts, R. Bruce.
NUDGE, A Knowledge-Based Scheduling Program.
In *Proc. International Joint Conference on Artificial Intelligence*, pages 257-263. Cambridge, MA., August, 1977.

[41] Good, Michael D.
An Ease of Use Evaluation of an Integrated Editor and Formatter.
Technical Report 266, Massachusetts Institute of Technology Laboratory for Computer Science, August, 1981.
(S.M. thesis.)

[42] Good, Michael D.
An Ease of Use Evaluation of an Integrated Document Processing System.
In *Proc. Human Factors in Computing Systems*, pages 142-147. Gaithersburg, MD., March, 1982.

[43] Gosling, James.
A Redisplay Algorithm.
In *Proc. Symposium on Text Manipulation*, pages 123-129. ACM SIGPLAN/SIGOA, June, 1981.

[44] Gray, James N.
Notes on Data Base Operating Systems.
In *Lecture Notes on Computer Science, Vol. 60*, pages 393-481. Springer-Verlag, 1978.

[45] Greif, Irene.
PCAL: A Personal Calendar.
Technical Memo TM-213, Massachusetts Institute of Technology Laboratory for Computer Science, January, 1982.

[46] Greif, Irene.
The User Interface of a Personal Calendar Program.
In *Proceedings of the NYU Symposium on User Interfaces.* New York University, May, 1982.
(Reprinted in MIT/LCS/TM-218, 1982.)

[47] Hamlin, Griffith, and Foley, James D.
Configurable Applications for Graphics Employing Satellites (CAGES).
In *SIGGRAPH '75 Proceedings*, pages 9-19. ACM Computer Graphics 9(1), Spring, 1975.

[48] Hammer, M., Ilson, R., Anderson, T., Good, M., Rosenstein, L., Niamir, B., Schoichet, S., and Gilbert, E.
The Implementation of ETUDE, an Integrated and Interactive Document Preparation System.
In *Proc. Symposium on Text Manipulation*, pages 137-146. ACM SIGPLAN/SIGOA, June, 1981.

[49] Heggestad, Harold M., and Weinstein, Clifford J.
Voice and Data Communication Experiments on a Wideband Satellite/Terrestrial Internetwork System.
In *Proc. International Conference on Communications*, pages A1.1.1-A1.1.8. IEEE, June, 1983.

[50] Heimbigner, Dennis M.
A Federated Architecture for Database Systems.
Technical Report TR-114, University of Southern California Dept. Computer Science, August, 1982.
(Ph.D. thesis.)

[51] Henderson, Cecelia E.
Locating Migratory Objects in an Internet.
Master's thesis, Massachusetts Institute of Technology Dept. Electrical Engineering and Computer Science, August, 1982.
(Available as M.I.T. Computation Structures Group Memo 224.)

[52] Henderson, D.A., and Myer, T.H.
Issues in Message Technology.
In *Proc. Fifth Data Communications Symposium*, pages 6-1 to 6-9. September, 1977.

[53] Herlihy, Maurice P.
Transmitting Abstract Values in Message.
Technical Report 234, Massachusetts Institute of Technology Laboratory for Computer Science, May, 1980.
(S.M. thesis.)

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

[55] Hiltz, Starr Roxanne and Turoff, Murray.
The Network Nation: Human Communication via Computer.
Addison-Wesley, 1978.

[56] Hiltz, Starr Roxanne and Turoff, Murray.
The Evolution of User Behavior in a Computerized Conferencing System.
Communications of the ACM 24(11):739-751, November, 1981.

[57] Hoare, C.A.R.
Monitors: An Operating Systems Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[58] Ingalls, D.
The Smalltalk Graphics Kernel.
BYTE 6(8), August, 1981.

[59] International Standards Organization ISO/TC97/SC16/WG1,2,3.
Reference Model of Open Systems Architecture, version 3, November 1978.

[60] International Standards Organization ISO/TC97/SC16/WG6.
Information Processing Systems - Open Systems Interconnection - Transport Protocol Specification, 1982.
(Reprinted in ACM Computer Communication Review 12, 3-4 (July/Oct 1982); also Arpanet RFC 892.)

[61] Israel, Jay E., and Linden, Theodore A.
Authentication in Office System Internetworks.
ACM Transactions on Office Information Systems 1(3):193-210, July, 1983.

[62] Jacobs, Irwin M., Binder, Richard, and Hoversten, Estil V.
General Purpose Packet Satellite Networks.
Proceedings of the IEEE 66(11):1448-1467, November, 1978.

[63] Jensen, Kathleen, and Wirth, Niklaus.
PASCAL User Manual and Report.
Springer-Verlag, 1978.

[64] Johnson, Paul R, and Thomas, Robert H.
The Maintenance of Duplicate Databases.
Arpanet RFC 677, Bolt, Beranek and Newman, January, 1975.

[65] Kamae, T., Ohtsuka, S., and Sato, Y.
Sketchfax - A Terminal Having Telewriting and Facsimile Capabilities.
In *Proc. International Conference on Communications*, pages D3.4.1-D3.4.5. IEEE, June, 1983.

[66] Kedziersky, Beverly I.
Communication and Management Support in System Development Environments.
In *Proc. Human Factors in Computing Systems*, pages 163-168. Gaithersburg, MD., March, 1982.

[67] Kernighan, Brian W., and Mashey, John R.
The Unix Programming Environment.
IEEE Computer 14(4):12-24, April, 1981.

[68] Kung, H.T., and Robinson, John T.
On Optimistic Methods for Concurrency Control.
ACM Transactions on Database Systems 6(2):213-226, June, 1981.

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

[70] Lampson, Butler W., and Redell, David D.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-117, February, 1980.

[71] Lampson, Butler W., and Sturgis, Howard E.
Crash Recovery in a Distributed Data Storage System.
Draft Report, Xerox Palo Alto Research Center, April 1979.

[72] Lantz, Keith A., and Rashid, Richard F.
Virtual Terminal Management in a Multiple Process Environment.
In *Proc. Seventh Symposium on Operating Systems Principles*, pages 86-97. ACM, November, 1979.

[73] LeLann, Gerard.
Algorithms for Distributed Data-Sharing Systems which use Tickets.
In *Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 259-272. Lawrence Berkeley Labs, CA., August, 1978.

[74] Lemmons, Phil.
BYTE West Coast: A Guided Tour of VisiOn.
BYTE 8(6):256 ff., June, 1983.

[75] Lindsay, Bruce G.
Object Naming and Catalog Management for a Distributed Database Manager.
In *Proc. 2nd International Conference on Distributed Systems*, pages 31-40. Paris, France, April, 1981.

[76] Lipinski, Hubert, and Adler, Richard.
Electronic Communication for Interactive Group Modeling.
In S. Schoemaker, editor, *Computer Networks and Simulation II*, pages 251-277. North-Holland, 1982.

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

[78] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J.C., Scheifler, R., and Snyder, A.
CLU Reference Manual.
Lecture Notes on Computer Science Volume 114, Springer-Verlag, 1981.

[79] McLeod, Dennis, and Heimbigner, Dennis.
A Federated Architecture for Database Systems.
In *Proc. National Computer Conference*, pages 283-289. AFIPS Conference Proceedings Vol. 49, May, 1980.

[80] Medhekar, R.A., Singh, H., Winchell, D.F., and Salchenberger, S.N.
A Layered Operational Software Architecture for Audiographics Conferencing.
In *Proc. International Conference on Communications*, pages C7.2.1-C7.2.7. IEEE, June, 1983.

[81] Menasce, Daniel A., Popek, Gerald J., and Muntz, Richard R.
A Locking Protocol for Resource Coordination in Distributed Databases.
ACM Transactions on Database Systems 5(2):103-138, June, 1980.

[82] Metcalfe, Robert M., and Boggs, David R.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM 19(7):395-404, July, 1976.

[83] Meyrowitz, Norman, and van Dam, Andries.
Interactive Editing Systems (Parts I and II).
ACM Computing Surveys 14(3):321-415, September, 1982.

[84] Miller, Robert B.
Response Time in Man-Computer Conversational Transactions.
In *Proc. Fall Joint Computer Conference*, pages 267-277. AFIPS Conference Proceedings Vol. 33, December, 1968.

[85] Mitchell, Neal B., and McLean, Michael.
Demonstration of Palette Teleconferencing Software, American Society of Civil Engineers Annual Conference, New Orleans, October 1982.

[86] Mockapetris, Paul V.
Analysis of Reliable Multicast Algorithms for Local Networks.
In *Proc. 9th Data Communications Symposium*, pages 150-157. October, 1983.

[87] Montgomery, Warren A.
Robust Concurrency Control for a Distributed Information System.
Technical Report 207, Massachusetts Institute of Technology Laboratory for Computer Science, December, 1978.
(Ph.D. thesis.)

[88] Murphy, Daniel L.
Storage Organization and Management in TENEX.
In *Proc. Fall Joint Computer Conference*, pages 23-31. AFIPS Conference Proceedings Vol. 41, 1972.

[89] Needham, Roger M., and Schroeder, Michael D.
Using Encryption for Authentication in Large Networks of Computers.
Communications of the ACM 21(12):993-999, December, 1978.

[90] Newman, William M., and Sproull, Robert F.
Principles of Interactive Computer Graphics, second edition.
McGraw-Hill, 1979.

[91] O'Brien, Michael T.
A Network Graphical Conferencing System.
Note N-1250-ARPA, Rand Corporation, August, 1979.

[92] Oppen, Derek C., and Dalal, Yogen K.
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.
ACM Transactions on Office Information Systems 1(3):230-253, July, 1983.

[93] Lorne Parker, editor.
Teleconferencing and Interactive Media.
University of Wisconsin Extension, Madison, 1980.

[94] Parker, D.S., Popek, G.J., Rudisin, G, Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., and Kline, C.
Detection of Mutual Inconsistency in Distributed Systems.
IEEE Transactions on Software Engineering SE-9(3):240-246, May, 1983.

[95] Pferd, W., Peralta, L. A., and Prendergast, F. X.
Interactive Graphics Teleconferencing.
IEEE Computer 12(11):62-72, November, 1979.

[96] Postel, Jon.
User Datagram Protocol.
Arpanet RFC 768, USC/Information Sciences Institute, August, 1980.

[97] Postel, Jon (editor).
Transmission Control Protocol - DARPA Internet Program Protocol Specification.
Arpanet RFC 793, USC/Information Sciences Institute, September, 1981.

[98] Postel, Jon (editor).
Internet Protocol - DARPA Internet Program Protocol Specification.
Arpanet RFC 791, USC/Information Sciences Institute, September, 1981.

[99] Reed, David P.
Naming and Synchronization in a Decentralized Computer System.
Technical Report 205, Massachusetts Institute of Technology Laboratory for Computer Science, September, 1978.
(Ph.D. thesis.)

[100] Reed, David P.
Implementing Atomic Actions on Decentralized Data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[101] Rubin, D., Craighill, E., and Rom, R.
Topics in the Design of a Natural Teleconferencing System.
In *National Telecommunications Conference Record*, pages 12.4.1-12.4.5. IEEE, 1978.

[102] Saltzer, Jerome H., and Schroeder, Michael D.
The Protection of Information in Computer Systems.
Proceedings of the IEEE 63(9):1278-1308, September, 1975.

[103] Sarin, Sunil K.
Interactive On-Line Conferences.
PhD thesis, M.I.T. Department of Electrical Engineering and Computer Science, June, 1984.

[104] Sarin, Sunil, and Greif, Irene.
Software for Interactive On-Line Conferences.
In *Proc. 2nd Conference on Office Information Systems.* ACM, Toronto, Canada, June, 1984.

[105] Schmidt, Eric E.
Controlling Large Software Development in a Distributed Environment.
Technical Report CSL-82-7, Xerox Palo Alto Research Center, December, 1982.
(Ph.D. thesis, University of California, Berkeley.)

[106] Schneider, Fred B., Gries, David, and Schlichting, Richard B.
Fault-Tolerant Broadcasts.
Technical Report 83-519, Cornell University Dept. Computer Science, August, 1983.
(To appear in Science of Computer Programming.)

[107] Shoch, John F., Dalal, Yogen K., Redell, David D., and Crane, Ronald C.
Evolution of the Ethernet Local Computer Network.
IEEE Computer 15(8):10-28, August, 1982.

[108] Smith, D.C., Irby, C., Kimball, R., and Harslem, E.
The Star User Interface: An Overview.
In *Proc. National Computer Conference*, pages 515-528. AFIPS Conference Proceedings Vol. 51, June, 1982.

[109] Solomon, Marvin, Landweber, Lawrence H., and Neuhengen, Donald.
The CSNET Name Server.
Computer Networks 6(3):161-172, July, 1982.

[110] Sproull, Richard F., and Cohen, Dan.
High-Level Protocols.
Proceedings of the IEEE 66(11):1371-1386, November, 1978.

[111] Stallman, Richard M.
EMACS, the Extensible, Customizable Self-Documenting Display Editor.
In *Proc. Symposium on Text Manipulation*, pages 147-156. ACM SIGPLAN/SIGOA, June, 1981.

[112] Stallman, Richard M.
A Local Front End for Remote Editing.
Memo 643, Massachusetts Institute of Technology Artificial Intelligence Laboratory, February, 1982.

[113] Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R.
Alto: A Personal Computer.
In D. Sieworek, G. Bell, and A.M. Newell, editors, *Computer Structures: Readings and Examples, 2nd ed.*. McGraw-Hill, 1981.
(Also Xerox PARC Technical Report CSL-79-11, 1979.)

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

[115] Tsichritzis, D.
Form Management.
Communications of the ACM 25(7):453-478, July, 1982.

[116] Wall, David W.
Selective Broadcast in Packet-Switched Networks.
In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 239-258. Lawrence Berkeley Labs, CA., February, 1982.

[117] Xerox Corporation.
Courier: The Remote Procedure Call Protocol.
System Integration Standard XSIS-038112, Xerox, December, 1981.

[118] Xerox Corporation.
Internet Transport Protocols.
System Integration Standard XSIS-028112, Xerox, December, 1981.

[119] Zdonik, Stanley B.
Object Management System Concepts.
In *Proc. 2nd Conference on Office Information Systems*. ACM, Toronto, Canada, June, 1984.

Biographical Note

Sunil Sarin was born in Delhi, India, in 1953, and spent his early childhood in Hong Kong. He attended the Bishop Cotton School, Simla, India, completing the Indian School Certificate in 1968. He received the Bachelor of Technology degree from the Indian Institute of Technology, Kharagpur, India, in 1974. At the Massachusetts Institute of Technology, Cambridge, U.S.A., he received the Master of Science and Electrical Engineer degrees in 1977 and 1978, respectively, and will receive the Doctor of Philosophy in Computer Science in 1984. Dr. Sarin worked as a Computer Scientist at Computer Corporation of America, Cambridge, in 1978, was awarded an I.B.M. Graduate Fellowship in 1979, and has presented papers at the ACM-SIGMOD Conference on Management of Data (1978) and the ACM-SIGOA Conference on Office Information Systems (1984). He enjoys writing programs and long documents, such as this.