Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems

by

James Michael Allen Begole

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

Clifford A. Shaffer, Chairman

Marc Abrams

Dennis G. Kafura

Ronald D. Kriz

Mary Beth Rosson

December, 1998
Blacksburg, Virginia

Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems

by

James Michael Allen Begole

Committee Chairman: Clifford A. Shaffer

Computer Science and Applications

(ABSTRACT)

This dissertation analyzes the usefulness of existing "conventional" **collaboration-transparency** systems, which permit the shared use of legacy, single-user applications. I find that conventional collaboration-transparency systems do not use network resources efficiently, and they impose an inflexible, tightly coupled style of collaboration because they do not adequately support important groupware principles: concurrent work, relaxed WYSIWIS, group awareness, and inherently collaborative tasks. This dissertation proposes and explores solutions to those deficiencies.

The primary goal of this work is to maintain the benefits of collaboration transparency while relieving some of its disadvantages. To that end, I present an alternate implementation approach that provides many features previously seen only in applications specifically designed to support cooperative work, called **collaboration-aware** applications. The new approach uses a replicated architecture, in which a copy of the application resides on each user's machine, and the users' input events are broadcast to each copy. I discuss solutions to certain key problems in replicated architectures, such as maintaining consistency, unanticipated sharing, supporting late-joiners, and replicating system resources (e.g., files, sockets, and random number generators). To enhance the collaborative usability of a legacy application, the new approach *transparently* replaces selected single-user interface objects with multi-user versions at runtime. There are four requirements of an application platform needed to implement this approach: process migration, run-time object replacement, dynamic binding, and the ability to intercept and introduce low-level user input events. As an instance of this approach, I describe its incorporation into a new Java-based collaboration-transparency system, called Flexible JAMM (Java Applets Made Multi-user).

This dissertation reports the results of a controlled empirical study that evaluated the usefulness of Flexible JAMM versus a representative conventional collaboration-transparency system, Microsoft NetMeeting. The results validate that Flexible JAMM meets its goals, and uncover usability problems in both systems, particularly with respect to using floor control. Additionally, the dissertation reports the results of an informal study that evaluated using Flexible JAMM as a groupware toolkit.

This dissertation demonstrates that it is possible to bring collaboration transparency closer to the advantages afforded by collaboration awareness. Furthermore, the prototype system demonstrates that collaboration-aware toolkits can include multi-user versions of some standard single-user components that require no collaboration-specific programming by the toolkit user. Thus, the results of this research advance the state of the art in both collaboration-transparency systems and collaboration-aware toolkits.

To my beloved grandparents,
Dorothy, Edgar, Elizabeth, and Herman
for their inspirational
love, passion, intelligence, and wisdom.

ACKNOWLEDGEMENTS

I am profoundly grateful to my advisor and committee chair, Clifford A. Shaffer. Throughout my time as a graduate student at Virginia Tech, he has provided logistics, assistance, guidance, ideas, editing and a healthy skeptical ear. I thank Randall B. Smith of Sun Microsystems Laboratories who provided early inspiration, background, and support during and after two internships at JavaSoft and Sun Microsystems Laboratories. I am grateful to the members of my committee for their individual influence on this research. I am indebted to Mary Beth Rosson, who provided considerable background, insight, focus, and refinement throughout the work. I am grateful to Marc Abrams for discussions on object-oriented design regarding our instruction of the sophomore object-oriented design course, and regarding software for the Learning in Networked Communities (LiNC) project. Dennis Kafura's research groups on problem-solving environments, distributed systems, and object-oriented techniques were fruitful forums for implementation discussions. I thank Ron Kriz, who saw this research transition from collaborative visualization to collaborative software in general, for providing an external perspective on collaborative computing on the desktop and in immersive virtual environments. I acknowledge John Harding and Microsoft Corporation for providing software development tools. Many fellow graduate students have had an influence on this work. Dennis Neale and Ray Reaux helped refine the usability study. I am indebted to Philip Isenhour for frequently helping me explore implementation problems and solutions. I want to particularly thank Craig Struble for the initial idea of a replicated approach to collaboration transparency and his work on the early prototype systems. Craig was always willing to spend time away from his own demanding research to discuss this work. I am grateful to the National Science Foundation for support under grants REC-9554206 (Network Infrastructure for Education) and DGE-9553458 (Graduate Research Trainee). Of course, I am grateful to my bride, Florence BeGole, who is eternally patient and supportive. Her love and encouragement have sustained me throughout. Thank you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

DILBERT® reprinted by permission of United Feature Syndicate, Inc.

Recent decades have seen "revolutionary" changes in information technologies. In the 1980s, The personal computer "revolution" sought to break the dominance of expensive centralized computing (Gates *et al.*, 1995; Gupta & Toong, 1985). Today, although the vision of pairing each person with a computer is not yet fully realized, personal computers are pervasive. Worldwide sales of Intel-compatible personal computers were approximately 82 million in 1997 (Kaplan, 1998) and are estimated to reach 90.7 million in 1998 (Turner, 1998). Computer Intelligence, Inc. recently estimated that 45 percent of the approximately 100 million homes in the U.S. contain a computer (McGarvey, 1998).

The 1990s have had a related "revolution." The need to transfer information among today's profusion of computers has led to explosive growth in networking, resulting in a remarkable number of users able to access the Internet. The exact number of Internet users worldwide is unknown, but is estimated to be well over 100 million in July, 1998 (Nua Ltd., 1998; Thibodeau, 1998), approximately double the number from the year before. Although the rate of new users will slow (Clemente, 1998), growth is expected to continue as more nations connect to the global network (Reuters, 1998; Thibodeau, 1998).

Now, with the proliferation of computers and near universal networking, we need to shift our focus from *personal* computing to *inter*-personal computing. People often work together, and, because a computer is used in much of that work, people need computer support for their collaborations. Additionally, network technology allows people to conduct computer-based work from different locations. The cost of travel for face-to-face collaboration makes distributed collaboration appealing, particularly as organizations become increasingly global.

This dissertation deals with systems that support real-time computer-based collaborations. Particularly, this dissertation discusses systems that provide the real-time shared use of legacy, single-user applications. The work addresses both technical and human factors related to the development and use of such systems. The remainder of this chapter defines the scope of this dissertation, describing the problem and defining relevant terms. The chapter includes a summary of this dissertation's contributions, which are elaborated in later chapters. The chapter ends with an overview of the organization of this dissertation and a summary of the contents of later chapters.

1.1 Collaborative Computing

Collaborative interactions fall in a range from **asynchronous**, where the points of interaction are separated by relatively long periods of time, to **synchronous**, where the interactions are simultaneous or separated by short periods of time (Grudin, 1994). There are many examples of successful asynchronous collaborative applications that are commonly used by casual computer users: electronic mail (e-mail), discussion groups (e.g., Usenet), file version control (Cederqvist *et al.*, 1993), and publishing information on the World Wide Web. In a recent survey conducted by the Graphics, Visualization, and Usability (GVU) Center of Georgia Tech (Read *et al.*, 1998), respondents considered some of these commonplace asynchronous technologies to be "indispensable."

Synchronous collaborative systems are not nearly as prevalent as asynchronous. However, synchronous collaborative applications do exist, and more are emerging daily, including media spaces (Bly *et al.*, 1993), shared whiteboards (Streitz *et al.*, 1994), joint exploration of data (Lee *et al.*, 1996), Multi-User Domains (MUD) (Nichols *et al.*, 1995), joint document editing (Mitchell *et al.*, 1995), as well as real-time communication via text, audio and video conferencing systems (AIM, 1998; Dutta-Roy, 1998). Whereas telephony is a widely used synchronous collaborative technology, use of *computer*-based synchronous collaborative systems is largely confined to computer enthusiasts and early technology adopters (Dutta-Roy, 1998; Grudin, 1994).

The reasons for the lack of acceptance of synchronous collaborative systems are both technical and social. Many of the technical challenges are being addressed by new approaches that expand the limits of current technologies, such as Internet Protocol version 6 (IPv6) (Miller, 1997; Tanenbaum, 1996) and Gigabit Ethernet (Seifert, 1998). The social hurdles are also being researched, most prominently in a subfield of human-computer interaction called computer-supported cooperative work (CSCW) (Grudin, 1994), which explores the wide spectrum of collaborative computing.

Synchronous collaborative applications are customarily placed in two categories. The first, **collaboration awareness**, includes applications specifically designed to support cooperative work. The developers were *aware* that their application would be used collaboratively. The second category is referred to as **collaboration transparency** because the sharing is provided by a mechanism that is unknown, or *transparent,* to the application and its developers (Begole *et al.*, 1997a). The primary distinctions between collaboration-aware applications and collaboration-transparency systems are summarized in the next section and described in detail in Chapter 2.

1.1.1 Collaboration Awareness

A collaboration-aware application is one that was developed with the aim of allowing multiple users to work cooperatively. Collaboration-aware applications may be either asynchronous (e.g., e-mail) or synchronous (e.g., multi-player games).

Developing synchronous groupware applications has requirements beyond those of single-user applications (Patterson, 1991). Such applications may be developed *ad hoc* or with a groupware toolkit, which are surveyed in Section 2.4.1. The goal of groupware toolkits is to facilitate the development of collaboration-aware applications so that they are only "slightly more difficult" to develop than otherwise functionally equivalent single-user applications (Roseman & Greenberg, 1996a; Lee

et al., 1996). Nevertheless, an application programmer must explicitly "build in" collaboration through calls to the toolkit support functions. Therefore, whether employing a groupware toolkit or not, a collaborative-software developer must make some additional effort over that of developing an otherwise functionally equivalent single-user application.

1.1.2 Collaboration Transparency

A collaboration-transparency system allows the synchronous shared use of legacy, single-user software by modifying the application's run-time environment. The collaboration-transparency system itself is collaboration-aware, but the single-user application is collaboration-unaware.

Several collaboration-transparency systems now exist, particularly for the X Window System. Examples include XTV (Abdel-Wahab & Feit, 1991), Hewlett Packard's SharedX (Garfinkel *et al.*, 1994) and Sun Microsystem's ShowMe SharedApp (Sun Microsystems, Inc., 1998). Recently, Microsoft has released NetMeeting, a freely available collaboration-transparency system for the Windows platform (Microsoft Corp., 1998) (collaboration-transparency systems are surveyed in detail in Section 2.4.2). Because it is free, available for a widely used platform, and supported by a major software vendor, NetMeeting is helping popularize collaboration transparency among casual computer users. Over 45 million copies of have reportedly been downloaded (Dutta-Roy, 1998).

There is an ever-growing set of legacy single-user applications that coworkers may like to use in support of their collaborative tasks. The primary benefit of collaboration-transparency systems is that, by supporting the shared use of such legacy applications, developers make no extra effort whatsoever for their applications to be shared. Additionally, application developers may consider using a collaboration-transparency system as a simple groupware toolkit. For example, a simple chess interface quickly becomes a two-player game when shared via collaboration transparency. As with a physical chess game, players must use social conventions to ensure conformance with the rules of play.

The next section contrasts the support for collaboration provided by collaboration transparency versus awareness. Following that, the research problem and hypothesis are stated.

1.2 Scope of Thesis

This dissertation primarily explores software in support of synchronous collaborations. In particular, this dissertation deals with systems that provide the synchronous shared use of legacy, single-user applications. The work addresses both technical and human factors related to the development and use of such collaboration-transparency systems. In this dissertation, I explore the needs of collaborating users, examine the limitations of existing systems, describe an alternate implementation design, and evaluate its effectiveness. The results have implications for future development of collaborative software.

1.2.1 Problem Statement

During the course of a project, coworkers engage in varying patterns of collaboration: tightly and loosely coupled, synchronous and asynchronous, private and shared, scheduled and opportunistic. For many years, researchers have called for software that allows collaborators to seamlessly transition among such styles of work.

Some collaboration-aware applications provide flexible support for varying styles of collaboration, but currently available collaboration-transparency systems generally do not. I refer to the currently available collaboration-transparency systems as **conventional** because they use the same general implementation approach, which is the primary cause of the usability deficiencies found in these systems. In particular, conventional systems do not adequately support the following groupware principles (Chapter 4 contains a thorough critique of conventional collaboration-transparency systems).

Concurrent work Conventional collaboration-transparency systems do not allow simultaneous input from more than one person.

Independent View In conventional collaboration-transparency systems, all participants see exactly the same view at the same time.

Group awareness Conventional collaboration-transparency systems provide only limited information about the locations and activities of other collaborators.

Network usage Conventional collaboration-transparency systems generally require higher network bandwidth than collaboration-aware applications.

Figure 1.1 illustrates the dimensions along which transparency and awareness generally differ. It is important to note that while conventional collaboration-transparency systems are clustered at

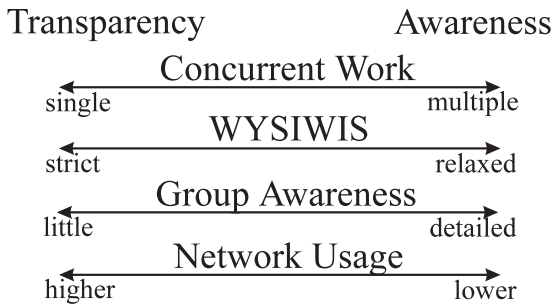


Figure 1.1: Four dimensions along which conventional collaboration-transparency systems and collaboration-aware applications differ.

the left end of each dimension, specific collaboration-aware applications appear anywhere on the scales. A particular collaboration-aware application may be no better than conventional collaboration transparency in one or all dimensions.

Despite the problems previously enumerated, collaboration transparency remains attractive because of the low development cost of sharing legacy applications. When a sufficiently powerful single-user application is suitable for a collaborative task, it makes sense to consider collaboration transparency. In fact, given the lower application development cost of sharing via collaboration transparency, is there any reason to develop a collaboration-aware application for such tasks? This is the motivating question of this dissertation.

1.2.2 Research Hypothesis

The primary goal of this work is to maintain the benefits of collaboration transparency while relieving some of its disadvantages, blurring the distinction between collaboration-transparent and collaboration-aware systems. The claim explored in this thesis is that a suitably implemented collaboration-transparency system can promote a single-user, collaboration-unaware application to the level of support for collaboration previously provided only in applications specifically designed for collaborative use. My hypothesis is: *Many of the features of collaboration-aware applications can be* transparently *provided in shared single-user applications.*

1.2.3 Approach

To examine the hypothesis, this work followed the familiar analyze-design-build-evaluate cycle. In the first stage of the work, I examined the different implementation approaches and levels of collaboration support provided by collaboration transparency versus collaboration awareness. Several questions were examined in the analysis. What are the usability needs of collaboration? How can these needs be supported by software? What implementation approaches are best to meet these needs? What are the causes of the inflexibility of conventional collaboration-transparency systems? What are the platform requirements to build a flexible collaboration-transparency system?

Understanding the implementation differences between conventional collaboration-transparency systems and collaboration-aware applications allowed me to design an alternative implementation approach to collaboration transparency that provides many features previously seen only in collaboration-aware applications. The approach is based on an object-oriented replicated architecture where selected single-user interface objects are dynamically replaced by multi-user extensions (Chapter 5). The replacement occurs at run time and is transparent to the single-user application and its developers. I implemented this approach in a new collaboration-transparency system called Flexible JAMM (Java Applets Made Multiuser).

I conducted an empirical study that evaluated the effectiveness of the prototype system versus a representative conventional collaboration-transparency system for performing tightly and loosely coupled tasks. The results of this evaluation confirm the hypothesis that many of the benefits currently seen in collaboration-aware applications can be provided in a collaboration-transparency system. I also conducted an informal evaluation from a software developer's perspective.

1.3 Summary of Contributions

This dissertation makes several contributions. The first is an analysis of conventional collaboration-transparency systems and a critique of their deficiencies (Chapter 4).

Another result is the design and implementation of a replicated-architecture collaboration-transparency system (Section 3.2.2). Replication uses network resources more efficiently and has lower application response time than the centralized approach used by conventional systems. These advantages are important for Internet-wide collaborations. The replicated approach to collaboration transparency has been explored previously, but the technology of the time made it impractical, and

researchers recommended against replication for collaboration transparency (Lauwers *et al.*, 1990; Minenko, 1998). This dissertation demonstrates that a replicated architecture for collaboration transparency is viable with current technology. This dissertation addresses known issues with replication and presents a new approach to transparent sharing of system resources, such as files, system time and network connections (Section 3.2.6).

The most significant result of this dissertation has been to blur the distinction between the capabilities of collaboration transparency and awareness. This dissertation presents a general design for a collaboration-transparency system that includes flexible support of multiple styles of collaboration (Chapter 5). Additionally, this dissertation identifies platform requirements to implement the approach (Section 5.2.3).

This dissertation makes several secondary contributions. A prototype flexible collaboration-transparency system, Flexible JAMM (Section 5.3), is freely available on the Internet (Begole, 1997). The development of Flexible JAMM included many reusable multi-user software components: telepointers, collaborator information and configuration components, a multi-user scrollable panel (Section 5.3.5), and a multi-user concurrent text editor (Section 5.3.7). These components are also used in a collaboration-aware interactive visualization system, called Sieve (Isenhour *et al.*, 1997). The development of these components also contributed to the design and development of a collaborative learning environment, the LiNC (Learning in Network Communities) virtual school.

The evaluation of flexible versus conventional collaboration transparency (Section 6.1) corroborates the results of previous studies of group awareness interface techniques (Gutwin *et al.*, 1996b; Gutwin & Greenberg, 1998). The evaluation also substantiates previous claims that implicitly passing control among collaborators is preferred over explicit control passing in some types of collaboration (Crowley *et al.*, 1990; Lauwers, 1990; McKinlay *et al.*, 1994) (Section 6.1.3).

1.4 Overview of Dissertation

This dissertation is organized as follows. Chapter 2 describes the foundation upon which this dissertation is built, reviewing the current state of the art in synchronous collaborative software, and summarizing the relevant results of other researchers in CSCW. In Chapter 3, I discuss implementation approaches and architectures to groupware applications. That chapter discuss issues and alternative solutions for each type of architecture, and presents a new approach to transparent sharing of system

resources in a replicated architecture. Chapter 4 critiques conventional collaboration-transparency systems, and describes the implementation causes of their deficiencies. The bulk this dissertation's contributions appears in Chapter 5, which presents an alternate implementation approach that alleviates many of the problems in conventional systems. That chapter also describes the incorporation of this approach into a new collaboration-transparency system called Flexible JAMM. Chapter 6 describes the methodology and results of a usability evaluation comparing Flexible JAMM to a representative conventional collaboration-transparency system, Microsoft NetMeeting. That chapter also presents an informal evaluation from the perspective of developers using Flexible JAMM as a groupware toolkit. Finally, Chapter 7 draws conclusions from the work described in previous chapters and discusses the implications of this work for future collaboration-transparency and collaboration-aware software.

Chapter 2

Research Foundations

This chapter provides context for the research reported in this dissertation by reviewing relevant prior research and the current state of the art in synchronous collaborative software.

2.1 Overview of Computer-Supported Cooperative Work

Many researchers in the 1960s, notably Douglas Engelbart (1963), saw that network technology provided an opportunity to change the way people work together. He and other researchers noticed the potential for more direct interaction among computer-based workers and called for the study of computer-supported cooperative work (CSCW) (Ellis *et al.*, 1991; Grudin, 1994).

Because collaboration and computers are prevalent in many areas of society, CSCW is a broad field. Collaborative applications include, but are not limited to, communications, group calendars, resource schedulers, collaborative learning, joint editing of documents, interactive online sales, joint exploration of data, simulations, and visualizations, as well as multi-player interactive games. Examples of all of these exist, and new products are continually emerging.

Jonathan Grudin (1994), describing the history of CSCW, categorized collaborative systems along dimensions of time and space, as seen in Table 2.1. Each grid in the matrix represents a subcategory of CSCW. There are three categories of *place,* which refers to the location of participants engaged in a computer-supported collaboration: (1) each collaborator may be in the same place, such as an electronic meeting room; (2) each collaborator may be in a different but predictable place, such as at their own desktop machine during a video teleconference; or (3) each may be in a different and

Table 2.1: Grudin's categories of collaboration according to time and place.

		Time		
		Same	Different and predictable	Different and unpredictable
Place	Same	Meeting facilitation	Work shifts	Team Rooms
	Different and predictable	Tele/video/ desktop conferencing	Electronic mail	Collaborative writing
	Different and unpredictable	Interactive multicast seminars	Computer bulletin boards	Workflow

not predictable place, such as participants in a computer bulletin board discussion (e.g., Usenet). The *times* during which activity occurs in the collaboration also differentiates CSCW systems into three categories: (1) the collaboration may exist in a single unbroken period, such as a meeting; (2) activity may occur at different but predictable times, such as sending an e-mail that is expected to be read within a certain period (typically a day); or (3) activity may occur at different and not predictable times, such as coauthors working on independent parts of an article at various times.

This dissertation primarily explores software in support of same-time, different-and-predictable place collaborations, or **desktop conferencing** systems. In particular, this dissertation discusses application-sharing systems that provide the real-time shared use of off-the-shelf, legacy, single-user applications. The dissertation addresses both technical and human factors related to the development and use of such systems. The next section discusses some of the human factors relevant to desktop conferencing systems.

2.2 Groupware Principles

Researchers in CSCW have identified many principles that should be incorporated in effective synchronous collaborative software. This section reviews some of the fundamental groupware principles.

2.2.1 Task Coupling

Dewan and Choudhary (1991a) refer to the degree to which collaborators work closely together versus independently as **tight** versus **loose coupling**. They state that a "collaborative system must support flexible coupling." That is, the system should allow collaborators to vary their collaboration style from tight to loose and back.

In a study of group writing, Posner and Baecker (1993) identified a wide variety of collaboration styles used by writing teams. They observed collaborations ranging from tightly to loosely coupled. They concluded that technology in support of collaboration must be "flexible and permissive, allowing groups to change strategies and processes at any time during the project with minimal distraction."

2.2.2 What You See Is What I See

A principle which supports tightly coupled collaboration is **strict What You See Is What I See** (WYSIWIS), which Stefik *et al.* (1987) defined as a mode of synchronous collaboration where all participants see exactly the same view at approximately the same time. Figure 2.1 shows two collaborators working with drawing software in a strict WYSIWIS mode.

Stefik *et al.* found that strict WYSIWIS is often too restrictive because it only supports tightly coupled collaboration. They propose four dimensions along which WYSIWIS may be relaxed:

Space The locations of all shared items are the same for all collaborators under strict WYSIWIS. However, it is sometimes useful to allow collaborators to maintain independent views of shared items.

Time Under strict WYSIWIS, changes in the shared information are seen by all collaborators nearly simultaneously. Relaxing this constraint allows the system to send updates at potentially more optimal intervals, perhaps different intervals for each collaborator. Fully relaxing this constraint leads to asynchronous applications, such as email.

Population Under strict WYSIWIS, the entire group sees the shared view. It is sometimes useful to allow subgroups to split off.

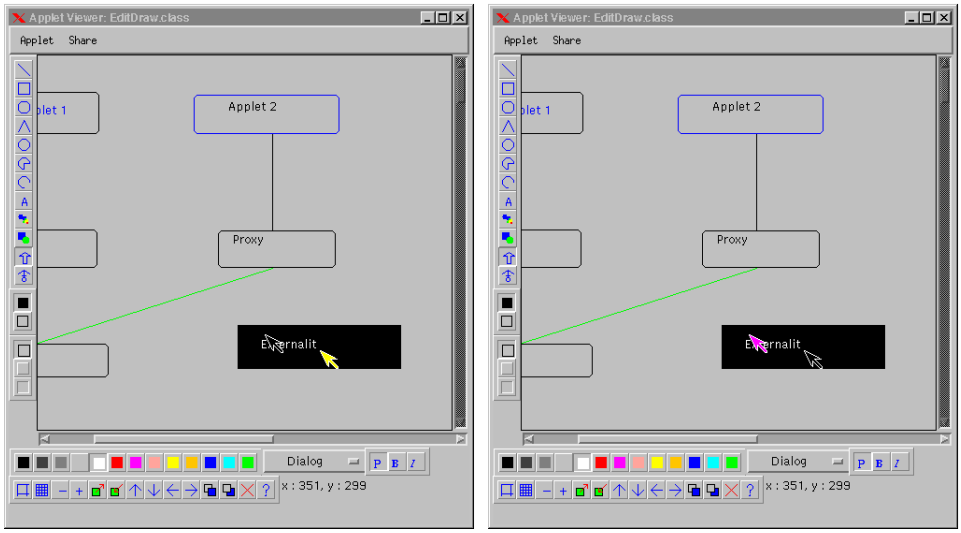


Figure 2.1: Strict What You See Is What I See (WYSIWIS) is a mode of synchronous collaboration where all participants see exactly the same view. Here, two collaborators (one in the left window, the other in the right window) see exactly the same view of a shared drawing.

Congruence Everyone sees the same representation of the shared items under strict WYSIWIS. Relaxing this constraint provides each collaboration with a different representation, perhaps depending on each one's role in the collaboration.

Relaxing WYSIWIS along these four dimensions provides support for both tightly and loosely coupled collaborations. However, the advantage of strict WYSIWIS is that the collaboration is grounded by the shared view; all collaborators know that they all see the same thing. Stefik *et al.* do not provide guidelines that constrain the relaxation of WYSIWIS so that the software still effectively supports collaboration. In what ways can WYSIWIS be relaxed before collaborators lose their sense of working together? The next section discusses constraints on the relaxation of WYSIWIS.

2.2.3 What You See Is What I Think You See

Randall Smith (1992) provides a general constraint on the relaxation of WYSIWIS that maintains the benefits of strict WYSIWIS. Smith argues that strict WYSIWIS is useful because collaborators know what each other are aware of. Smith points out that real world collaborations, such as face-to-face conversations, are effective without operating under strict WYSIWIS because collaborators have

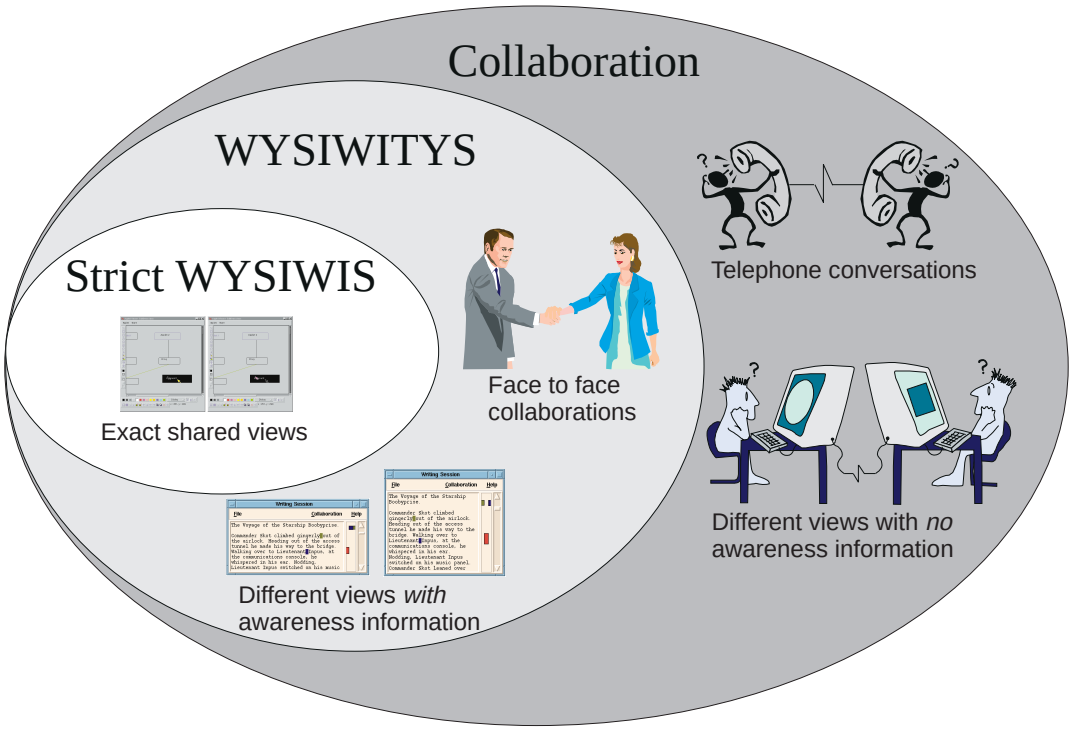


Figure 2.2: Collaborations occur with varying degrees of information about what each collaborator can see. In the inner category, strict What You See Is What I See (WYSIWIS), computer-based collaborators share the exact same view. In the middle category, What You See Is What I Think You See (WYSIWITYS), real-world and computer-based collaborators may not have the same exact view, but are aware of what each can see. There are some collaborations, represented by the outer category, in which collaborators do not know what others see. (based on a diagram by Randall Smith (1992))

a sense of what each other can see. That is, person B knows what person A can see. Smith refers to this principle as **What You See Is What I Think You See** (WYSIWITYS). The WYSIWITYS principle requires that each collaborator be able to construct a model of each other's experience.

Strict WYSIWIS conforms to WYSIWITYS and is thus contained in WYSIWITYS, as illustrated in Figure 2.2. WYSIWITYS provides a useful guideline as to how far WYSIWIS may be relaxed and still maintain meaningful support of collaboration. By providing information that achieves WYSIWITYS, the collaboration is grounded in the shared data as equally as under strict WYSIWIS. The next section describes the type of information that a groupware application must provide to support WYSIWITYS.

2.2.4 Group Awareness

When people work together in the same physical location, they are generally aware of each other's locations and actions. However, when the collaboration is distributed, much of this awareness information is lost. Without this information, groups have difficulty coordinating access control, work-flow, and communication. This section describes research into providing distributed groups with the information necessary to maintain awareness.

Greenberg, Gutwin, Roseman *et al.* from the University of Calgary's Grouplab project have contributed significantly to understanding awareness in groupware. They describe the following aspects of group awareness (Gutwin *et al.*, 1996a; Gutwin, 1997).

- **Presence awareness**[1] is the sense of the presence and general activity of group members. Who is available? Where are they now?
- **Engagement awareness**[2] is the knowledge of a collaborator's level of engagement in the activity, including another person's amount of interest, attention, and emotional state. In same-place collaborations, visual cues, such as gestures, eye contact, and body language, support this form of awareness. Such information may also be conveyed audibly through tone of voice, periodic acknowledgments (or lack thereof), and non-verbal sounds (Short *et al.*, 1993).
- **Structural Awareness** is the knowledge of participants' roles, positions on issues, status, and group processes. Effective collaboration depends on collaborators' implicit understanding of such information (Mintzberg, 1993; Viller, 1993).
- **Workspace awareness** is defined by Gutwin *et al.* (1996a) as the up-to-the-minute knowledge of other participants' interactions and locations within a workspace. In real-world collaborations, a workspace is one physical location, whereas in computer-based collaborations, a workspace is some shared data that are accessed via software and the collaborators may be physically co-located or distributed.

Table 2.2 lists the elements of workspace awareness and the questions that users may have with respect to each. Gutwin *et al.* propose that developers use this framework to design support for workspace awareness in collaborative software. This framework primarily relates to providing

[1] Gutwin *et al.* (1996a) label this category *informal* awareness.
[2] Gutwin *et al.* (1996a) refer to this category as *social* awareness.

Table 2.2: Elements of workspace awareness. Gutwin *et al.*'s (1996a) framework of workspace awareness information for designing collaborative software.

Element	Relevant Questions
1. Identity	Who is participating in the activity?
2. Location	Where are they?
3. Activity Level	Are they active in the workspace? How fast are they working?
4. Actions	What are they doing? What are their current activities and tasks?
5. Intentions	What are they going to do? Where are they going to be?
6. Changes	What changes are they making? Where are changes being made?
7. Objects	What objects are they using?
8. Extents	What can they see?
9. Abilities	What can they do?
10. Sphere of Influence	Where can they have effects?
11. Expectations	What do they need me to do next?

information about *individual* work within a group. Another aspect of workspace awareness, described by Ackerman and Starr (1996), is the aggregation of individual activity, called **social activity**. Social activity information is used by groups in several ways. First, it helps individuals focus their own activity. Second, knowing whether or not others are working can have a profound effect on an individual's motivation. Finally, acceptance of groupware systems often depends on maintaining a certain threshold of social activity (Grudin, 1988).

2.3 Interface Support of Workspace Awareness

Several techniques are used in collaborative systems to provide information to support each of the aspects of awareness. Media spaces provide presence awareness information through video or periodically updated still images of remote locations associated with distributed group members (Bly *et al.*, 1993; Harrison & Minneman, 1993; Mantei *et al.*, 1993). Engagement awareness can be provided with audio and video communications channels (Buxton, 1992). Researchers within the CSCW community have investigated providing support for structural awareness using electronic meeting rooms (Mantei, 1993), decision and communication support systems (Pinsonneault & Kraemer, 1993), and policy and role mechanisms within collaborative software (Edwards, 1996). Ackerman and Starr

Figure 2.3: A telepointer indicates a remote participant's cursor location in the workspace. Typically, each user has a unique color. Shapes and names may also be used to distinguish users.

Figure 2.4: (a) Simple telepointer. (b) Telepointer conveying action information.

(Ackerman & Starr, 1996) describe social activity indicators which provide visualizations of the level of group-wide activity within a collaborative system. The type of awareness with which this dissertation is primarily concerned, however, is workspace awareness. The remainder of this section describes software support of workspace awareness.

2.3.1 Telepointers

A widely used workspace awareness interface element is the **telepointer**, an indication of a remote participant's screen pointer. A common variant of the telepointer is the **telecaret,** which represents a remote user's text insertion cursor, or caret. Examples of telepointers are shown in Figure 2.3.

In its simplest form, a telepointer provides fundamental awareness information: identity, and location. Additionally, a telepointer can provide action information, such as mouse button presses and releases. Greenberg *et al.* (1996) extended the notion of a telepointer to provide action information . They described a **semantic telepointer,** which changes appearance based on the actions of remote participants. Figure 2.4 shows (a) a typical and (b) an action-augmented semantic telepointer. Tang (1991) found that real-world collaborators frequently use hand gestures to communicate. Gestures include both movement and action. Thus, action telepointers enhance collaborators' ability to gesture.

By changing its shape, color, or label, a semantic telepointer can convey much of the workspace awareness information listed in Table 2.2: identity, location, activity level, actions, changes, and abilities.

A telepointer provides a visual representation of a remote participant, but is only useful when the telepointer is visible. In a location-relaxed WYSIWIS collaboration, not all of the remote participants' telepointers are necessarily visible at any given time. One means of providing action information when telepointers are not visible is via auditory cues. In addition to displaying the word "click" on the telepointer seen in Figure 2.4, a system could play a sound that represents a mouse click. Gaver (1993) demonstrated that activity awareness can be enhanced by auditory cues. However, the auditory cues used by Gaver were specific to the application he tested (a simulation of a bottling plant). Generic actions like mouse clicks have context-dependent semantics. For example, a click executed within an on-screen button causes the button's function to execute, while a click in a plain panel typically has no effect. A collaborator cannot infer that the application state will change from simply hearing a click. Therefore, it is not clear that the benefits of the task-specific sounds used in Gaver's study will extend to generic (yet context-dependent) actions, like mouse clicks.

2.3.2 Multi-user Scrollbars and Radar Views

One aspect of workspace awareness not provided by telepointers is the extent of each collaborator's view of the shared workspace. When strict WYSIWIS is employed, collaborators' extents are obvious, since they are the same for each. However, additional information must be provided when an application relaxes WYSIWIS so that each collaborator can have an independent view of the shared data.

One interface mechanism used to indicate view extents is a multi-user scrollbar. Many times, data are too large to display completely, and only a portion can be shown at one time. A scrollbar is a widely used software interface item with which a user can navigate the viewed portion of too-large data. The user slides a "thumb" button to move his or her position. In many implementations, the size of the "thumb" button with respect to the total scrollbar size is proportional to the size of the view with respect to the whole data. Thus, a scrollbar indicates the position and extents of the user's view. A multi-user scrollbar extends this by displaying the position and extents of each participant in a collaboration (Baecker *et al.*, 1993). Figure 2.5 shows an example of a multi-user scrollbar for two participants in a collaborative text editor called Calliope by Alex Mitchell (1996). On the right,

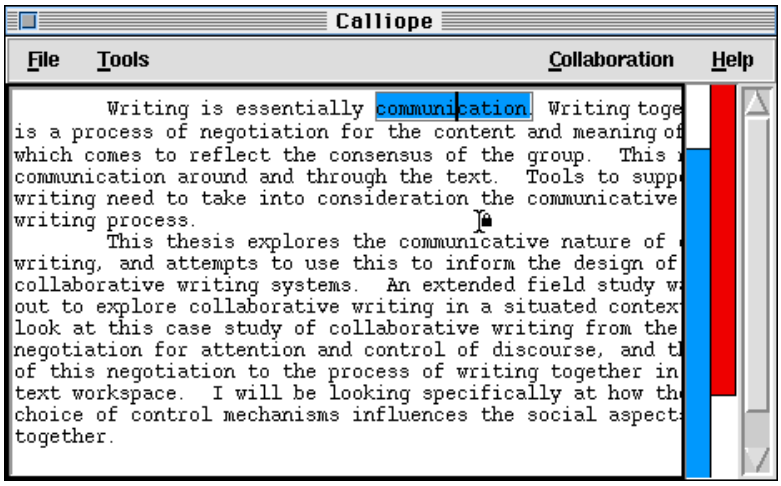


Figure 2.5: The multi-user scrollbar used in a collaborative text editor, called Calliope (Mitchell, 1996).

a conventional scrollbar is used to move the local user's position. Next to that, colored rectangles indicate each participant's position and extents. The local participant's view extends to the top of the document (indicated by the rightmost of the two rectangles), while the remote participant's view extends to the bottom (indicated by the leftmost of the two rectangles).

A problem with multi-user scrollbars, such as the one shown in Figure 2.5, is that they cannot effectively display more than a few participants' independent locations. The required display space increases with each additional collaborator. In the Shared Alternate Reality Kit (SharedARK), Smith *et al.* (1989) introduced an alternative to multi-user scrollbars, called a **multi-user radar view**. A radar view is a miniature overview of the shared workspace with a unique indication (typically, a rectangle) of each collaborator's view of the workspace. By allowing the view indicators to overlap, the multi-user radar view requires a fixed amount of display space regardless of the number of collaborators. In many implementations, the radar view updates the miniature display whenever the workspace changes. Thus, participants can see when and where changes are being made outside of their full-size view. Figure 2.6 shows three participants using a radar view to work in a shared text editor. Gutwin and Greenberg (1998) have demonstrated that multi-user radar views enhance the usability of collaborative applications for some tasks.

The combined use of telepointers and multi-user radar views supports nine of the eleven workspace awareness elements listed in Table 2.2. The two remaining elements, intentions and ex-

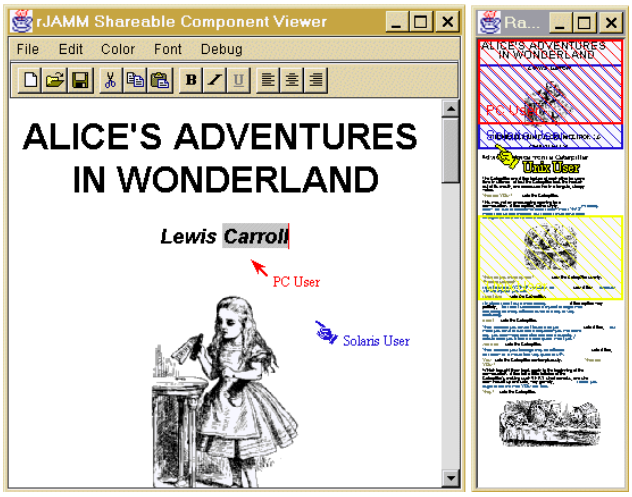


Figure 2.6: A shared text editor employing a multi-user radar view. The *radar view* window on the right displays a miniature overview of the shared document. A uniquely colored and shaded rectangle indicates each participant's scroll position. Three participant positions are shown here: (1) *PC User* and (2) *Solaris User* overlap near the top while (3) *Unix User* is near the center.

pectations, can be communicated by collaborators themselves via separate communications channels (i.e., video or audio channels).

The remainder of this chapter reviews collaborative software systems and their support of awareness needs.

2.4 Synchronous Collaborative Systems

Synchronous collaborative applications are customarily placed in two categories. The first, **collaboration awareness**, includes applications specifically designed to support cooperative work. The application's developers were *aware* that their application would be used collaboratively. The second category is referred to as **collaboration transparency** because the sharing is provided by a mechanism that is unknown, or *transparent,* to the application and its developers (Begole *et al.*, 1997a). This section describes notable research in collaboration-aware applications and toolkits, and collaboration-transparency systems.

2.4.1 Groupware Toolkits

Synchronous collaborative application development has requirements beyond those of single-user application development, giving groupware applications a higher development cost than single-user applications. Patterson (1991) identified the following three development requirements of synchronous groupware that single-user application development does not necessarily require.

1. **Concurrency:** As multiple users simultaneously interact within a collaborative application, concurrent inputs may conflict. The collaborative application must deal with conflicts from concurrent inputs. In a single-user application, the application receives a stream of inputs from a single source.

2. **Separation of interface and data:** To allow collaborators to access data from different locations, the application's data must be separated from its interface. Separating interface and data is useful to single-user applications, too, but is not required.

3. **User roles:** In a typical collaboration, each coworker performs a different subtask. Ideally, a collaborative application will support each role. In a single-user application, the application only needs to support the role of the single user.

The developer of a real-time collaborative application may address these issues *ad hoc,* or by employing a groupware toolkit. Groupware toolkits attempt to facilitate collaborative application development by providing implementations of some common problems. The goal of groupware toolkits is to facilitate the development of a collaboration-aware application so that it is only "slightly more difficult" to develop than an otherwise functionally equivalent single-user application (Roseman & Greenberg, 1996a; Lee *et al.*, 1996). This section first presents features found in most groupware toolkits. The second part of this section lists groupware toolkits and compares their support for common groupware requirements.

Groupware Toolkit Capabilities

Greenberg and Roseman (1998) state that groupware toolkits should provide four major components: communication abstractions, distribution architecture, session management, and multi-user interface components.

Communication Abstractions Synchronous groupware applications are distributed. Thus, groupware developers must use some mechanism to communicate among the distributed components. Groupware toolkits provide abstractions to facilitate such communication. For example, most groupware toolkits provide a method to send a message to the distributed components of the application.

Software Distribution Architecture Distributed software architectures fall in a range from **centralized,** where the shared application is maintained in one physical location, to **replicated,** where the shared application is copied to each collaborator (Greenberg & Roseman, 1998; Crowley *et al.*, 1990). Groupware architectures are discussed in greater detail in Chapter 3. A groupware toolkit provides an implementation of one or more distribution architectures and provides the developer with mechanisms necessary for maintaining data consistency within the selected architecture.

Multi-user Interface Components Section 2.3 described several user interface techniques to provide workspace awareness information in collaborative software. Such general awareness components should be included in a toolkit. There may also be application-specific awareness needs. Therefore, the toolkit should allow the development of application-specific multi-user interface components.

Session Management A **session** is the collection of synchronously collaborating users and the software employed in the collaboration. A groupware toolkit must provide some means by which users may start, find, join and leave a collaborative session (Edwards, 1994; Greenberg & Roseman, 1998). Generally, groupware toolkits provide developers with facilities to create, and manage sessions. Some provide a user interface to allow users themselves to manage sessions.

Notable Groupware Toolkits

The following groupware toolkits are presented in approximately the chronological order in which they appeared in the research literature. These reviews only consider each toolkit's suitability to create real-time collaborative applications. Some of these systems provide further capabilities such as persistence (e.g., DistView/CBE), domain-specific support (e.g., Sieve's data flow and visualization support), and integration with third-party software (e.g., audio/video conferencing tools launched by Tango). These further capabilities should be considered when evaluating each toolkit's applicability to a particular collaborative need.

MMConf In 1990, Crowley and other researchers at Bolt, Beranek, and Newman, Inc. (BBN) reported the development of an X-based toolkit written in C called MMConf. MMConf uses a replicated architecture. It intercepts and distributes low-level user input events. MMConf provides a single shared telepointer that can be controlled by one user at a time. MMConf avoids concurrency conflicts by using **floor control**, as described in Section 3.2.3.

An interesting feature of MMConf is its support for accessing files in a replicated architecture. When replicas open a file, MMConf ensures that each replica has a copy of the same file. Most replicated systems do not address how to share system resources, such as files. Section 3.2.6 presents an approach to sharing system resources in addition to files, such as sockets and system time, under a replicated architecture.

Suite Dewan and Choudhary (Dewan & Choudhary, 1991b; Dewan & Choudhary, 1995) of Purdue University developed the Suite toolkit, which they first reported in 1991. Suite uses a replicated architecture and defines a set of multi-user programming primitives in C for updating replicas.

By following Suite's programming conventions and using Suite functions to modify data, the developer need not consider whether or not the application will later be used by a single person

or multiple people. This sort of collaborative application development is called **collaboration unaware**. Collaboration unawareness is similar to collaboration transparency. Under unawareness, the developer uses a particular library to develop the application, and the library handles the distribution architecture and other aspects of collaboration completely. Generally, the developer knows that using the library will allow the application to be shared, but is unaware of how that sharing occurs does not need to program support for the collaboration explicitly. In collaboration transparency, however, the underlying sharing mechanism is completely unknown, or transparent, to the application and its developer. That is, the developer created the application without any notion that it would be shared later. Traditionally, collaboration-transparency systems are able to share any application written to the standard Application Program Interface (API) of a particular operating/windowing system, such as the X Window System or Microsoft Windows.

Suite provides mechanisms by which the developer can override the default, collaboration-unaware behavior of an application. Suite provides functions to send messages to all or a subset of users in the current session, and to set and read shared environment properties. Dewan and Choudhary introduced the notion of task coupling (the degree to which collaborators work closely versus independently). A suite developer can specify the application's task coupling support by changing the granularity and timing of replica updates. For example, the developer of a text chat application may choose to send each character as soon as it is entered, or only a whole line at a time.

Rendezvous Rendezvous is a groupware toolkit and language developed by Hill *et al.* (1994) at Bellcore in the early 1990s. Rendezvous runs on the X platform and is implemented in Common Lisp.

Rendezvous is unique among groupware toolkits in that it uses a semi-replicated architecture, but the replicated processes all reside on a single host. Thus, Rendezvous applications are conceptually replicated, but physically centralized.

One of Rendezvous' contributions is the easy-to-use communication abstraction of its programming paradigm, called Abstraction-Link-View (ALV), which separates an application's data from its interface. A single instance of an "abstraction" contains the data that are shared among the collaborators. Replicated "views" display the shared data. Each view may display the data differently, allowing relaxed WYSIWIS among the collaborators. Each view is associated with an abstraction via a "link," which ensures that changes to the abstraction are reflected in the link's corresponding

view. That is, when the abstraction changes, each link updates its corresponding view. Conversely, when a user changes a view, the link updates the abstraction. The replicated views interact directly and exclusively with the centralized abstraction. Thus, a Rendezvous developer does not need to propagate changes explicitly.

GroupKit GroupKit (Roseman & Greenberg, 1996a) is a groupware toolkit developed by Greenberg, Roseman, *et al.* at the University of Calgary first reported in 1994. GroupKit, and applications developed with is, use Tcl/Tk, which runs on multiple platforms including Microsoft Windows, X and Macintosh. Thus, GroupKit applications have multi-platform capabilities similar to applications developed with the Java-based groupware toolkits, described later.

GroupKit applications are fully replicated. Only the "registrar," which maintains a list of active sessions, is centralized. The registrar provides session management capabilities. GroupKit provides three programming abstractions for communication among replicas. One, called multicast remote procedure call, allows a replica to invoke a procedure on remote replicas. Another communication abstraction is provided through the use of a shared "environment." All replicas see any change made to the state of the shared "environment." Finally, GroupKit provides mechanisms to multicast and receive messages.

The GroupKit team has pioneered research into multi-user interfaces. Thus, GroupKit includes several multi-user interface components, such as semantic telepointers, multi-user radar panes, and multi-user fish-eye lenses.

Clock Graham *et al.* (1996) at York University created the Clock synchronous groupware environment and ClockWorks application builder, which they initially reported in 1996. Clock uses a conceptually centralized model-view-control paradigm where the shared data are held in the centralized model, but updates are cached, pre-sent, or pre-fetched, essentially replicating the data. The MVC paradigm, like Rendezvous' ALV, allows relatively simple development. When any replicated view-controller changes the data in the centralized model, the other replicas are implicitly updated. Thus, the programmer is not concerned with the communications taking place among modules.

COAST In 1996, Schuckmann *et al.* (1996) at the German National Research Center for Information Technology described COAST (COoperative Application Systems Toolkit), a Smalltalk-based groupware toolkit. COAST employs a replicated architecture and a variant of the model-view-control

programming paradigm. Changes to shared data are made via COAST's transaction protocol and implicitly propagated to replicas. Thus, applications developed with COAST are collaboration unaware.

Prospero Also in 1996, Dourish (1996b; 1996a) described the Prospero toolkit, with which programmers can build collaborative applications using the Common Lisp Object System (CLOS). Prospero supports the creation of a variety of asynchronous to synchronous groupware applications. The developer can specify a range of distribution architectures, from nearly centralized to fully replicated. Whereas most toolkits emphasize preventing state divergence among replicas, Prospero does not attempt to prevent divergence, but provides a framework to merge divergent replicas. Developers must create application-specific procedures to merge divergent replicas and restore consistency. Prospero does not provide multi-user interface components.

Java-based Groupware Toolkits

Aside from any inherent programming features, Java is especially attractive for writing collaborative software since it simplifies distribution of software and largely eliminates issues with heterogeneous platforms. While these characteristics are useful for software in general, they are critical for collaborative software because each user within the collaboration must have a copy of the software. Accordingly, several Java-based groupware toolkits have been developed in recent years.

The DistView Toolkit and Collaboratory Builder's Environment The DistView Toolkit (Prakash & Shim, 1994) and Collaboratory Builder's Environment (CBE) (Lee *et al.*, 1996) are Java-based collaborative systems from Prakash *et al.* at the University of Michigan, described in 1996. CBE is essentially the session management interface for collaborative applications developed with the DistView toolkit. CBE uses the metaphor of a room to represent a session, similar to TeamRooms (Roseman & Greenberg, 1996b). All users in a room share the applications that the room contains. Applications may be removed from a shared room to be worked on privately and later returned to the room. CBE includes support of user roles and access privileges.

DistView provides a semi-replicated architecture, where the shared state of an application resides on a central server and the user interfaces are replicated. Similar to the approach of Rendezvous, when one user modifies the shared state, the replicated interfaces are updated.

Java Shared Data Toolkit The Java Shared Data Toolkit (JSDT) (Burridge, 1998), developed by Rich Burridge, was released by JavaSoft in August 1998, although beta versions were available in 1997. JSDT is a multi-point communications package for replicated collaborative systems. JSDT is based on the International Telecommunication Union (ITU) T.122 recommendation for Multipoint Communication Service, and roughly corresponds to the *presentation layer* of the Reference Model for Open Systems Interconnection (OSI). JSDT uses the notion of a **session** which contains one or more communication **channels**. Messages sent to a channel are distributed to all processes that are registered as listeners of the channel.

JSDT defines an application programming interface (API) which is implemented on top of different network transport protocols. For example, the JSDT distribution includes implementations based on standard sockets, Java Remote Method Invocation (RMI) (Wollrath *et al.*, 1996), and the Light-weight Reliable Multicast Protocol (LRMP) (Liao, 1998). This separation of API and implementation provides flexibility that allows the groupware developer to match the unique requirements of a particular distributed application with an optimal implementation.

Java Collaborative Environment Researchers at the National Institute of Standards and Technology (NIST) and Old Dominion University (ODU) have developed a Java-based groupware toolkit called the Java Collaborative Environment (JCE) (NIST, 1998), initially reported in 1996 (previously called the Java Collaborator's Toolkit (Abdel-Wahab *et al.*, 1996)). JCE uses a replicated architecture where user inputs are intercepted and broadcast to replicas, similar to MMConf.

To capture user inputs, JCE requires a developer to use JCE's interface library, called `collawt`, which contains a replacement for each interactor (e.g., button, checkbox, text area) found in the standard Java user interface library, the Abstract Window Toolkit (AWT). Typically, a single-user application can be converted to a JCE collaborative application by replacing the AWT package with `collawt` in the application source code. Since the developer must make this change to the application source code, and is therefore aware that the application will be used collaboratively, JCE is not a collaboration-transparency system.

JCE is easier to use than some groupware toolkits. The only requirement for a JCE-based collaborative application is that it use JCE's interface library. In contrast to some replicated groupware toolkits (e.g., GroupKit), the JCE developer is not required to determine when to send messages to replicas because all user inputs are automatically distributed. On the other hand, JCE's ap-

proach to event broadcasting does not allow the developer to optimize network usage and imposes strict WYSIWIS. Furthermore, JCE components do not show visual feedback to remotely generated events. That is, a screen button is not drawn depressed when it is remotely pressed. This is because JCE broadcasts events after the local replica of the interactor has acted on the event. In our own prior work (Begole *et al.*, 1997b), we described the reasons for this problem, which we called *post-peer* interception, and a solution, called *pre-peer* interception.

Sieve Sieve is a collaborative environment[3] developed by Isenhour, Begole, and Shaffer at Virginia Tech beginning in 1997 (Isenhour *et al.*, 1997; Isenhour, 1998). Sieve uses and extends a popular component architecture, JavaBeans, as a framework for developing synchronous collaborative software (Begole *et al.*, 1998a). JavaBeans specifies a change-notification mechanism, called **bound properties**, where an object notifies registered listeners whenever the value of a property is changed. Sieve listens for property changes from each "bean." When a change notification is generated, Sieve packages it in a message that is sent to the corresponding replicas. These messages are sent in the same order to each replica using JSDT. When such a message is received, the local replica of the changed component is found, and the changed property is set to the new value.

The developer decides which parts of the component state to expose via bound properties. This allows the developer some control over the degree of WYSIWIS under which the component will be shared. If all of the state is accessed via bound properties, then the component will be shared under strict WYSIWIS. Typically, however, only the essential data properties need to be bound. By not exposing interface data, such as scrollbar position, the collaborators may work under relaxed WYSIWIS.

Many useful collaboration-unaware components may be quickly developed and shared with the bound properties approach. For some components, however, property changes may not provide sufficiently fine granularity for efficient network usage. For those cases, a developer may create application-specific events that Sieve will propagate and apply to corresponding replicas. Creating application-specific events and handlers is equivalent to using multicast-send and receive procedures, as seen in GroupKit and Suite.

[3]Sieve was initially intended to support collaborative interactive visualization (CIV). Hence, the name "Sieve" is a pun on CIV and the filtering provided by Sieve's data-flow library.

Habanero Chabert *et al.* (1998) at the National Center for Supercomputing Applications (NCSA) described a Java-based groupware toolkit called Habanero in 1998, but a version was available in 1997. Habanero uses a replicated architecture.

Habanero's primary communication abstraction is to distribute *events* to replicas. By default, Habanero distributes semantic events, such as the *action* event that results from a user clicking on-screen button, rather than low-level user input events. A developer can choose to distribute low-level events, such as mouse events, or new application-specific event types. Thus, Habanero's event propagation is more flexible than that used by JCE. However, like JCE, Habanero uses post-peer interception (Begole *et al.*, 1997b) when low-level input events are distributed. Thus, user interface components in Habanero do not provide visual feedback to remotely generated inputs.

A developer can also create application-specific "events," which Habanero will propagate to replicas. As with Sieve, defining application-specific "events" and handlers is equivalent to creating a message protocol.

Tango Tango (Beca *et al.*, 1997) is a groupware toolkit by researchers at the Northeast Parallel Architectures Center at Syracuse University, initially reported in 1997. Tango-based applications can be developed in Java, JavaScript, C/C++, and other languages. Therefore, Tango is not strictly a Java groupware toolkit, like the others in this section, but it is largely based on the Java platform.

Tango's session-management facilities are launched as a Java applet via a web browser. In addition to providing a set of collaborative utility programs (e.g., text chat and shared white board), Tango launches helper applications that provide audio/video conferencing (Buena Vista) and application sharing (NetMeeting).

Tango-based applications use a replicated architecture. Like Habanero, the Tango communications layer distributes a message among replicas in the form of an "event." The application developer must explicitly send a message, or event, when a property of the application is changed. The developer must also implement routines that receive messages and act upon them.

Summary of Groupware Toolkits

The previous list of groupware toolkits is not exhaustive, but represents the primary variants. Each has varying degrees of support for the four components of a groupware toolkit enumerated at the beginning of this Section: architecture, communication abstraction, session management, and multi-

user interface components. Table 2.3 summarizes the features of each toolkit. Most of the toolkits employ some form of replicated architecture. In fact, the one exception, Rendezvous, replicates portions of the application, but they all reside on one central host. Various communication abstractions are employed: distributing low-level user input events (MMConf, JCE, and Habanero); collaboration-unaware development by implicitly propagating state changes to all replicas (Suite, Rendezvous, COAST, Clock, DistView, and Sieve); periodic data merging (Prospero); multicast remote procedure call (GroupKit); properties stored in a shared environment (Suite and GroupKit); and simple multicast message sending (Suite, GroupKit, JSDT, Sieve, Habanero, and Tango). It is possible to create relaxed WYSIWIS applications in all of the toolkits, except MMConf and JCE. However, some of the toolkits facilitate relaxed-WYSIWIS development more than others. As an indication of how difficult it would be to generate a relaxed-WYSIWIS application, the table lists the WYSIWIS mode seen in the toolkit's example applications. Generally, groupware toolkits provide developers with facilities to create and manage sessions. Some provide a user interface that allows users themselves to control sessions. Only five of the thirteen toolkits in this survey provide even the most rudimentary awareness information with a telepointer. In all but MMConf and JCE, an application developer could create application-specific awareness components.

We turn next to the other category of synchronous collaborative software, collaboration transparency.

2.4.2 Collaboration-Transparency Systems

The earliest synchronous collaborative systems were collaboration-transparency systems. In 1968, Douglas Engelbart *et al.* developed NLS (oN Line System) (Engelbart, 1975). NLS made several contributions to the computing field, including the mouse input device and a hypertext-like journal. Additionally, with respect to collaboration, NLS provided a facility which allowed users to collaborate by sharing the view of their entire screen, called Shared-Screen. One user at a time could then control the screen. Generally, applications in NLS's time used the entire screen, and this type of collaboration-transparency system is referred to as **screen sharing**.

In the late 1980s, Lantz, Lauwers, *et al.* (Lauwers & Lantz, 1990; Lauwers *et al.*, 1990) advanced application sharing to the granularity of individual windows, which they referred to as **window sharing**. They introduced the term "collaboration transparency" to encompass both screen sharing and window sharing. Since then, many collaboration-transparency systems have been implemented.

Table 2.3: Feature comparison of groupware toolkits.

Toolkit	**Architecture**	**Communication Abstractions**	**WYSIWIS in example applications**	**Session Interface**	**Included multi-user components**
MMConf	Repl.	Input event distribution	Strict	No	Single shared pointer
Suite	Repl.	Implicit change propagation, multicast send, shared environment, coupling control	Relaxed	No	
Rendezvous	Centr.	Implicit change propagation	Relaxed	No	Telepointer
GroupKit	Repl.	Multicast send, Multicast remote procedure call, shared environment	Relaxed	Yes	Telepointer, radar view, fish-eye view, others
Clock	Semi-Repl.	Implicit change propagation	Relaxed	No	Telepointer
COAST	Repl.	Implicit change propagation	Relaxed	No	
Prospero	Semi-Repl.	Periodic data merging	Relaxed	No	
DistView/CBE	Semi-Repl.	Implicit change propagation	Strict	Yes (CBE)	
JSDT	Repl.	Multicast send	Relaxed	No	
JCE	Repl.	Input event distribution	Strict	No	
Sieve	Repl.	Implicit change propagation	Relaxed	Yes	Telepointer, radar view
Habanero	Repl.	Multicast send	Strict	Yes	
Tango	Repl.	Multicast send	Strict	Yes	

Several collaboration-transparency systems are described below according to the application platform they support.

X Window-based

Window sharing systems were facilitated by the introduction of the X Window System in 1987 (Gettys *et al.*, 1990), which defines a graphics protocol that separates an application's display from its process. An application's process, called an X client, sends graphics requests to the display server, called an X server. The client and server need not reside on the same machine, a capability not found in other widely used windowing systems, such as Apple's MacOS and Microsoft Windows. X's separation of display and process yields a natural approach to implementing collaboration transparency: intercepting and distributing an X client's display requests to multiple X servers. This centralized approach, called **display broadcasting**, is the most common implementation approach to collaboration transparency (distribution architectures are reviewed in more detail in Chapter 3). Applications shared under this approach use a strict WYSIWIS mode of sharing because each collaborator sees the same graphical display of the application.

The earliest window-sharing collaboration-transparency system, VConf (Lauwers, 1990), was developed on TheWA, a precursor of the X Window System, which similarly separates display and process. There were two versions of VConf, one centralized and the other replicated. X has since supplanted TheWA, and several X-based collaboration-transparency systems now exist, such as Dialogo (Lauwers, 1990), XTV (Abdel-Wahab & Feit, 1991), VisualTek's X/TeleScreen (X/TeleScreen, 1998), Hewlett Packard's SharedX (Garfinkel *et al.*, 1994) and Sun Microsystem's ShowMe SharedApp (Sun Microsystems, Inc., 1998). See Minenko (1996) for a more complete survey of X-based collaboration-transparency systems.

Windows-based

Commercial collaboration-transparency systems exist for Microsoft Windows, notably Microsoft's NetMeeting (Microsoft Corp., 1998) and Intel's ProShare (Proshare, 1998).

NetMeeting conforms to the ITU T.128 recommendation for Multipoint Application Sharing, which was proposed by Microsoft and PictureTel. T.128 defines a graphics protocol that is conceptually similar to the X protocol. Thus, T.128 facilitates the display-broadcasting described previously for X-based collaboration-transparency systems. Other products that conform to T.128,

and can therefore receive each other's display, include Data Connection's DC-Share (Downes, 1997), DC-WebShare (Data Connection, 1998), and PictureTel's LiveShare Plus (PictureTel Corporation, 1997). T.128-compliant clients have been implemented on X, Windows, and Macintosh, but T.128 application-sharing servers currently exist only on Windows. Like the X-based display-broadcasting systems described previously, NetMeeting and other Collaboration-transparency systems that use T.128 can only share applications in strict WYSIWIS mode. Strict WYSIWIS and other limitations of the conventionally implemented collaboration-transparency systems will be discussed in greater detail in Chapter 4.

Macintosh

Netopia distributes separate Windows- and Macintosh-based versions of its remote control system, Timbuktu Pro (Timbuktu Pro, 1998). In addition to remote control, Netopia recently announced that Timbuktu Pro allows macintosh users to share NetMeeting-hosted applications, via T.128 (Loney, 1998).

Wolf *et al.* (1996) at the University of Ulm, Germany described issues and approaches to application sharing among heterogeneous platforms. The prototype systems they report allow a Macintosh-based application to be shared among remote Macintosh and X clients.

Another Macintosh-based application-sharing system was RTZ Software's The Virtual Meeting (TVM) (Snyder, 1996), which no longer appears to be available. This was the only commercially available application-sharing system that used a replicated, event-broadcasting architecture (See Section 3.2.2 for more details of this approach).

2.5 Summary

This chapter reviewed relevant prior research and the current state of the art in synchronous collaborative software.

Researchers in the 1960s saw that networking technology provided an opportunity to change the way people worked by allowing more direct interaction among computer-based workers. The field called computer-supported cooperative work (CSCW) studies the wide spectrum of collaborative computing. This dissertation is concerned with the subcategory of CSCW called **desktop conferencing** in Grudin's matrix (1994) (see Table 2.1).

Researchers have identified several principles that effective collaborative software should incorporate. The degree to which collaborators work closely together versus independently is referred to as **tight** versus **loose coupling**. Synchronous collaborative software should allow the collaboration to flow between periods of tight and loose coupling. **Strict What You See Is What I See** (WYSIWIS) describes a mode of collaboration where all participants see exactly the same view at approximately the same time. Strict WYSIWIS is useful in tightly coupled collaborations, but is often too restrictive. Therefore, WYSIWIS may be relaxed in four dimensions: (1) space (participants may have different locations in the shared workspace), (2) time (changes may not be immediately propagated), (3) population (subgroups may share different views), and (4) congruence (participants may see different representations of the shared data).

The principle called **What You See Is What I Think You See** (WYSIWITYS) requires that participants be able to know what each other can see, although the views may be different. WYSIWITYS provides a constraint on how far WYSIWIS may be relaxed and yet maintain effective support for the collaboration (see Figure 2.2). WYSIWITYS is supported by software that provides information about **workspace awareness**, which is the up-to-the-minute knowledge of other participants' interactions and locations within a workspace. Gutwin *et al.* (1996a) propose a framework of eleven workspace awareness elements: identity, location, activity level, actions, intentions, changes, objects, extents, abilities, sphere of influence, and expectations (see Table 2.2).

Several user-interface techniques exist to convey workspace awareness information. A widely used technique is a **telepointer**, an indication of a remote participant's screen pointer. Extensions to the basic telepointer concept are **telecarets**, which represent remote users' text insertion carets, and **semantic telepointers**, which dynamically change appearance based on the actions of remote participants.

A single-user scrollbar allows a user to position a viewport into data that are too large to display all at once. A multi-user scrollbar extends this by also displaying the viewport positions of other collaborators side by side. A multi-user **radar view** extends this further by overlapping remote participants' viewports in a miniature view of the too-large data. The combined use of telepointers and multi-user radar views supports nine of the eleven workspace awareness elements, leaving intentions and expectations to be communicated directly among participants.

The chapter next examined the state of the art in synchronous collaborative software emphasizing the extent to which the previously discussed groupware principles are supported. Synchronous

collaborative applications are customarily placed in two categories. The first, **collaboration awareness**, includes applications specifically designed to support cooperative work. The second category is referred to as **collaboration transparency** because the sharing is provided by a mechanism that is unknown, or *transparent,* to the application and its developers.

Synchronous collaborative application development has a higher cost than single-user application development. Three factors contribute to the cost of synchronous collaborative application development (Patterson, 1991): (1) concurrency, (2) interface and data separation, and (3) user roles. These factors may also arise in a single-user application, but they cannot be avoided in collaborative systems. Groupware toolkits attempt to facilitate collaborative application development by providing implementations of at least some of the following common requirements: communication abstractions, distribution architecture, multi-user interface components, and session management. Several groupware toolkits were surveyed, and Table 2.3 compares their features.

The chapter next reviewed collaboration-transparency systems for each of the three major computing platforms: (1) the X Window System, (2) Microsoft Windows, and (3) Macintosh. All but two of the known collaboration-transparency systems use a centralized display broadcasting approach, where the graphical output of a single instance of the shared application is multicast to collaborators. Under X, this is naturally implemented by intercepting and multicasting X protocol messages. On Windows, the ITU T.128 recommendation for Multipoint Application Sharing specifies a graphics protocol that is conceptually similar to the X protocol. T.128-compliant clients have been implemented on X, Windows, and Macintosh, but T.128 application-sharing servers currently exist only on Windows.

Chapter 3

Groupware Architectures

Synchronous collaborative systems are inherently distributed. That is, components of the system execute on different host machines and communicate via a network. This chapter describes common software distribution architectures and their tradeoffs. The chapter begins with a brief discussion of distribution architectures in general, and next describes implementations of these architectures for synchronous groupware.

3.1 General Distributed Software Architectures

Distributed systems are used in a variety of applications: reservation systems for airlines, hotels, and car rental agencies; bank account databases; the World Wide Web (WWW), and groupware applications. Software distribution architectures fall in a range from **centralized**, where all of the shared data are maintained and processed at a single location, to **replicated**, where each site maintains and processes a complete copy of the shared data. The diagram in Figure 3.1 demonstrates a fully centralized architecture, where two different host machines have access to the same data. Figure 3.2 shows a fully replicated architecture for the same scenario.

Between the two extremes, a developer may choose to replicate portions of the shared data and centralize others. In fact, the fully centralized model is purely conceptual because, as a minimum, some representation of the data must be distributed in order to be useful. Thus, some replication occurs even in systems that we refer to as fully centralized. For example, most WWW browsers keep

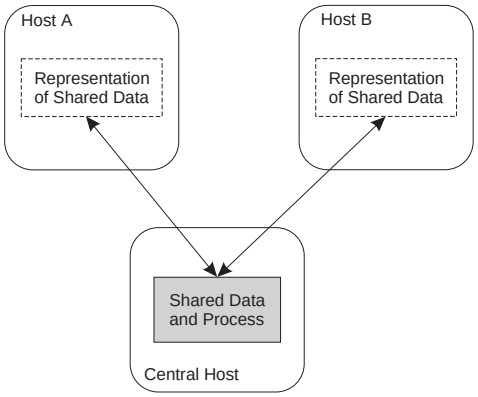


Figure 3.1: Centralized architecture. The shared data, indicated by the shaded box, are stored and processed at a single host. A person on a remote host views and manipulates the data via a representation, indicated by a dashed box. Here, two hosts, A and B, have access to the centralized data. Arrows indicate network traffic.

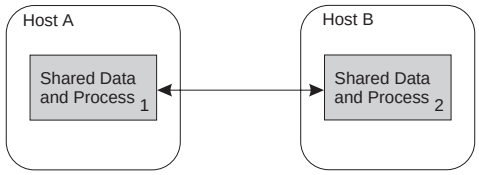


Figure 3.2: Replicated architecture. Each host maintains and processes a full copy of the shared data. When the data change on one host, all replicas must be made consistent. Arrows indicate network traffic.

a copy of each viewed page in a local cache of finite size. The original page's data is maintained at a central location, but are replicated on the host of each browser that views the page.

Each end of the centralized-to-replicated spectrum has merits and pitfalls. Typically, an implementation will use a hybrid approach. It is important to understand the tradeoffs so that a particular implementation will contain the advantages of each, rather than the disadvantages. Next, Sections 3.1.1 and 3.1.2 contrast the advantages and disadvantages of centralization versus replication in general. Sections 3.2.4 and 3.2.5 contrast the implications of using a centralized or replicated architecture in a synchronous groupware system.

3.1.1 General Tradeoffs of Centralization

A centralized model has a simple network connection layout. All data representations connect to and communicate with only one central server, resulting in a star topology. This topology has linear scalability. That is, the number of two-way network connections is simply n, where n is the number of remote representations.

A centralized architecture enforces data consistency by requiring that all modifications to the data be routed through a central authority. However, this can lead to unacceptable performance due to communication delays as each remote interaction must travel to and from the central site.

The centralized approach may be useful when the participating machines have different computation capabilities. For example, it is preferable to execute a complex computation on a powerful central machine, when the remote hosts are relatively weak. Conversely, centralization can be a disadvantage when the central machine is weak or burdened by several simultaneous processes or network connections.

The centralized model supports the desirable separation of user interface (representation) from application functionality (shared data). However, this requires the development of two separate processes, which has some cost beyond developing a single application as in the replicated approach.

3.1.2 General Tradeoffs of Replication

In a fully replicated model, each site holds a full copy of the shared data. Replication allows, but does not require, more complex connection topologies than does centralization. For example, it is possible to have the nodes communicate directly with each other, forming a complete graph. Thus, each replica can send a message directly to every other replica. However, this fully connected, peer-to-peer topology has exponential growth in the number of connections, $n(n-1)/2$. However, replicated systems may also route all communications through a single server, resulting in a star topology with linear growth, as described previously for the centralized approach. This topology will require each message from a replica to go through a central site, resulting in two transmissions, instead of one as in the fully connected topology.

When all hosts in a replicated system have roughly equivalent computing capabilities, local performance can be improved over the centralized architecture by processing data locally before updating remote copies. On the other hand, heterogeneous processing speeds can lead to perceived inconsistencies as some machines will display updated data more quickly than others (Greenberg & Marwood, 1994).

Because multiple copies of the data exist, consistency of the data is more difficult to guarantee than under centralization, where only one instance of the data exists. A related difficulty is the possibility of conflicting modifications generated from the concurrently running replicas. Approaches to ensuring consistency and concurrency control are discussed in Section 3.2.6.

Replicating the application process and data results in redundancy, which has disadvantages and advantages. The primary disadvantage is that it may be considered wasteful to have multiple processors performing the same operations on copies of the same data. The primary advantage, however, is that this redundancy helps reliability overall. If one host fails, the other hosts can continue processing their own copies. Hill *et al.* (1994) note that the reliability of the centralized model also can be enhanced by maintaining one or more redundant copies and switching to one copy in case the original site fails. However, applying redundancy to a centralized system is a form of replication, resulting in many of the same problems as any replicated system, thereby losing some advantages of centralization.

In a replicated model, only one application needs to be developed, as opposed to the two required by the centralized approach (interface and process). However, this single application will likely need to contain interface and process modules with similar development cost to that of the separate modules in centralization.

3.1.3 Summary of General Tradeoffs

Centralized architectures guarantee consistency of the data among participants because there is only one copy of the data. However, a centralized implementation typically requires higher network bandwidth to distribute graphics information than does a replicated implementation which distributes only minimal update information. Furthermore, centralized implementations have slower application feedback due to network latency as each user interaction must travel to and from the central location. Centralized approaches are less fault tolerant because the central host is a single source of possible system-wide failure. In contrast, replicated architectures can have lower bandwidth requirements, can have faster application feedback to interaction as the local copy can be updated before remote copies, and are more fault tolerant because multiple copies of the application and its data exist. However, the complexity of maintaining consistency among shared data replicas makes the centralized approach attractive.

3.2 Groupware Architectures

The preceding discussion applied to distributed systems in general, such as airline reservation systems. Such systems attempt to hide simultaneous users from each other, giving each the illusion

of sole access to the data. In contrast, users of groupware systems are aware that they are simultaneously accessing the shared data. This section describes implementation approaches and issues specific to distribution architectures for synchronous collaborative systems.

3.2.1 Centralized Display Broadcasting

Of the collaboration-aware applications and groupware toolkits reviewed in this dissertation, only Rendezvous (Hill *et al.*, 1994) and Kansas (Smith *et al.*, 1997) use a fully centralized approach. Some groupware toolkits, like Clock and DistView, use hybrid architectures in an attempt to maintain the advantages of each approach while obviating the disadvantages. Typically, the shared data are centralized and there are multiple copies of view-controllers. Developers of collaboration-aware software can keep the interface separate from the process, but it is difficult to make the separation in pre-existing single-user applications. Thus, collaboration-transparency system developers must choose to have the application fully centralized or fully replicated. Because of the difficulties with replication outlined in Section 3.2.5, centralization is commonly used by collaboration-transparency systems in the form of **display broadcasting** (Begole *et al.*, 1997a; Jones, 1993), where the graphical output of the shared application is distributed to each participant's host. In Figure 3.3, a central **conference agent** receives all user input and merges the events so that a single instance of the shared application receives a single stream of events. The conference agent then distributes display updates to each participant's host.

Display broadcasting is close to the concept of a fully centralized architecture (Figure 3.1). The client's processing requirement is low (receive and display graphics) and the data it maintains is primitive graphics information, rather than application-specific data. Therefore, the client has no knowledge of the semantics of user input to or graphic output from a shared application.

Another reason centralized implementations are common among conventional collaboration-transparency systems, is that the earliest window-sharing collaboration-transparency systems were implemented on the X Window System and its precursor TheWA, which are particularly suited to a centralized architecture. X and TheWA distribute display and input capabilities, allowing the display of a single instance of an application to be distributed to multiple users. Display broadcasting is also feasible on other platforms. For example, NetMeeting distributes graphics information via a protocol called T.128, which is similar to the X graphics protocol (see Section 2.4.2 for a description of T.128).

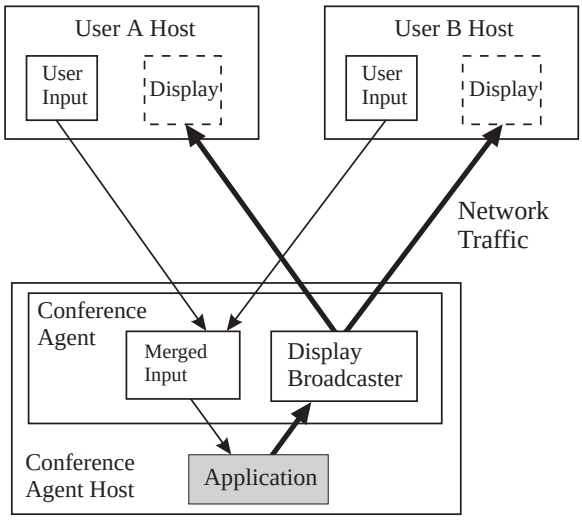


Figure 3.3: A centralized application (shaded box) shared via display broadcasting. Thin arrows indicate event traffic, which are merged and delivered to the application. Bold arrows indicate (high bandwidth) display data, which is distributed and displayed on each collaborator's host, represented by the dashed boxes.

3.2.2 Replicated and Semi-replicated Approaches

Fully replicated architectures are often used by collaboration-aware groupware toolkits, such as GroupKit (Roseman & Greenberg, 1996a) and DistView (Prakash & Shim, 1994), primarily because they provide faster local application response than the centralized model. Some groupware systems use a semi-replicated approach, such as Patterson *et al.*'s (1996) centralized notification server, where only the shared data are centralized and views are replicated. The Clock groupware system of Graham *et al.* (1996) uses a conceptually centralized model-view-control paradigm where the shared data are held in the centralized model, but updates are cached, pre-sent, or pre-fetched at each participant's host, essentially replicating the data. See Section 2.4.1 for a review of groupware toolkits.

In collaboration-transparency systems, the few replicated implementations use **event broadcasting**, where each participant has a copy of the application and user inputs are distributed to each copy. Figure 3.4 illustrates the replicated approach in which one component of a central conference agent is an **event broadcaster** whose job is to multiplex an input event from any participating collaborator to every other collaborator's replica. This allows every collaborator to invoke events to

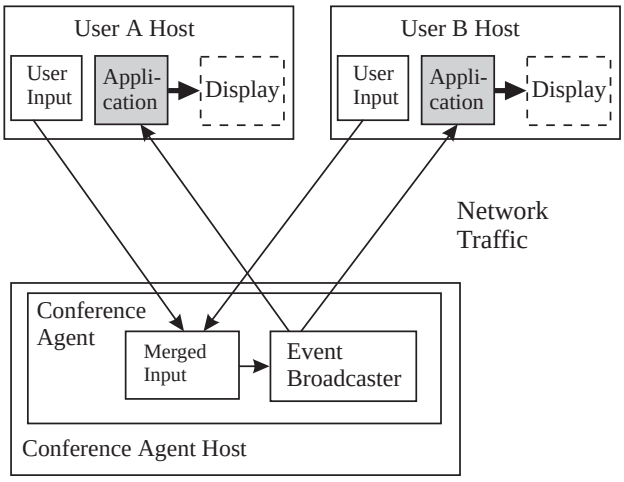


Figure 3.4: A replicated application (shaded boxes) shared via event broadcasting. Arrows indicate event traffic.

the system, and those events along with the feedback that the events create are seen by all collaborators. For example, if one collaborator moves a scrollbar, all participants see their corresponding copy of the scrollbar move. The replicated data remains consistent because each deterministic application receives the same input stream.

3.2.3 Common Groupware Problems

In a groupware application, multiple users may simultaneously generate potentially conflicting inputs. Strategies for concurrency control in groupware have been investigated by many researchers (Ellis & Gibbs, 1989; Greenberg & Marwood, 1994; Karsenty & Beaudouin-Lafon, 1993; Mitchell, 1996; Sun & Ellis, 1998), and are briefly described here.

Coarse-grain Locking The easiest way to prevent conflicting operations is to use **locking**, where only one user at a time can change the shared data. At the coarsest granularity, a user may be required to hold a lock on the entire shared application. This is called **floor control**, and is commonly used in collaboration-transparency systems. The disadvantage of floor control is that it prohibits participants from working simultaneously in the application, limiting participants' ability to collaborate in parallel (see also Sections 4.1.1 and 6.1.3).

Fine-grain Locking It is possible to provide fine-grain locking, allowing collaborators to safely manipulate different portions of the shared data concurrently, such as paragraphs, sentences or words in a text editor. Locking policies range from **optimistic**, where a lock is tentatively granted locally upon request (the pre-locked state must be returned to if a conflict occurs), to **non-optimistic**, where the local copy must wait for the lock to be granted by a global (possibly distributed) locking arbitrator. Optimistic policies allow local interactions to be processed quickly in a replicated system.

3.2.4 Problems in Centralized Groupware

The primary advantage of the centralized approach is that it avoids the problem of maintaining data consistency among replicas because only one copy of the shared data exists. However centralization has the following disadvantages.

Higher Network Usage Requirement Display broadcasting generally requires more network bandwidth than replicated approaches, which send only minimal update information. In a study of X protocol traffic, Dunwoody and Linton found that users of typical X-based applications generate relatively few input event messages compared to the number of graphics messages, approximately 1:25 (Dunwoody & Linton, 1988; Lauwers, 1990). Furthermore, event messages are always 32 bytes long, but graphics request messages have sizes ranging from 32 bytes to 64 Kilobytes (for those that contain bitmap images) (Gupta & Hwu, 1993). These factors account for the nearly 1:10 ratio of event to graphics traffic noted by R. Smith (1996b) in an analysis of network traffic for an X-based shared environment called Kansas (Smith *et al.*, 1997).

The lower network demand of event broadcasting is apparent when we note that inputs are transmitted in both replicated and centralized architectures, but the centralized approach uses additional bandwidth to distribute the display information. Savings in network usage is especially relevant to sharing applications over a wide-area network, like the Internet.

Slower Application Response to User Inputs In the X environment, it is possible to have the application client and the X server reside on different hosts. Thus, the application process can execute on a different host than the host on which the user interface is displayed. When that is the case, network latency and throughput constraints result in noticeable delays of application response to user input. The delays are caused by the network lag associated with each input traveling from

the user's host to the application's host and the returning graphics messages. Users may find this delay acceptable, particularly on a local area network where latency is typically 10 milliseconds and throughput is often 3 Megabits/sec or greater. When the application to be shared already resides on a different host than the intended display, it can be argued that X-based centralized collaboration-transparency systems do not further slow application response to user inputs because multicasting the display messages to additional display servers will not add to the delay already present.

In practice, however, a single-user application and its display typically reside on a one machine. Latency is negligible within the single machine and throughput is only constrained by the machine's internal data transfer rate. Therefore, sharing an application via centralized display broadcasting does delay application feedback to user inputs. As observed by Bhola *et al.* (1998), the delay may be unacceptable to users, particularly on wide-area networks, where latency within the USA typically ranges between 150 and 300 milliseconds and data throughput via a modem connection is no greater than 56 Kilobits/sec.

Under a replicated environment, the response to local user input can be as fast as any single-user application. User inputs can be processed locally first, then distributed to remote replicas. The updates to remote replicas will be affected by network latency, but the local response will not.

Device Dependencies Another difficulty for centralized display broadcasting systems is differing capabilities of heterogeneous display servers. Display devices have several properties, such as color depth, number of simultaneous colors, and resolution (Abdel-Wahab & Jeffay, 1994; Wolf *et al.*, 1996). It is difficult to display a single instance of an application consistently on multiple displays of differing capabilities. An approach is to choose a commonly available set of capabilities, typically the least capable display. This may result in less than optimal display quality for some collaborators.

Under a replicated architecture, each copy of the shared application can use the capabilities of its local display. Note that this may result in differences in what each collaborator sees.

In addition to the implementation difficulties described here, centralization imposes restrictions on the collaborative usability of the shared application. Chapter 4 enumerates the usability deficiencies of conventional collaboration-transparency systems, which are primarily due to the use of centralized display-broadcasting architectures.

3.2.5 Problems in Replicated Groupware

Replication has several advantages over centralization: lower bandwidth requirement, faster local application response to user input, and fault tolerance. However, replication has some disadvantages, primarily related to the complexity of maintaining consistency among shared data replicas. This section describes problems and solutions for a replicated system.

Replicated Collaboration Transparency

Replicated architectures for collaboration transparency has attracted prior research with mixed results. At the time of their work on VConf and Dialogo, Lauwers *et al.* (Lauwers, 1990; Lauwers *et al.*, 1990) concluded that while a replicated architecture holds promise, it could not maintain collaboration transparency with the then-present technology. From their experiences with MMConf, a groupware toolkit that uses event broadcasting, Crowley *et al.* (1990) concluded that "it is almost impossible for a replicated architecture to assure correct behavior in an unmodified single-user application." However, they further concluded that relatively minor modifications to applications and toolkits are required to allow applications to run robustly under a replicated architecture. Summarizing earlier work, Minenko (Minenko, 1998; Minenko, 1996) deemed event broadcasting as "unusable" for the following reasons.

1. Sharing must be based on the assumption that application behavior is deterministic.
2. Application environments of all involved hosts must be identical.
3. Copies of the shared application's executables must be available on each host.
4. Accommodating late-comers into an existing collaborative session is nearly impossible because the state of shared applications must be updated by playing back the input events of the whole session. This process is unstable and can continue for a long time.
5. The most significant disadvantage is difficulty in maintaining the consistency between the multiple copies of the application. For example, the loss of input to one instance of the application due to network loss can throw it out of synchronization with the others.

However, these difficulties can be overcome in the following ways (See also Begole *et al.* (1997a; 1997b)).

1. Applications are in fact deterministic. Applications that seem nondeterministic by behaving differently from one session to another are receiving different inputs. Interacting with the run-time environment for information such as current time, random number seeds, files, sockets, etc. can lead to seemingly nondeterministic behavior. However, if *all* input to an application is replicated to each copy of the application, the states of all copies should be consistent.

2. Related to the previous point, the application environment can be made identical by ensuring that each copy of the application receives the same response for the same system call (e.g. system time) and that all applications share input–output resources such as files and sockets. This implies that such resources must be controlled by the conference agent, which will distribute input and output to all participants. Section 3.2.6 describes an approach to transparently provide the same input to all replicas from sources such as files, sockets and system time.

3. Copies of the application software can be provided to all participants. Java executables, for example, can be delivered to any host connected to the Internet.

4. Event replay is not the only way to provide late-comers with a replica of the current state of an application. Chung *et al.* describe an application-migration approach called **image-copy,** where an image of the process in memory is copied (Chung & Dewan, 1996; Chung *et al.*, 1993). The time required by this approach is dependent on the size of the application, rather than the length of the collaboration.

5. Consistent input to all replicas will result in consistent replica states. Approaches to ensuring consistency are described next in Section 3.2.6.

Replication is suitable for applications that do not have **time-dependent** inputs. That is, for a given stream of inputs, the resulting application state will be the same regardless of the time interval between any two inputs. In contrast, consider an application where a user selects a *moving* object by clicking their mouse on it. This interaction is time-dependent because if the click does not arrive at all replicas at the time that the object has moved under the click location, the resulting states will differ. Because network lag varies, time dependency is a general problem for replicated systems. A developer of a replicated collaboration-aware application that has a time-dependent interaction can address the problem (e.g., see (Begole & Shaffer, 1997)), but it is difficult to handle transparently. This limitation of replication applies to a relatively small set of applications, such as

animated games and simulations. Many useful applications may be shared under a replicated event broadcasting approach. In prior work (Begole *et al.*, 1997a; Begole *et al.*, 1997b), we found that current technology can support a replicated architecture and that it is preferred for a Web-based, or other wide-area network, collaboration-transparency system.

3.2.6 Ensuring Consistency in Replicated Groupware

Because multiple copies of the data exist, consistency of the data is more difficult to guarantee under replication than under centralization, where only one instance of the data exists. A related difficulty is the possibility of conflicting modifications generated from the concurrently running replicas, described in Section 3.2.3. This section describes approaches to ensuring data consistency combined with concurrency control.

Serialization

Consistency is assured if each deterministic replica receives the same inputs in the same order. **Serialization** is the process of imposing a total ordering on events which change the data so that each copy receives events in the same order. Serialization may be **optimistic,** where events may be received and acted upon out of order, in which case inconsistencies must be detected and repaired, or **non-optimistic**, where events are always received in order, typically by routing through a central server. When optimistic serialization is used, the local copy of an application can act on an event before it is broadcast, allowing faster feedback of local interactions than in non-optimistic serialization. However, optimistic serialization is vulnerable to conflicting operations generated by concurrently running replicas (e.g., Figure 4.1).

Operational Transformation

One form of optimistic serialization is called **operational transformation** (Ellis & Gibbs, 1989; Sun & Ellis, 1998). An operational transformation algorithm requires data changes to be represented by atomic operations. The operation is applied locally, stamped with a global time, and then distributed to replicas. A set of operations may arrive in different orders at each replica. Each replica *transforms* incoming remote operations so that they are consistent with its own set of previously applied operations. Section 5.3.7 describes operational transformation in greater detail.

System Resources

So far, I have discussed distribution of user input and display output for collaborative systems. In addition, applications commonly acquire input from and write output to many sources, such as files, network connections, and other processes. Additionally, an application may access its run-time environment for information such as the current time, host name, operating system name, and environment variables. I use the term **external resources** for sources of input or output that are external to the application and not provided to or from the user. That is, an external resource is any input–output source other than display output or user input.

A centralized system can access external resources directly on the central host, just as a conventional single-user application does. A replicated system, however, is not as easily handled. For example, if a replicated application needs to read a file, on which host should the file be opened? If each replica asks for the name of the host on which it is running, each will receive a different name. Querying the system time will result in a different value on each host. In order to ensure consistency among the replicas, the system must ensure that all replicas receive the same response to such queries.

Replicated output can also pose a problem. In some cases it is acceptable to allow each replica to generate a copy of the output. For example, other than being wasteful, there is little problem if each replica of a collaborative editor wrote a copy of the same file on its local host. However, in some cases it is not acceptable to have more than one output from all replicas. For example, it would be more than just wasteful, if each replica of a collaborative electronic mail application posted a copy of the same message.

Not all external resources should necessarily be replicated. Applications depend on many "environment variables," such as the user's home directory, current working directory, and executable search path. If replicas are running on heterogeneous platforms and one is given the value of an environment variable from a different platform, the replica may behave incorrectly. For example, the user's home directory is stored in an environment variable, named `$HOME` on UNIX machines and is different for each user. If a replica running on one user's machine requested the value of `$HOME` and was given the home directory of a different user and that directory did not exist, the replica would fail to access that directory. Therefore, the groupware application developer must take care to distribute only those parts of the external environment required for data consistency among the replicas.

Many *ad hoc* collaborative applications and groupware toolkits use a replicated or semi-replicated architecture. However, few groupware toolkits provide support for handling external input and output resources. The developer must be aware of the issues, problems, and solutions to manage external resources correctly. This section describes strategies to handle external input–output resources in a replicated system. The first part describes *ad hoc* methods to use when there is no support for external resources. The second part describes transparent replication of external resources.

Explicit Replication

The following strategies are used by groupware system developers to handle external resources in a replicated system.

Network Accessible Resources An external input resource may be accessible from all replicas. For example, a file that is delivered by a server can be loaded by all replicas, so long as they all have access to that server. Many replicated whiteboard applications, for example, use this approach to load image files from the WWW.

Single-source Resources Typically, an external resource is only available from one source and cannot be directly accessed by each replica. For example, a particular data file may reside on only one of the hosts in the replicated system. Even if all hosts have access to a similar resource, such as the system time, we may want to designate one as the source of that resource. Thus, we can ensure that all replicas receive the same information.

A common approach is to have just one replica read the data and explicitly distribute the data to the other replicas. For example, consider a single-user text editor that reads an input file and appends the file contents to a document by invoking some function, say `append(String newText)`[1]. There are several ways a replicated multiuser editor might handle this situation. Figure 3.5 shows a code fragment in which data are read by one replica. The replica explicitly generates a message and sends it to all replicas, including itself.

It is unnecessarily wasteful for the originator to send the data to itself. Some toolkits (e.g., GroupKit, and JSDT) provide a means to only send a message to other replicas of the application,

[1] Java language syntax is used in all program examples in this dissertation

```
 1.  public static final int MAXBUFFSIZE = 128;

 2.  public void readFile(String fileName) {
 3.    FileReader inFile = new FileReader(fileName);
 4.    char inBuffer[MAXBUFFSIEZE];
 5.    int numRead = 0;

 6.    while (inFile.ready()) {
 7.      numRead = inFile.read(inBuffer, 0, MAXBUFFSIEZE);
 8.      sendMessage("appendText", numRead, inBuffer);
 9.    }
10.  }

11.  // on each replica, the following exists:

12.  public void receiveMessage(String msgType, DataInputStream dis) {
13.    if ("appendText".equals(msgType)) {
14.      String inputChars = dis.readUTF();  // reads a string of text
15.      appendLocal(inputChars);
16.    } else if ("someOtherMessage".equals(msg) {
17.      //....
18.    }
19.  }

20.  public void appendLocal(String inputString) {
21.    doc.append(inputString);  // text is appended to each document replica
22.  }
```

Figure 3.5: Sample code to read a file into a replicated collaborative text editor. Data are read from the file and then sent in a message to all application replicas (lines 1–10). At each replica, when the message is received, `receiveMessage()` is invoked (line 12), which determines the type of message received and handles each type appropriately. For example, when the "`appendText`" message that was sent in line 8 is received by each replica, the text is extracted from the message and then appended to the local copy of the document by invoking `appendLocal()` (lines 12–22).

```
 6.  while (inFile.ready()) {
 7.    numRead = inFile.read(inBuffer, 0, MAXBUFFSIEZE);
 8.    appendLocal(new String(inBuffer, 0, numRead));
 9.    sendToOthers("appendText", numRead, inBuffer);
10.  }
```

Figure 3.6: Code to send file data only to remote replicas. Locally, the data are applied directly in line 8. This code replaces lines 6–9 in Figure 3.5

not including the local replica. Using such a function, lines 6–9 of the code in Figure 3.5 could be replaced by the code seen in Figure 3.6.

In the above examples, the developer was required to specify a message protocol, which consists of several components. First, a message must be created that contains the information to distribute. Second, the message must be explicitly sent to the proper receivers. Third, each receiver must parse and handle the message correctly. Each type of message must have a unique identifier, so that the receiver can handle each type differently. For example, in Figure 3.5, a message of type `appendText` contains text to append to the local document, whereas some other action is performed upon receipt of a message of type `someOtherMessage`.

Another approach is provided by GroupKit, called Multicast Remote Procedure Call (MRPC). MRPC alleviates the developer from having to send and receive messages explicitly. With MRPC, any component of a replicated system can directly invoke procedures on remote replicas. This is similar to remote procedure call or remote method invocation, such as provided by the Common Object Request Broker Architecture (CORBA) (Mowbray & Zahavi, 1995), Microsoft Distributed Component Object Model (DCOM) (Brown & Kindell, 1996), and Java Remote Method Invocation (RMI) (Wollrath *et al.*, 1996). MRPC adds the ability to make the invocation on *multiple* remote processes simultaneously. When the invocation is made, MRPC marshals the parameters of the call, distributes the data, and invokes the method on remote components. Thus, the twenty-two lines of code in Figure 3.5 could be simplified to the twelve lines seen in Figure 3.7.

Transparent Replication

It is possible to relieve the developer from having to treat external resources in a collaborative application differently than resources are conventionally handled in single-user applications. One approach is to automatically replicate a resource so that each replica has an identical copy. MMConf, a groupware toolkit, and Dialogo, a collaboration-transparency system, implemented this approach

```
 1.  public static final int MAXBUFFSIZE = 128;
 2.  FileReader inFile = new FileReader("sample.txt");
 3.  char inBuffer[MAXBUFFSIEZE];
 4.  int numRead = 0;
 5.  while (inFile.ready()) {
 6.    numRead = inFile.read(inBuffer, 0, MAXBUFFSIEZE);
 7.    invokeOnAll("appendLocal", new String(inBuffer, 0, numRead));
 8.  }

 9.  // on each replica, the following would be invoked:

10.  public void appendLocal(String inputString) {
11.    doc.append(inputString);  // text is appended to each document replica
12.  }
```

Figure 3.7: Code using a multicast remote procedure call to send file data to replicas.

(Lauwers, 1990). In Dialogo, a subdirectory was designated as a "conference directory," and any file placed in it was replicated to other participants' conference directories.

Physically copying all external resources does not work in all cases. For example, if the collaborative application uses the fully qualified path name to a file, but each participant's conference directory resides in a different path, some replicas may fail to load the file. Additionally, differing file naming conventions (e.g., Macintosh versus UNIX file systems) prevent uniform file names across replicas. Furthermore, many external resources are closely tied to the underlying system. For example, each replica can query the system time, but it will be different on each replica's host. Finally, in some cases it is infeasible or impossible to literally replicate an external resource. For example, consider a client-server system where the server accepts only one connection at a time. It would be impossible for each replica of the client to have an individual socket connection to the server.

Proxied External Resources

Although it can be impractical to literally copy each external resource, it is still possible to provide the same input data to each replica. This can be accomplished by intercepting data from a single instance of the external resource and distributing the data to all replicas. In prior work, we designed and implemented such a system for sharing input files in a Java-based applet-sharing system, called JAMM (Java Applets Made Multiuser) (Begole *et al.*, 1997b), a precursor to Flexible JAMM described in Chapter 5.

In this semi-replicated approach, the external resource object resides at a single location and is accessed via replicated **proxies** (Gamma *et al.*, 1996) that multiplex input and output to and from the single instance of the external resource. JAMM replaces the original resource objects with proxies when the user initially shares the applet using a mechanism provided by Java Object Serialization (JOS), which is used to send a copy of the application in its current state to a late joiner. The proxy acts in place of the original resource and appears to be an instance of the original to an application. The application programmer is completely unaware that this replacement happens, maintaining transparency.

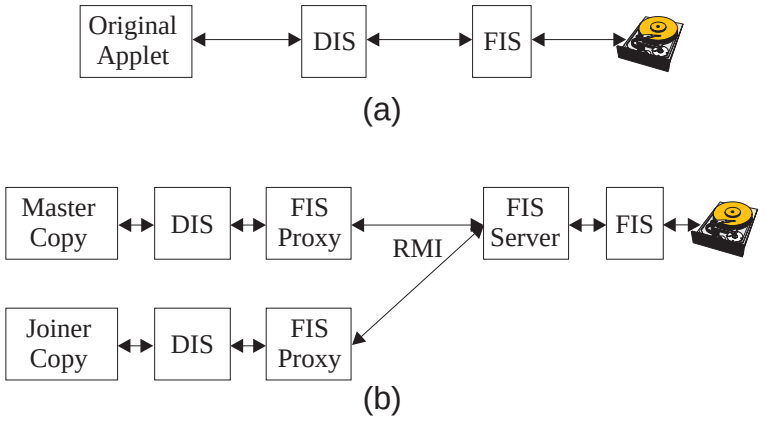


Figure 3.8: (a) The original applet has a `java.io.DataInputStream` (DIS) connected to a `java.io.FileInputStream` (FIS), which reads a file from a physical disk. (b) A proxied FIS object is shared by two application replicas. The original FIS resides at the original (master) host and is accessed via a server object. Proxies replace the original FIS in each copy of the applet and communicate with the server via RMI.

Figure 3.8a shows an example of an applet with a `FileInputStream` object. The applet reads data through an intermediary `DataInputStream`, which provides convenient access methods to data in the file. Figure 3.8b shows a proxied `FileInputStream`. A proxy resides at each replica of the shared application and communicates with the original object through a Java RMI server.

The remote file implementation consists of a client, called the proxy, and a resource server. The proxy is a class that extends the original external resource class. Therefore, as a subclass of the original, the proxy will pass the same run-time type checks as the original. The server includes an interface which is required by Java RMI, and the server implementation. The server implementation,

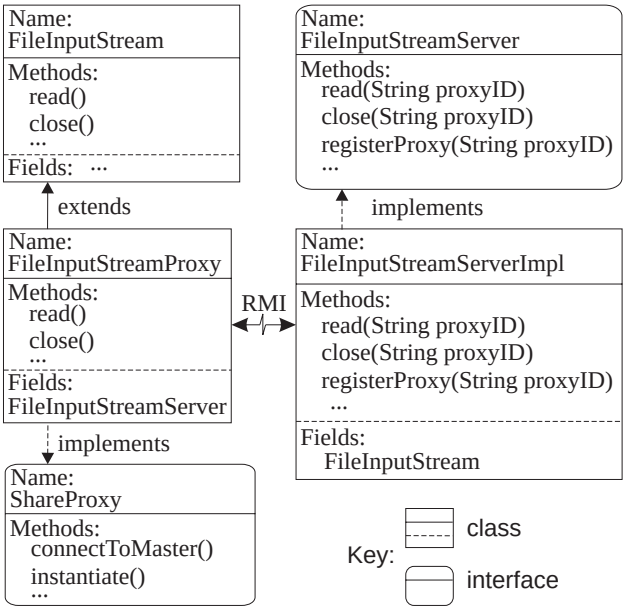


Figure 3.9: Implementation of a proxy for the `FileInputStream` class.

when instantiated, is passed a reference to the original resource, from which the server retrieves data. The data are cached by the server so that, as each replica makes the same data request, the server will give the same data to each.

Figure 3.9 shows how this scheme is implemented for a `java.io.FileInputStream` proxy and server. The proxy implements a `ShareProxy` interface which defines a method for connecting to the server. This is called when the proxy object is created in a joining replica, and registers this proxy with the server. Thus, the server knows how many proxies will be making requests.

The server keeps track of how many requests each proxy replica makes. When a replica makes a request, the server checks to see if the replica is the first one to make this request. If so, then the real external resource is queried for data, which is cached for the rest of the replicas, and then sent back to the requesting replica. If the replica is not the first to make a request, then the data for the request being made was previously cached, and so are sent directly to the requesting replica. Data are only buffered so long as they are needed. When all the active replicas have made the same request, the data for that request are released.

Output proxies, such as `FileOutputStream`, use a similar approach, except that only the first request to send data is actually written to the external resource. All subsequent write requests from other replicas are dropped.

Introducing Proxies After Sharing In the JAMM prototype, an external resource could only be replaced by a proxy at the time the applet was shared. That was because JAMM used JOS to replace an external resource object with a proxy. When the applet is initially shared, JOS allows the replacement of external resources with proxies. This was an important limitation of the JAMM prototype's implementation of sharing external resources. External resources could only be shared if they existed *before* the applet was shared because JAMM could not create a server or proxy *after* an applet had been shared.

Flexible JAMM (Chapter 5) has the added capability to create external resource servers and proxies after the application has been shared. To accomplish this, it was necessary to modify Java-level classes and native platform code in the standard Java run-time environment. Thus, the implementation uses a non-standard Java run-time environment.

In the new implementation, instead of *replacing* the external resource with a proxy, each external resource class contains a reference to either a proxy or a local resource object. This approach follows the **bridge** design pattern, described by Gamma *et al.* (1996). A bridge decouples an abstraction (e.g., `FileInputStream`) from its implementation (e.g., a physically remote file or a physically local file), allowing the implementation to change at run time. When such an external resource object is used in a shared application, the external resource object accesses its proxy object, otherwise the local resource is accessed. A resource object can switch from using a local resource to a remote proxy, if the application using it is later shared. The original local resource is wrapped by a resource server, as in Figure 3.8.

Java External Resources On the Java platform, external resources are easily identified. As a rule, a Java external resource class will access the local resource via a **native** method, which is a platform-specific implementation of a method to which the platform-independent Java Virtual Machine (JVM) passes control. For example, the `java.io.FileInputStream` reads an integer from a physical file via a native method named `read()`. External resource classes in the core Java 1.1 class library are listed in Table 3.1 (the list is also posted on the Flexible JAMM Web site (Begole,

```
 1.   boolean shared = false;

 2.   public boolean inShareMode() {
 3.     return shared;
 4.   }

 5.   private void setShared(boolean value) {
 6.     shared = value;
 7.   }

 8.   FileInputStream proxy = null;

 9.   public int read() throws IOException {
10.     if (inShareMode())
11.       return proxy.read();
12.     else
13.       return readNative();
14.   }

15.   private native int readNative() throws IOException;
```

Figure 3.10: Code to replace the original public native `FileInputStream.read()` method. If the `FileInputStream` is being used within a shared application, the proxy will be accessed for data (line 11). Otherwise, the private native method, `readNative`, will be accessed (lines 13 and 15).

1997)). Note that not all external resources should be proxied, such as `java.io.FileDescriptor` and `java.lang.Runtime`.

Some native methods are **private**, meaning they can only be invoked by objects of the class in which they are defined. These are indirectly invoked by an application via **public** Java methods that in turn call the private native method. In my implementation of post-shared external resources, these public Java methods are overridden. If the resource is being used normally, i.e., not shared, the native method is invoked as before. Otherwise, a proxy is accessed.

A **public native** access method, such as `FileInputStream.read()`, is not as easily wrapped because the native implementation is directly executed when the method is called. For these cases, I privatized the original public native method and renamed it in the form <*originalName*>`Native`. Then the method with the original name is turned into a Java implementation (i.e., non-native) that follows the bridge pattern as described. For example, the code seen in Figure 3.10 replaced the public native `FileInputStream.read()` method.

Table 3.1: Java external resource classes.

External Resource	**Reason**
java.io.FileInputStream	FileInputStream reads a file sequentially.
java.io.FileOutputStream	FileOutputStream writes to a file sequentially.
java.io.RandomAccessFile	A RandomAccessFile reads from or writes to a file in a nonsequential manner.
java.io.File	File contains platform-specific information for path and file separators, and provides operations to create a directory, determine a file's access permissions and type, etc.
java.io.FileDescriptor	FileDescriptor represents an open file or socket. Applications should not create FileDescriptor objects directly, so this externality should not need to be proxied.
java.lang.Runtime	Runtime provides information about the run-time environment, such as the amount of free memory. Runtime should not be proxied because each replica will have a machine-specific run-time environment.
java.lang.Process	Process objects are returned by exec() calls on Runtime and provide access to a child process's input and output streams. Process is an abstract class and is therefore not inherently an external resource, but subclasses of it are.
java.lang.System	System accesses properties, standard input and output streams (stdin, stdout, and stderr) and the current time.
java.net.InetAddress	InetAddress provides internet address information including the local host name and address. InetAddress itself has no native methods, but contains a platform-specific subclass of InetAddressImpl, which has native methods.
java.net.DatagramSocket java.net.Socket	These classes provide input and output to a network. These classes do not contain native methods themselves, but use platform-specific subclasses of the abstract SocketImpl and DatagramSocketImpl classes.
java.util.Random	Random generates psuedo-random numbers based on an initial seed value. The system time is the default seed. Random is not an external resource, but accesses system time.

Instantiating a Proxied External Resource When an object of an external resource class is constructed, it must determine if it should access a proxy or a local resource. Not all external resource objects should be proxied when sharing an application under Flexible JAMM. For example, the Java Virtual Machine needs to open files to read class data, and query the system time to determine when to execute garbage collection. Only an external resource object that is instantiated by the shared application should be proxied.

To test for this, Flexible JAMM loads all classes in a shared application via an implementation of `java.lang.ClassLoader`. Each `java.lang.Object` has a reference to its class, which in turn has a reference to the `ClassLoader` in which it is defined (Gosling *et al.*, 1997). Therefore, an instance of `ClassLoader` defines a unique group of objects, the objects instantiated from classes loaded in the `ClassLoader`.

Flexible JAMM uses class loader scope in a manner similar to how Java applet security determines whether to allow access to a restricted resource, such as a file. The applet security manager checks to see if an object was loaded via an applet `ClassLoader`, which are not allowed access, or via the system's default class loader, which are allowed access. In Flexible JAMM, when an application is shared, a flag is set in the application's `ClassLoader` to indicate that objects in this class loader's scope are shared. When constructed, an external resource class queries the Flexible JAMM security manager to determine if it should construct a proxy or local resource. The security manager in turn checks the class loader of each object on the execution stack. If any class loader is set to share mode, then the external resource should and does construct a proxy.

If a shared external resource is being instantiated on the "master" replica, then Flexible JAMM's proxy manager creates a server and a proxy. The proxy manager also sends a message containing the reference to the remote server to all replicas. On each remote replica, the proxy manager creates a proxy and waits for the reference to the remote server to arrive. Once the reference arrives, the replica's proxy connects to the remote server, completing the proxy–server connection as seen in Figure 3.8.

System Time Problem The previously described approach works for the majority of the external resource classes listed in Table 3.1. However, there is a problem applying this solution to `java.lang.System`, which is used to query the current time. `System` is unique among the listed classes in that an application does not instantiate `System` objects, but calls a (static) class method,

`currentTimeMillis()`, to access an external resource, the current time. Therefore, for each replica, there is one instance of the time proxy contained by the `System` class on the replica's host. When a newcomer's `System` proxy registers itself with the server, the server sets the newcomer's current request number to be the same as the latest request number. A problem arises if several objects on a newcomer's replica query the system time for an initial value. The time server cannot know how many requests such initializations will make, and therefore the newcomer's requests can become unsynchronized with the other replicas.

For example, consider an application that contains a timer object which is used to query and display the current time once a second. When shared, this timer will be proxied so that all replicas will display the same time. So far, this example will work using the proxy approach as outlined. That is, for each participant, the server will set the proxy's current request number to be the same as the latest request number. When the application replica queries the time proxy, the proxy queries the server, and the server will return the latest requested time. However, suppose the application has another timer which is used to blink a text caret in a text field. Now, when the application is copied to and initialized at a remote site, two timers will query the time, which will increment the current request number on the server to one greater than that of the other replicas. Therefore, the newcomer will be one time request ahead of the other replicas. This problem cannot be avoided because the time server cannot know whether a time request is to be used to initialize a newcomer (for which the server could return the time and not increment current request number) or is part of the expected application behavior (for which the server should increment the current request number).

Discussion

Most groupware toolkits use a replicated or semi-replicated environment. Yet few provide facilities to ease the use of external data resources, such as files, sockets, and system time. It is possible to transparently provide semi-replicated access to external resources via the proxy-server approach described in this section. By incorporating this approach, a replicated collaboration-transparency system can provide shared applications with access to external resources. A groupware toolkit can use this approach to further facilitate the development of synchronous collaborative applications by simplifying the use of replicated external resources.

Table 3.2: Feature comparison of groupware architectures.

	Ensuring Consistency	Bandwidth	Network Latency	Type of Network	Client Load	Number of Users	Type of User Interface
Replicated Collaboration-Transparency	difficult	lower	longer	LAN–WAN	higher	2–10, and >10	frequent graphics changes, no time dependencies
Centralized Collaboration-Transparency	easy	higher	shorter	LAN	lower	2–10	infrequent graphics changes, time dependencies
Semi-replicated Collaboration-Aware	easy–difficult	lower	longer	LAN–WAN	higher	>2	frequent graphics changes, time dependencies

3.2.7 Comparison of Architectures

Full centralization and full replication define the limits of the range of distribution architectures for groupware systems. Typical collaboration-aware systems use a hybrid architecture, generally maintaining a central copy of the shared data and replicating the user interface (see Table 2.3 for a summary of distribution architectures used by groupware toolkits). However, it would be difficult for a collaboration-transparency system to determine the points of separation between an application's interface and data. Therefore, collaboration-transparency systems must use full centralization (more common) or full replication (less common).

Table 3.2 summarizes the tradeoffs of each approach for collaboration transparency. The values of the entries are approximations intended to illustrate the distinctions among the approaches, not represent absolute limits. For comparison, the table includes the characteristics of an archetypal semi-replicated collaboration-aware application. Each of the comparison factors is described next.

The first factor in Table 3.2 is the difficulty of implementing strategies to ensure consistency of the shared data, which is among the more difficult implementation problems and is therefore a primary concern of developers. Guaranteeing consistency is more difficult under replication because multiple copies of the data exist, than under centralization where there is only one instance of the data. Another key factor is the network bandwidth required by the system. Replication generally uses less network bandwidth than centralization, as described previously in Section 3.2.4. The next factor is the impact of message delay due to network latency. Under replication, locally generated changes can be applied immediately to the local copy of the application. Under centralization, each user interaction must travel from the user's site to the central site and back. Therefore,

replication is tolerant of longer network latency than centralization (see Bhola *et al.* (1998) for an empirical study of application responsiveness under differing architectures, latency, and consistency strategies). Combining the bandwidth and latency characteristics, provides the criteria for the type of network that can support the different architectures. Replication can be used on local- to wide-area networks with high- and low-bandwidth (e.g., modem) connections, whereas centralization is effectively limited to high-bandwidth local-area networks. The computational load used by the client end of each of these architectures is also important. Replicated systems impose a higher load on each collaborator's host than centralized because replicated systems execute a copy of the application on each host. Another factor by which groupware systems are measured is their scalability in terms of number of users. Collaboration-transparency systems are generally intended to support a small number of simultaneous users (generally between 2 and 10) because passing control of the application becomes unwieldy as the number of users grows. Both architectures generally can support groups of small size. However, a collaboration-transparency system can be used as a presentation system, so that one person can demonstrate a piece of software to a potentially large audience, for example. The scalability of the number of participants is constrained by the available network bandwidth, and replication scales better than centralization because replication uses less bandwidth and does not necessarily have a message distribution bottleneck at the central application host. A final point of comparison is the type of application each approach can effectively share. Centralization can be effective when sharing an application that does not frequently change the graphical output because the fewer graphics update messages will need to be sent than when sharing applications that produce frequent graphics changes. On the other hand, replication can be used effectively with either type of application interface. However, replication is not suitable for application that have time-dependent user interactions.

3.3 Summary

This chapter describes common software distribution architectures for synchronous collaborative applications. Tradeoffs of architecture approaches are discussed both in general and as the architectures are implemented in groupware systems.

Software distribution architectures fall in a range from **centralized**, where all of the shared data are maintained and processed at a single location, to **replicated**, where each site maintains

and processes a complete copy of the shared data. Centralized architectures guarantee consistency of the data among participants because there is only one copy of the data. However, centralized display-broadcasting typically requires higher network bandwidth to distribute graphics information than does the replicated event-broadcasting approach. Furthermore, centralized implementations have slower feedback to user inputs due to network latency as each user interaction must travel to and from the central location. Centralized approaches are less fault tolerant because the central host is a single source of possible system failure. In contrast, replicated architectures can have lower bandwidth requirements as only minimal update information needs to be transmitted, can have faster feedback to user input as the local copy can be updated before remote copies, and are more fault tolerant since multiple copies of the application and its data exist. However, the complexity of maintaining consistency among shared data replicas makes the centralized approach attractive.

The centralized approach is commonly used in collaboration-transparency systems, particularly on the X Window System, which defines a graphics protocol allowing an application's process to execute on a different host than its display. T.128 defines a similar graphics protocol that is used by some collaboration-transparency systems on the Windows platform.

Most groupware toolkits use a replicated or semi-replicated approach. However, replication has previously been deemed unsuitable for collaboration transparency. Researchers have identified several difficulties that must be addressed to maintain consistency among the replicas: heterogeneous run-time environments; access to application executables; apparently nondeterministic application behavior; distributing inputs other than user input events; and accommodating late-comers. These difficulties can be overcome in the following ways: each replica's run-time environment can appear identical by ensuring that each replica receives the same response to the same system call; access to application executables can be provided to all collaborators via HTTP; applications are deterministic and it is possible to deliver the same inputs to all replicas, ensuring replica consistency; replicated proxies can access centralized remote external resources to ensure that all replicas receive the same data; newcomers can receive an image of the application's process, data, and state. However, replication is not suitable for applications that depend on the time at which an input arrives because differing network lags will cause inputs to arrive at different times on each replica.

There are several approaches to ensuring consistency and managing concurrent inputs for replicated synchronous systems. Consistency is assured if each deterministic replica receives the same inputs in the same order. **Serialization** is the process of imposing a total ordering on events which

change the data so that each copy receives events in the same order. One approach to avoiding conflicts between concurrent users is **locking**, where only one user at a time can change the shared data. At the coarsest granularity, **floor control** allows only one collaborator to hold a lock on the entire shared application. Floor control is commonly used in collaboration-transparency systems. The disadvantage of floor control is that it prohibits participants from working simultaneously in the application. Collaboration-aware systems often use a finer-grained locking. A form of optimistic serialization, called **operational transformation**, allows concurrent data changes, and avoids conflicts between operations while providing fast local application response to the change. Each replica *transforms* incoming remote operations so that they are consistent with its own set of previously applied operations.

Applications commonly acquire input from and write output to **external resources**, which are input–output sources other than display output or interactive user inputs, such as files, sockets, and system time. External resources raise difficulties for a replicated system. How can replicas access an external resource that resides on only one collaborating host? The system must ensure that each replica receives the same data from external input resources. Additionally, replicating output is not always feasible or desirable.

Few groupware toolkits provide facilities to ease the use of external data resources. A groupware developer must explicitly ensure that the data are distributed. Common programming approaches are described.

It is possible to access external resources in a collaborative application in the same way as in a conventional single-user application. One approach is to automatically replicate a resource, such as a file, so that each replica has an identical copy. However, physical replication is not always possible because of different file name conventions on heterogeneous hosts (e.g., Macintosh versus UNIX file systems), the resource is closely tied to the underlying system (e.g., system time), or multiple instances of the resource are not allowed (e.g., a server that allows only one network connection).

A semi-replicated approach to transparent replication of external resources is presented. The external resource object resides at a single location and is accessed via replicated **proxies** that multiplex input and output to and from the single instance of the external resource. The problem of creating proxies after an application is shared is described and a solution is presented. An implementation of this solution has been applied to the standard Java 1.1 class library which required modifying some standard Java classes, such as `java.io.FileInputStream` and `java.lang.System`.

Each modified external resource class now contains a reference to either a proxy or a local resource object. When such an external resource object is used in a shared application, the external resource object accesses its proxy object, otherwise the local resource is accessed. A resource object can switch from using a local resource to a remote proxy, if the application using it is later shared. The chapter describes a problem related to using this approach for system time. By incorporating this approach, applications shared via a replicated collaboration-transparency system can transparently access external resources, and groupware toolkits can further facilitate the development of synchronous collaborative applications.

Chapter 4

Conventional Collaboration Transparency

Section 2.4.2 describes several existing collaboration-transparency systems, which I refer to as **conventional** because they share the same basic implementation and resulting limitations described in this chapter. Conventional collaboration transparency systems use a centralized architecture to share an application (Section 3.2.1): inputs from users at multiple sites are merged and delivered to a single instance of the application. Any change to the graphical output of the application is broadcast to all participants.

Prior research in collaboration transparency has focused on feasibility and technical problems, not on the usefulness of these systems for meaningful collaborative work. Researchers have made scattered complaints about conventional collaboration transparency, but no thorough critique exists. I have found that conventional collaboration-transparency systems do not adequately support collaboration. Conventional collaboration-transparency implementations do not use network resources efficiently, and they impose an inflexible, tightly coupled style of collaboration because they do not adequately support the following important groupware principles: concurrent work, relaxed WYSIWIS, group awareness, and inherently collaborative tasks. This chapter enumerates the shortcomings and explores their causes.

4.1 Deficiencies

This section describes the deficiencies of conventional collaboration-transparency systems in key groupware principles: no concurrent work, strict WYSIWIS, limited group awareness, inherently collaborative tasks. In addition, this section describes conventional systems' inefficient use of network resources.

4.1.1 No Concurrent Work

Under conventional collaboration-transparency systems, only one participant at a time can work in the shared application. The controlling participant is said to *hold the floor.* Floor control policies range from **explicit,** where participants deliberately request and release control of the application, to **implicit,** where the collaboration-transparency system automatically requests control for participants when they begin to generate input events. Crowley *et al.* (1990) describe four variants of floor control policy: explicit request with explicit grant; explicit request with implicit grant; implicit request with explicit grant; and implicit request with implicit grant. I use the term *implicit floor control* only for the last policy because requesting and granting are both automated, and the term *explicit floor control* for the first three policies because they all require direct action by the users.

Even when the interface is easy to use, explicit control diminishes collaborators' feelings of involvement in the collaboration because it makes the turn taking apparent. This effect was observed in the study reported in Section 6.1. Under implicit control, on the other hand, collaborators do not request and release control directly, and may not even be aware that only one of them at a time may interact with the system. Nevertheless, under both models, only one collaborator at a time may actually use the application.

The degree to which collaborators work closely together versus independently is referred to as **tight** versus **loose coupling** (Dewan & Choudhary, 1991a). During the course of a collaborative session, coworkers engage in varying patterns of collaboration: tightly and loosely coupled, synchronous and asynchronous, private and shared, scheduled and opportunistic. During synchronous collaboration, floor control imposes a tightly coupled mode of collaboration because while one participant is in control, the others may only observe the work. Thus, floor control limits the collaborators' ability to work in parallel. Many researchers have cited the imposition of tightly coupled work as

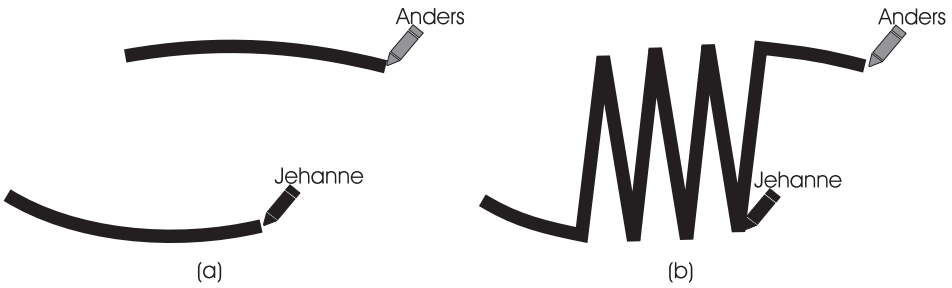


Figure 4.1: (a) Desired and (b) actual output caused by event interleaving.

a serious problem with conventional collaboration transparency (Prakash & Shim, 1994; Reinhard *et al.*, 1994; Schuckmann *et al.*, 1996).

Cause: Application-level Locking

Floor control is used to avoid potential problems associated with interleaving input events from multiple collaborators. Collaboration-transparency systems must maintain the intended behavior of the shared application. If input events from multiple users are simultaneously applied to the shared application, event streams for non-atomic events (i.e., mouse drags) may become interleaved, and the result may not be what the users intended.

For example, in Figure 4.1, Anders and Jehanne are sharing a drawing application, and they simultaneously attempt to draw separate freehand curves by dragging their mouse cursors. However, the location of the previous mouse-drag event for Anders is different from that of Jehanne. Unless some protection mechanism is enforced, Anders and Jehanne's mouse-drag events are interleaved and delivered to the application. Figure 4.1a shows what was intended. Figure 4.1b shows the potential result of the interleaved mouse-drag events. The application has drawn a line between the alternating mouse positions of both Jehanne and Anders, instead of between the sequential positions of Jehanne's mouse, followed by sequential positions of Anders' mouse. This is an unexpected and undesirable result.

To avoid such conflicts, conventional collaboration-transparency systems use floor control, restricting input to only one participant at a time. Floor control is essentially locking at the coarsest possible granularity – the whole application. In contrast to application-wide locking, a collaboration-aware application may also use locking at a finer granularity to permit safe, simultaneous ma-

nipulation of different portions of the shared data, as described in Section 3.2.3. Such fine-grained locking supports a more loosely coupled collaboration than floor control allows.

4.1.2 Strict WYSIWIS

In conventional collaboration-transparency systems, participants have the same view of the shared application in a manner referred to as strict **What You See Is What I See (WYSIWIS),** where the participants see exactly the same view of the shared application at the same time (Stefik *et al.*, 1987). This also contributes to the imposition of tightly coupled collaboration, since participants must view the same portion of the shared data simultaneously. In contrast, collaboration-aware applications often support multiple modes of collaboration by relaxing WYSIWIS so that participants can simultaneously view different portions of shared data (Gutwin *et al.*, 1996b; Smith, 1992).

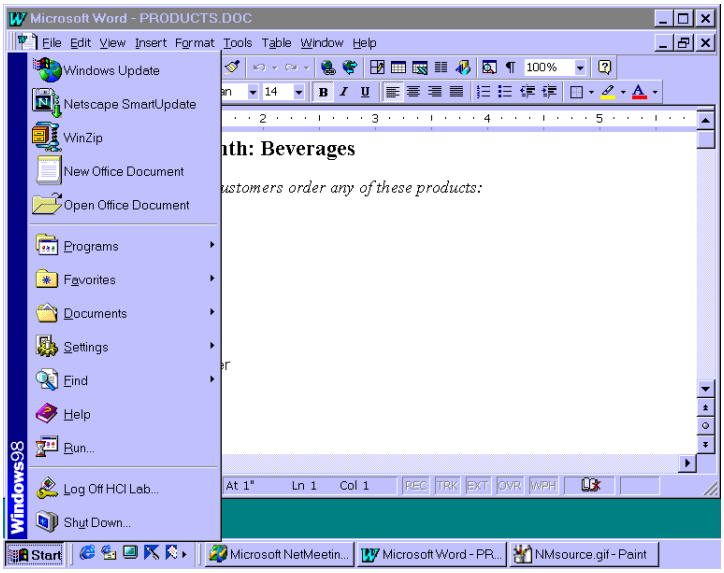
Cause: Centralized Architecture

The architecture used by all conventional collaboration-transparency systems is a centralized, display-broadcasting architecture (Section 3.2.1). In this approach, participants receive only a graphical depiction of a central instance of the shared application. A centralized architecture ensures that the state of the shared application is consistent among the participants because there is only one instance of the application. Unfortunately, because centralized display broadcasting leads to sharing the application in a strict WYSIWIS manner, centralized display broadcasting becomes a principle cause of conventional collaboration transparency's poor support of groupware usability principles.

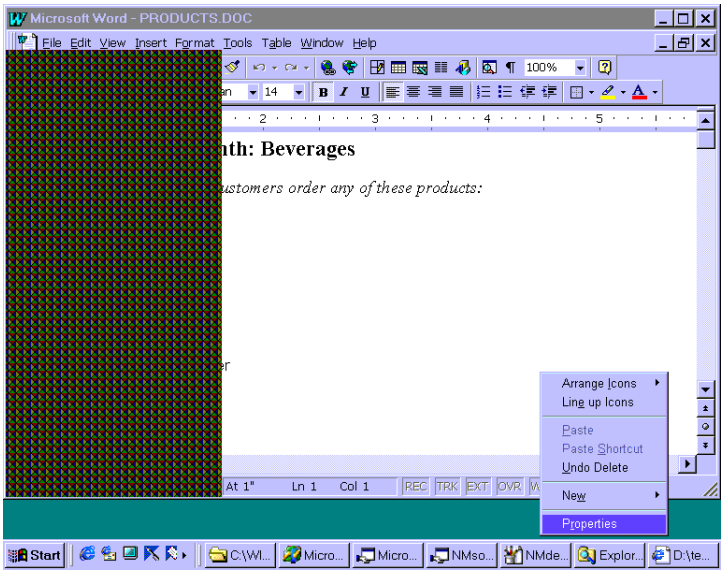
In Microsoft NetMeeting, display broadcasting causes such severely strict WYSIWIS that remote views do not display portions of the application that are obstructed by windows or menus on the central host. Figure 4.2 shows an example of strict WYSIWIS in an application shared with Microsoft NetMeeting.

4.1.3 Limited Group Awareness

Schuckmann *et al.* (1996) point out that conventional collaboration transparency does not provide detailed information concerning participants' actions and locations. Typically, conventional collaboration-transparency systems provide awareness information only by means of **telepointers,** representations of remote users' cursor positions (telepointers are described in Section 2.3.1).



Shared application on the central host



View of the shared application on a remote host

Figure 4.2: Strict WYSIWIS under NetMeeting. The top figure shows the shared application (Microsoft Word) at the central host. A menu overlays the shared application. The bottom figure shows the shared application on a remote host. The segment of the application that is obstructed by the menu on the central host is also not visible on remote hosts. Note that the occlusion only occurs when it originates from the central host. For example, the popup menu in the lower right of the remote host does not occlude the view of the application at any other host.

In some collaboration-transparency systems, like Microsoft NetMeeting, one pointer is shared among all participants and is controlled by only one at a time (the one in control of the application). Therefore, the only awareness information provided is the cursor location of the controlling participant. No information is available about other collaborators' cursor locations. Not all collaboration-transparency systems have only one shared pointer. Sun's ShowMe SharedApp, for example, displays a unique telepointer for each participant.

The single shared pointer in Microsoft NetMeeting has a related problem. Microsoft NetMeeting uses the system pointer (i.e., the "hardware" cursor) of each host for the shared telepointer. This prevents collaborators from working in windows not associated with the shared application because their pointer will move independent of their mouse. Thus, NetMeeting further prevents independent work, and imposes a tightly coupled style of collaboration, as described in the previous section.

Cause: Strict WYSIWIS

In fact, telepointers do provide sufficient information about participants' locations for a conventional collaboration-transparency system, since participants share a strict WYSIWIS view of the application. However, G. Smith (1996a) as well as Greenberg and Roseman (1996) demonstrated that in environments that allow users to have simultaneously independent views of the shared application, called **location-relaxed WYSIWIS,** simple telepointers are inadequate. Section 2.3 describes other multi-user interface elements that provide more detailed awareness information than simple telepointers.

4.1.4 Task Support

Another fundamental distinction between collaboration transparency and awareness is in the type of tasks they can support. Collaboration-transparency systems are only suitable for tasks that extend from individual to collaborative performance, such as text editing. In addition to supporting such individual-to-collaborative types of tasks, collaboration-aware applications can support inherently collaborative tasks, such as voting by secret ballot, group decision making, and some multi-player games. For example, consider a single-user role-playing adventure game. The player manipulates a single character in the adventure. Sharing the single character via collaboration transparency is not equivalent to a Multi-User Domain (MUD) where each player manipulates an individual character with unique properties.

Cause: single-user application sharing

By definition, collaboration-transparency systems share existing, single-user applications. These applications were created to support one individual at a time. A collaboration-transparency system supports collaboration externally from the application, whereas a collaboration-aware application can provide internal support for cooperative work, including information, such as participant roles, and behavior, such as enforcing access policies.

4.1.5 Network Usage

In addition to deficiencies in groupware usability, conventional collaboration-transparency systems compare poorly to collaboration-aware applications in terms of efficient use of network resources.

Cause: Centralized Architecture

The primary cause of the higher network demand for conventional collaboration transparency is the centralized display broadcasting architecture used by all conventional systems. Section 3.2.4 describes the higher network demand of centralized architectures over that of replicated architectures.

Cause: Lack of Knowledge of Application Semantics

Another reason that a conventional collaboration-transparency system uses higher network bandwidth is lack of knowledge of a shared application's semantics, which prevents the system from optimizing network communications. A conventional collaboration-transparency system cannot determine how a shared application will respond to a given user input event. Therefore, the system must send *all* inputs to the application even though some might be safely omitted. For example, in some applications, it might be acceptable to send no mouse position updates, only mouse button press and release messages. However, because mouse position information may affect the state of some applications, the system forwards all position updates.

Developers of collaboration-aware applications, on the other hand, can take advantage of their knowledge of the application's semantics, transmitting only the information needed to maintain state consistency among the replicas and provide acceptable performance (Dourish, 1996a).

Table 4.1: Comparison of collaboration transparency versus awareness.

	Advantages	**Disadvantages**
Collaboration Transparency	• No extra development cost • Collaborate using legacy applications	• No concurrent work • Strict WYSIWIS • Limited group awareness information • Single-user tasks • Higher network bandwidth
Collaboration Awareness	• Concurrent work possible • Relaxed WYSIWIS • Detailed group awareness information • Can support inherently collaborative tasks • Lower network bandwidth	• Extra development cost • Potential advantages not always applied

4.2 Awareness versus Transparency

Collaboration awareness and transparency offer widely different support for collaboration. Each limitation of conventional collaboration transparency contrasts directly to a capability seen in *some* collaboration-aware applications and toolkits. Table 4.1 summarizes the characteristics of each approach.

Although collaboration-aware systems may appear generally superior to conventional collaboration-transparency systems, it is important to realize that not all collaboration-aware systems include the potential advantages. For example, only five of the thirteen groupware toolkits surveyed in Section 2.4.1 provide even the most rudimentary awareness information with a telepointer, and only two include radar views. The omission of multi-user components from a groupware toolkit means a developer must implement directly support for desired groupware usability features. Implementing multi-user interface components is not simple, which may explain why toolkit creators do not include them. Telepointers, for example, are particularly difficult to implement because common windowing systems (i.e., Macintosh, X, and MS Windows) only support one screen pointer. Requiring a developer to create standard multi-user components is counter to a groupware toolkit's fundamental goal of facilitating groupware development.

In terms of development cost, conventional collaboration transparency has the advantage over collaboration awareness because conventional collaboration transparency requires no modification

to single-user applications. Therefore, when an existing single-user application is suitable for a collaborative task, it makes sense to consider collaboration transparency. Generally, however, well designed collaboration-aware applications provide better support for collaboration and are the only choice for inherently collaborative tasks.

The purpose of this chapter is not to argue against collaboration transparency as a useful technology, but to point out deficiencies and their causes in current implementations. The next chapter describes a new implementation approach to collaboration transparency that provides many capabilities previously seen only in collaboration-aware systems.

4.3 Summary

This chapter describes deficiencies and their causes in conventional collaboration-transparency systems in contrast to capabilities seen in some collaboration-aware applications. Conventional collaboration-transparency implementations impose an inflexible, tightly coupled style of collaboration because they do not adequately support the following important groupware principles: concurrent work, relaxed WYSIWIS, group awareness, and inherently collaborative tasks. The deficiencies and causes are summarized next.

(1) To avoid conflicts arising from interleaved nonatomic events (i.e., mouse drags), conventional collaboration-transparency systems use floor control to prevent input from more than one person at a time. Such application-wide locking limits participants' ability to collaborate in parallel and diminishes their feelings of involvement. (2) In conventional collaboration-transparency systems is that all participants see exactly the same view at the same time, called strict WYSIWIS because the centralized application sends the same display information to all participants. (3) Conventional collaboration-transparency systems only provide rudimentary awareness information by means of telepointers. Many systems do not even provide a telepointer for each collaborator. Additionally, telepointers alone are inadequate to convey users' locations when WYSIWIS is relaxed. (4) collaboration-transparency systems are only suitable for tasks that extend from individual to collaborative performance, such as text editing, whereas collaboration-aware applications can support tasks that are inherently collaborative, such as voting by secret ballot.

In addition to usability deficiencies, conventional collaboration-transparency systems generally require higher network bandwidth than collaboration-aware applications. This is primarily because

conventional collaboration-transparency systems use a centralized, display-broadcasting architecture, whereas collaboration-aware applications are typically replicated. Another reason for higher network usage is that a conventional collaboration-transparency system cannot determine which input events can be safely modified, delayed, or dropped, and must therefore transmit all events. Developers of a collaboration-aware application, however, can optimize the application's communications.

In terms of development cost, collaboration transparency has the advantage over collaboration awareness because, at the application level, collaboration transparency has no development cost. However, a well designed collaboration-aware application will generally provide better support for collaboration than a conventional collaboration-transparency system. On the other hand, not all collaboration-aware applications are well designed or include the potential advantages. Additionally, for many tasks, there may not be a suitable collaboration-aware application while a powerful single-user application may exist. Therefore, in many cases, sharing an existing application via collaboration transparency is a useful means of computer support for a synchronous collaboration. The purpose of this chapter is not to argue against collaboration transparency as a useful technology, but to point out deficiencies and their causes in current implementations, leading to the alternate approach presented in Chapter 5.

Chapter 5

Flexible Collaboration Transparency

As discussed in Chapter 4, conventional implementations of collaboration transparency impose an inflexible style of collaboration because they do not adequately support key groupware principles: concurrent work, relaxed WYSIWIS, detailed group awareness, inherently collaborative tasks, and efficient network use. Despite these problems, collaboration transparency remains attractive because of the ability to share existing, legacy applications. This chapter describes a new implementation approach that adds support for fundamental groupware principles in collaboration transparency, providing support for multiple styles of collaboration. I term this model of application sharing, *flexible* collaboration transparency because it allows collaborators to transition among various patterns of collaboration. The approach to flexible collaboration transparency combines a replicated architecture and run-time replacement of particular single-user interface objects with multi-user extensions.

This chapter first enumerates the technical and usability specifications for flexible collaboration transparency. Section 5.2 describes a general approach that satisfies the requirements. This approach has been implemented in a prototype system called Flexible JAMM (Java Applets Made Multi-user), which is described in Section 5.3. The chapter is summarized in Section 5.4.

5.1 Specifications

Before describing my approach to flexible collaboration transparency, this section lists the desired features of such a system.

5.1.1 Technical Specifications

In terms of sharing existing applications, the collaboration-transparency system should support the following technical specifications.

- The underlying platform's Application Programming Interface (API) should remain unchanged. That is, the interface and functionality of any modifications to the application's run-time environment should appear unchanged to the developer, and no modification to the application should be necessary. This requirement is fundamental to the notion of transparency and provides collaboration transparency with its primary benefit: no additional application development cost to allow shared use of the application.
- The collaboration-transparency system should not impose any performance cost on an application when it is not being shared. That is, any performance penalty caused by the modifications to the underlying platform should apply only to applications when they are shared, and not to any non-shared applications that may be running concurrently.
- Minimal network bandwidth should be used. This is especially important to support Internet-wide sharing.

5.1.2 Usability Specifications

In addition to the preceding technical specifications, a flexible collaboration-transparency system should support the following usability specifications.

- The collaboration-transparency system should permit **unanticipated sharing**. A person should be able to initiate sharing of an application at any time during that program's execution. In contrast, some application-sharing systems, such as SharedApp (Sun Microsystems, Inc., 1998), require that shared applications be launched from within the system. Thus, users must anticipate the need to share an application before they start it, which discourages spontaneous collaborations.

- **Late-joining** to a session should be allowed. Collaborators may want to join an on-going session, and late-joiners to a session should receive the current state of the shared application. This is a corollary to unanticipated sharing.
- The system should support multiple styles of collaboration, allowing collaborators to work together closely or independently. To that end, the system should support the following features. Addressing these goals in collaboration transparency is the primary focus of this dissertation.
  - Simultaneous work.
  - Implicit floor control where "turn taking" is required.
  - Location-relaxed WYSIWIS.
  - Detailed group awareness information.

5.2 General Approach

The technical and usability deficiencies found in conventional collaboration-transparency systems are due to their common implementation approach: a single, centralized instance of the shared application with display broadcasting (Section 3.2.1). This section describes an alternate approach, which is to use a replicated architecture and replace selected single-user interface objects with multi-user extensions dynamically. This approach to collaboration transparency provides features previously seen only in collaboration-aware applications.

5.2.1 Replicated Architecture

The first characteristic of this approach is the use of a replicated architecture. Section 3.2.5 describes advantages and disadvantages associated with using a replicated architecture for collaboration transparency. Briefly, replication requires less network bandwidth than conventional centralized systems, but replication introduces the possibility that replicas of the shared data may become inconsistent (Crowley *et al.*, 1990; Lauwers *et al.*, 1990; Minenko, 1998). We have found that a replicated architecture can support transparent sharing of many, though not all, single-user applications (see Section 3.2.5 and (Begole *et al.*, 1997b)).

In addition to network efficiency, replication allows concurrent work and relaxed WYSIWIS. Because each collaborator has an independent copy of the shared application, the interface states can differ while ensuring the application's replicated data remain consistent. Prior researchers have emphasized the network efficiency of replicated architectures for collaboration transparency, but did not report the potential usability advantages (Crowley *et al.*, 1990; Lauwers *et al.*, 1990; Minenko, 1998). Collaboration-aware applications and toolkits, on the other hand, often support loosely coupled collaboration by using a replicated or semi-replicated approach (see Section 2.4.1 and Table 2.3 for a summary of collaboration-aware systems).

5.2.2 Single-User Interface Object Replacement

The second characteristic of the new approach is the run-time replacement of selected single-user interface objects with multi-user equivalents. Today, user interfaces are commonly created by combining objects from an object-oriented toolkit, such as the Java Abstract Window Toolkit (AWT), Microsoft Foundation Classes (MFC), and wxWindows (Smart, 1998). These libraries include often-used interface elements, such as the classic Windows, Icons, Menus, Pointers, and Scrollbars (WIMPS). Many compound interface components are also widely available, such as scrollable lists, editable text areas, and dialog windows.

For example, Figure 5.1 shows a document-editing application that contains a single-user scrollpane which embeds a document. It is possible to replace the scrollpane object without modifying the application source code. For example, in Figure 5.2 the scrollpane has been replaced by a multi-user version that adds a radar view of the document indicating each collaborator's scroll position. Radar views are also discussed in Section 2.3.2. These screen images were taken from the prototype flexible collaboration-transparency implementation described in Section 5.3.

The object replacement occurs at run time. The multi-user version must implement the same interface for the replaced object as the original version did, so that the replacement appears to the application to be an instance of the original. Therefore, the replacement is transparent to the single-user application and its developers. For example, Figure 5.3a shows the object diagram of a simple application with a single-user scrollpane object, which is replaced in Figure 5.3b by a multi-user scrollpane and an associated radar view.

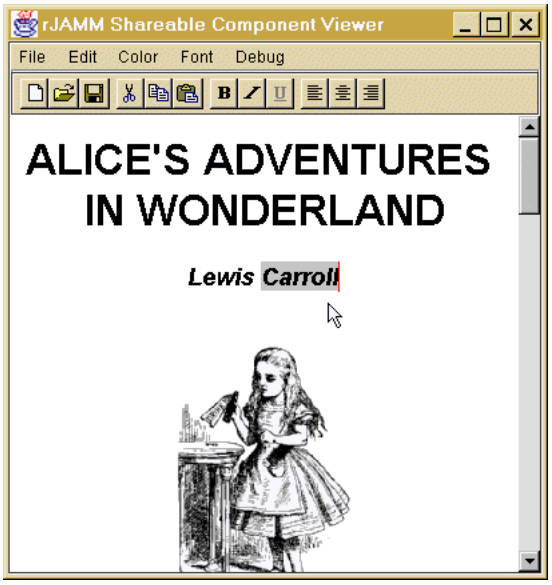


Figure 5.1: A single-user editor, called Stylepad. A scrollable panel object contains the document.

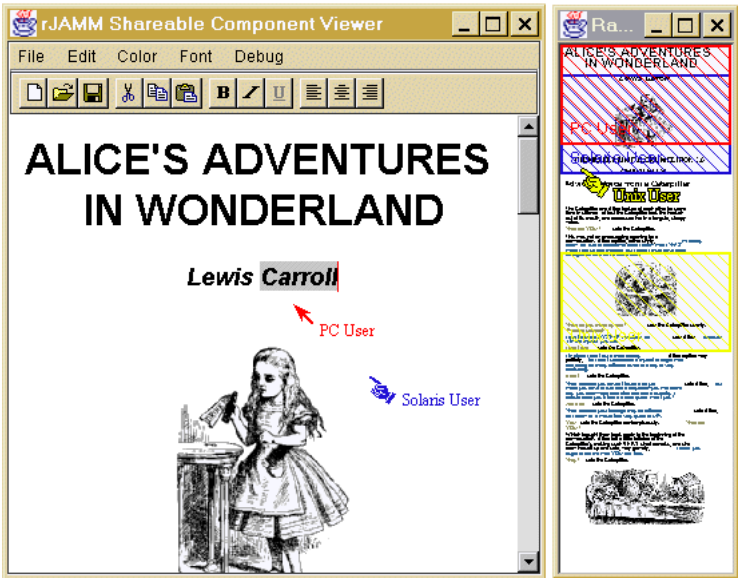


Figure 5.2: Stylepad shared via Flexible JAMM. The single-user scrollable panel is replaced by a multi-user version, which displays a miniature view of the document in the window on the right, called a *radar view*. A uniquely colored and shaded rectangle indicates each participant's scroll position. Three participant positions are shown here: *PC User* and *Solaris User* overlap near the top while *Unix User* is near the center.

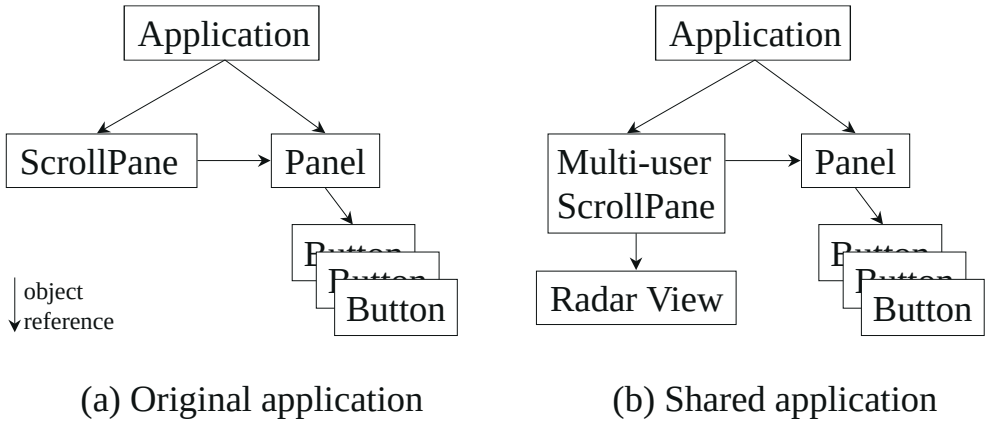


Figure 5.3: The objects of a simple application. (a) The original application (left) contains a single-user scrollpane which embeds a panel containing a set of button objects. (b) A multi-user scrollpane has replaced the original scrollpane and added a radar view object. The multi-user scrollpane provides the same user functionality as the original plus multi-user capabilities.

Semantic Knowledge

A flexible collaboration-transparency system has knowledge of the semantics of input events that occur within the replaced interface objects. For example, clicking the mouse in a scrollbar causes the scroll position to change. With this information, the system can optimize its use of network resources by minimizing the size and number of update messages distributed to replicas. For example, the system can choose not to distribute events which simply change the scroll position, thus allowing each replica to have an independent scroll position.

Without knowing how an event will be used by an application, a conventional collaboration-transparency system cannot determine whether the event may be safely modified, delayed, or dropped. Thus, conventional collaboration-transparency systems distribute all input events. Knowledge of low-level input semantics allows a flexible collaboration-transparency system to provide relaxed WYSIWIS, implicit floor control, concurrent work, and fast local application response, described later in Section 5.3.6 in the context of the prototype implementation.

5.2.3 Platform Requirements

The new approach requires the application platform to have the following capabilities: process migration, run-time object replacement, dynamic binding, and the ability to intercept and introduce low-level user input events. The need for each of these requirements is described next.

Process Migration

One of the usability requirements outlined in Section 5.1.2 is to allow new participants to join an ongoing collaboration. In a centralized architecture, this is straightforward because there is only one instance of the shared application (Chung & Dewan, 1996; Chung *et al.*, 1993). In a replicated system late-joiners must receive a copy of the application and the state in which it currently exists.

Transferring a process from one execution environment to another is called **process migration**. Execution environments may differ physically (different machines) or temporally (same machine, different time). Typically, the process stops execution in some state, is transferred to a new environment, and continues execution from the previous state. In a replicated collaborative system, the initial process continues to execute along with the transferred process.

There are two general approaches to process migration: **event replay,** where the system logs all state-changing events during the session and replays them to a new instance of the process, and **image-copy,** where an image of the process in memory is copied (Chung & Dewan, 1996). Events in the event-replay scheme may be low-level user inputs, such as mouse and keyboard inputs, or application-specific semantic updates, such as a database queries. Event replay has these disadvantages: the migration time depends on the length of the execution, and potentially expensive operations are executed multiple times. Typically, events are replayed as quickly as possible, but if the application state depends on the precise time at which an event occurred, then the replay must be slowed to correspond with the original event timings. Because of these problems, Chung and Dewan (1996) concluded that the time to migrate a process using event replay is approximately an order of magnitude greater than image-copy schemes. However, image copy requires support by the execution platform. The source platform must provide information about the process' address space, execution state (i.e., program counter, execution stack, and register values), and connections to system resources (e.g., files, other processes, and network connections). The destination platform must be able to recreate the process state and restart execution at the appropriate point. Few current platforms support the image-copy approach, whereas event-replay is usually feasible.

In the future, image copy may be feasible on more computing platforms than currently. Process migration has received much attention by the research community, but the commercial uses of such technology are only recently being appreciated. Distributed computing standards, such as the Common Object Request Broker Architecture (CORBA) (Mowbray & Zahavi, 1995), Microsoft Distributed Component Object Model (DCOM) (Brown & Kindell, 1996), and Java Remote Method Invocation (RMI) (Wollrath *et al.*, 1996), have given software creators and users a taste for Internet-wide computing. However, under the current distributed computing frameworks, while data travel freely among many hosts, process locations are fixed. As wireless, mobile computing becomes increasingly common, developers will find an increasing need for process migration. To address this need, future distributed computing standards, such as Sun's recently announced Jini architecture (Kelly & Reiss, 1998), will likely include support for process migration.

Run-time Object Replacement

As described previously, a fundamental characteristic of the new approach is the replacement of selected objects after the application has begun execution. The replacement must occur during execution to support the goal of unanticipated sharing, described in Section 5.1.2. To perform this replacement transparently (i.e., without modifying the application source code), the execution platform must provide a replacement mechanism that the collaboration-transparency system can use. Like process migration, dynamic object replacement is not currently supported by many platforms, but is feasible on the Java platform, as described later in Section 5.3.1.

Dynamic Binding

Transparent object replacement also depends on **dynamic binding**, the run-time resolution of a function invocation or data access. Dynamic binding allows an instance of one class to function in place of another, provided the replacement implements the same interface (i.e., the accessible data and behavior of an object). The objects may share an interface explicitly by inheriting from a common ancestor, as in Java and C++, or implicitly by responding to the same messages or method invocations, as in Smalltalk and Self. In many object-oriented environments, such as Smalltalk and Java, dynamic binding is the norm, whereas in others, such as C++, dynamic binding must be explicitly programmed.

Dynamic binding is necessary for flexible collaboration transparency so that one object can take the place of another without modifying the application. Because the application source code is not modified, the substitution is *transparent* to the shared application and its developers.

User Input Event Interception and Introduction

A collaboration-transparency system must be able to intercept user input events. In a centralized implementation, the events from each participant are sent to the single instance of the shared application. In a replicated implementation, the events must be intercepted and distributed to all replicas *before* they are applied to the local application. The events must be applied in the same order on all replicas. Therefore, locally generated events must be held back until the system determines the total order of locally and remotely generated events. Equally important, the collaboration-transparency system must be able to introduce remotely generated events into the local event queue.

5.3 Prototype: Flexible JAMM

Flexible JAMM (Java Applets Made Multi-user) is a replicated-architecture collaboration-transparency system that incorporates the approach of dynamically replacing single-user objects with collaboration-aware versions (see also (Begole *et al.*, 1998b)). Flexible JAMM's distribution architecture is based on an earlier replicated collaboration-transparency system, called JAMM (Begole *et al.*, 1997a; Begole *et al.*, 1997b), which we developed at JavaSoft during in the Fall of 1996. Flexible JAMM extends the capabilities of that system by introducing support for multiple collaboration styles. Figure 5.4 shows a single-user text editor shared via Flexible JAMM. Other applications shared via Flexible JAMM appear elsewhere in this dissertation: Figure 5.2 on page 79, and Figure 6.12 on page 133.

Aside from any inherent programming features, Java is especially attractive for writing collaborative software because it simplifies distribution of software and largely eliminates platform compatibility issues. While these characteristics are useful for software in general, they are critical for collaborative software because each collaborator must have access to the software on their platform. Additionally, the Java environment supports the requirements listed in Section 5.2.3, as described next.

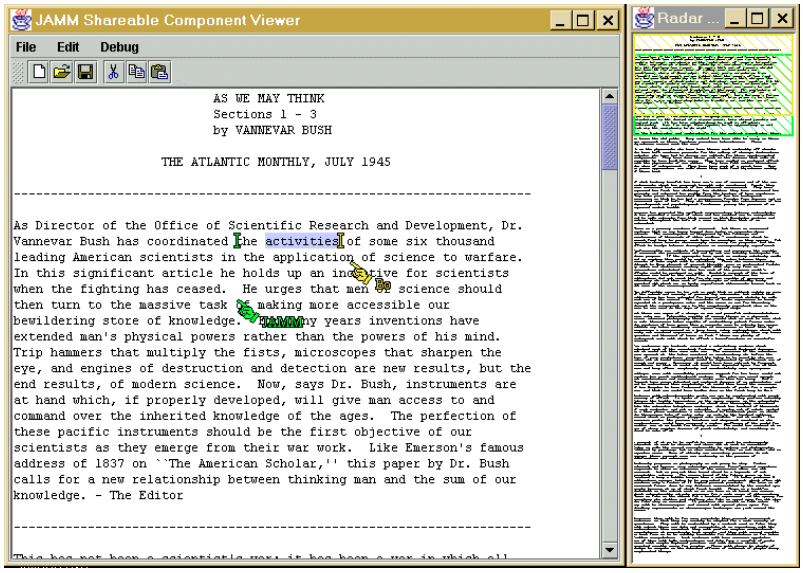


Figure 5.4: A single-user text editor application, called Notepad, shared via Flexible JAMM.

5.3.1 Process Migration

Flexible JAMM uses an image-copy scheme provided by Java Object Serialization (JOS) to send the current state of a shared application to newcomers. JOS is part of the core Java 1.1 class library and provides the means to copy an object's current state for storage or network transfer.

JOS provides the basic capability needed to migrate applets to new participants. However, some JOS characteristics, while practical in normal usage, are too restrictive for *transparent* migration. One limitation is that JOS does not transfer the class (static) variables which may be considered part of an object's state. Class variables are omitted so that when an object is received via JOS, the already-existing values for class variables are not overwritten by new values. In the initial version of JAMM (Begole *et al.*, 1997b), we modified JOS so that the whole state of the application object, including class variables, was transferred. To prevent overwriting already existing class variables on the receiving end, the replica's classes were loaded in a separate **ClassLoader,** which confines the scope of a class and its variables (Arnold & Gosling, 1996).

In Flexible JAMM, however, I did not modify JOS to transfer class variables. There were two reasons for this. First, the modification requires changes to the native platform implementation of the Java classes, which requires porting, or at least recompilation, of JOS on each platform to which

the Java virtual machine is ported. Thus, the first prototype JAMM forfeited a key advantage of using Java: platform independence. Second, JOS now provides a means to explicitly transfer (or otherwise restore) class variable state.

JOS is not transparent to application developers. For an application to be copied via JOS, the developer must choose to make all application classes **serializable**. In most cases, implementing serializability is straightforward. Much of the core Java library is serializable and classes that inherit from serializable classes are implicitly serializable. Many standard Java classes are not serializable, however, such as `java.awt.Image` and `java.lang.Thread`. Objects of non-serializable classes contain platform-specific data that may not be transferable to a remote Java Virtual Machine (JVM), that is potentially running on a different platform. In some cases, it is possible to transform the object into a platform-independent format, transfer this format, and instantiate a platform-dependent copy at the receiving site. Flexible JAMM takes this approach for two otherwise unserializable classes: `java.awt.Image`, and `java.awt.FontMetrics`.

Objects of the `Thread` class present a more difficult problem. Although the `Thread` class provides high-level execution state information, such as *running, stopped,* etc., it does not provide precise, low-level information about the current execution, such as the program counter and execution stack. A migrated process replica must have this information to restart the process. To address this difficulty, Flexible JAMM's strategy is to use the `java.applet.Applet.stop()` and `Applet.start()` methods, which a web browser calls to notify an applet when the browser leaves and revisits a page. The intended behavior for these methods is that the applet stop its running threads and restart them, respectively. While many developers correctly implement these methods, there is no guarantee that all will. For example, `Applet.stop()` may not clean up all threads and `Applet.start()` may incorrectly *re*initialize variables. Using `Thread.suspend()` and `Thread.resume()` is insufficient because we cannot determine the suspended execution state of the thread, which is necessary to resume it correctly at the receiving end. Another strategy might be to call `Thread.stop()` and `Thread.start(),` but this also depends on the developer having implemented restartable threads. Therefore, Flexible JAMM relies on applets conforming to the `Applet.start()` and `Applet.stop()` design.

If a developer-defined class does not inherit from a serializable class, the developer must explicitly make it serializable. Often, simply declaring a class to implement the `java.io.Serializable` interface is all that is necessary. However, some classes are not so easily made serializable. For

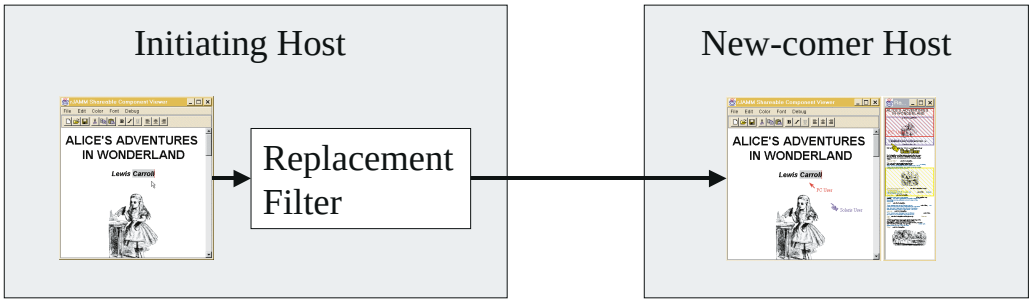


Figure 5.5: The replacement filter replaces objects sent from the initiating host to the newcomer host. When the application is first shared, the initiating host is treated as a newcomer.

example, a common problem is that an interface object will get and hold a reference to the top-level, machine-specific window in which it is contained. Suppose such an object is sent from host A to host B. When the object is reconstructed on host B, the object will be placed in a new top-level window on B, but the object will refer to a window on A and may attempt to access it, causing a run-time error. The solution is not to hold a reference to a top-level window, but look up the top-level window as needed. In this case, and most others, serializability is not difficult to implement. However, a developer may not foresee any need for serializability and therefore neglect to include it.

In the future, Java developers may increasingly make applications serializable. One reason is that serialization is required for a class to be considered a Java Bean (Hamilton, 1997), which is a popular component architecture for the Java platform. Another reason is that the Jini distributed computing architecture relies on JOS for process migration.

Because of the limitations described in this section, JOS does not fully support *transparent* image copy. The Flexible JAMM web site contains a set of instructions to make existing applications serializable (Begole, 1997). Section 6.2 describes an informal evaluation of the instructions. Flexible JAMM is able to use JOS for process migration, requiring little or no modification to existing application source code.

5.3.2 Run-time Object Replacement

JOS also provides the object-replacement capability needed for flexible collaboration transparency. For each object reachable from the initial target object during serialization, JOS provides a mechanism to inspect and replace it with an instance of an equivalent class (i.e., a subclass or an

implementation of the same interface). The replacement occurs at run time when the application is initially shared, and does not inspect or modify application source code. Thus, the replacement of one object with another is transparent to the application and its developers.

Flexible JAMM introduces a replacement filter into the outgoing stream of objects sent via JOS, as seen in the diagram in Figure 5.5. When the application is first shared, Flexible JAMM treats the initiating host as a newcomer. Therefore, the initiating host receives the multi-user version of the shared application.

The replacement filter uses the following procedure to make a replacement (Figure 5.6 lists the critical code Flexible JAMM uses for the replacement process). As each object passes through the output stream, the replacement filter compares the object's class to each entry in a table of replaceable classes (`classCheck`). The replacement table contains a dictionary of each replaceable class and its replacement class (the table is initialized in lines 6 and 7). If the object is not an instance of a replaceable class, or it is an instance of a replacement class (meaning the single-user object has already been replaced by a multi-user object) (line 21), the filter returns the object unchanged. Otherwise, the filter creates a replacement object (lines 24–29), which is returned in place of the original (lines 30 and 40). When this stream of objects is received and reconstructed, all references to the original are updated to refer to the replacement.

Each replacement class must have a constructor that takes an instance of the original class as its single parameter. Flexible JAMM instantiates replacement objects by calling the constructor with that signature. That constructor is found in lines 24–26 and invoked in line 29. The replacement constructor must initialize the replacement object to the same state as the original.

5.3.3 Dynamic Binding

Java uses dynamic binding and run-time type checking for each method invocation and data access. Therefore, after an object is replaced, the replacement will be used at the next invocation or access. The replacement must be a subclass of the original or implement the same interface(s) as the original. In Flexible JAMM, replacement classes are subclasses of the original, as described previously in Section 5.3.2.

```
 1. public class rJammObjectOutputStream extends ObjectOutputStream {
 2.   ClassLoader loader = new rJammClassLoader();
 3.   private Hashtable classCheck = new Hashtable();
 4.   private void setupClassTable() {
 5.     try {
 6.       classCheck.put(loader.loadClass("com.sun.java.swing.JScrollPane"),
            loader.loadClass("edu.vatech.cs.ui.JRadarPane"));
 7.       classCheck.put(loader.loadClass("com.sun.java.swing.text.PlainDocument"),
            loader.loadClass("edu.vatech.cs.rJamm.ui.text.rJammSharedDocument"));
 8.     } catch (ClassNotFoundException ex) {
 9.       System.out.println("Error setting up class-check table: "+ex);
10.       ex.printStackTrace();
11.     }
12.   }
13.   protected Object replaceObject(Object obj)
14.     throws IOException
15.     {
16.       // ...
17.       Enumeration e = classCheck.keys();
18.       while (e.hasMoreElements()) {
19.         Class testClass = (Class)e.nextElement();
20.         Class replaceClass = (Class)classCheck.get(testClass);
21.         if ( (testClass.isInstance(obj)) &&
                  !(replaceClass.isInstance(obj)) ) {
22.           try {
23.             // instantiate a replacement
24.             Class [] parmTypes = new Class[1];
25.             parmTypes[0] = testClass;
26.             Constructor replaceCons = replaceClass.getConstructor(parmTypes);
27.             Object [] parms = new Object[1];
28.             parms[0] = obj;
29.             Object replaceObj = replaceCons.newInstance(parms);
30.             obj = replaceObj;
31.           } catch (Exception ex) {
32.             // For all exceptions, just return the original object
33.             System.err.println("Error replacing object: "+ex);
34.             System.err.println(
                  "Unable to replace instance of class: "+ testClass+"\n"+
                  "             with instance of class: "+replaceClass);
35.             ex.printStackTrace();
36.           }
37.         }
38.       }
39.       // ...
40.       return obj;
41.   }
42. }
```

Figure 5.6: Flexible JAMM's object replacement code.

5.3.4 User Input Event Interception and Event Introduction

Java's user input event queue has two important limitations that affect the development of a collaboration-transparency system (see also (Begole *et al.*, 1997a)): (1) input events may be received by the application before they can be intercepted at the Java layer and (2) it is not possible to introduce remotely generated events into the local event queue. In the first JAMM prototype (Begole *et al.*, 1997b), we circumvented these problems by adding support for event interception and introduction into the native-platform code of the AWT.

We call the interception/introduction point implemented in JAMM and illustrated in Figure 5.7 **pre-peer** because events are intercepted before they reach the native implementation of an interface object, called a **peer**. In contrast, a **post-peer** interception point would exist anywhere along the event path in the box labeled "Java Objects." Post-peer interception is the only option if an AWT-based collaboration-transparency system is to be implemented *entirely* in Java. However, post-peer interception does not allow all replicas of the shared application to display graphical feedback to user inputs.

User interface elements are normally expected to deliver feedback upon user input (such as drawing a depressed screen button when the button is activated), signaling that the input has been received. It is equally important to provide this graphical feedback to remote collaborators. Without such notice, collaborators may be surprised by sudden changes in the application. As a minimum, users should see interface element reactions to each collaborator's inputs to the system. For example, if collaborator A clicks on and drags the thumb button of a scroll bar, the other collaborators should see A's telepointer move to the scroll bar, and then see the scroll bar itself moving on their displays.

Pre-peer interception provides access to all low-level user input events and allows us to deliver events to peers so that the GUI component can perform its graphical response to the inputs. The downside is that it requires modifications to the native implementation of the AWT for each platform to which the AWT is ported.

Swing

Flexible JAMM avoids the problems associated with pure AWT-based event interception and introduction by requiring that shareable applications be implemented using the Java Foundation Classes' (JFC) user interface library, called Swing (Java Foundation Classes, 1998). Swing is implemented entirely in Java and has no platform-dependent portion, such as the AWT's peers. Therefore, event

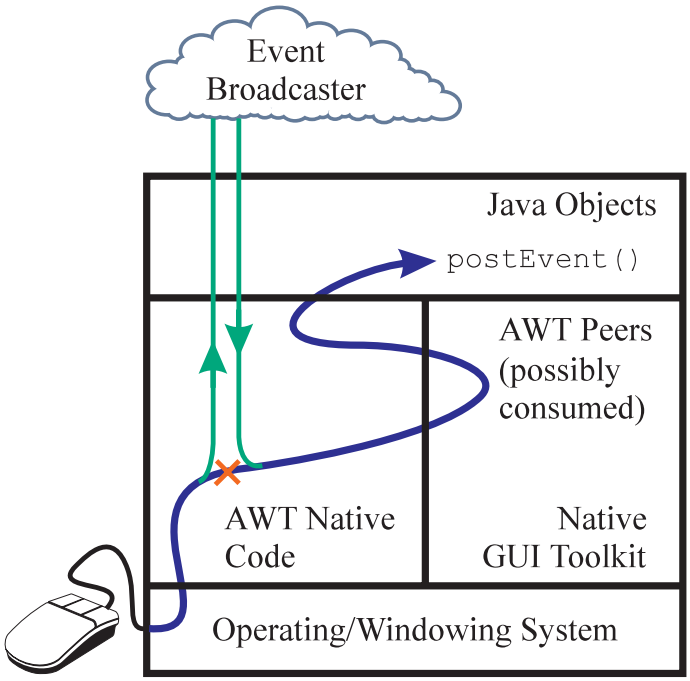


Figure 5.7: The curved line extending from the mouse illustrates the path of a user input event in the Java AWT. After the event is generated at the hardware level, the operating/windowing system (e.g., MS Windows, or X) puts it in the event queue of the Java AWT native code. The AWT dispatches the event to the native GUI toolkit (e.g., MS Windows, or Motif), which contains the native interface components (AWT peers). A peer may consume the event or return it to the AWT native code, which again examines the event and this time delivers it to the Java applet-level AWT objects with `Component.postEvent()`. The lines extending to and from the cloud labeled "Event Broadcaster" show the path of an intercepted event. Locally generated events must be intercepted at that point so that the native-platform peers do not consume them before they can be distributed. Similarly, a remotely generated event must be introduced at that point so that the native-platform peer can receive and potentially act on the event.

interception can be implemented entirely in Java at a post-peer interception point. Thus, Flexible JAMM preserves Java's advantage of platform independence that JAMM's pre-peer interception had forfeited.

There are several additional reasons why I chose Swing for the implementation of Flexible JAMM. First, interface components in a Swing window are drawn on various *layers*. This allows a fairly straightforward implementation of telepointers, which are drawn on a layer "above" the application's components. In contrast, telepointers in the AWT-based JAMM prototype were not pure Java code, but required native platform implementations. Second, the Swing text package facilitates implementing concurrent editing by having synchronized insert and remove methods and notification of changes. Additionally, Swing's text component has separate interface and data classes, whereas in the AWT these are combined. Thus, Flexible JAMM is able to replace just the data object of a text component, allowing different interface implementations for concurrent editing. For example `JTextArea` and `JTextField` provide different interfaces using instances of the same data model, `PlainDocument`. Finally, because Swing is a more complete interface library than the AWT and will be a part of the core Java library in Version 1.2 (it is currently available as a free, but separate, package), I expect Swing to supplant the AWT as the default user interface library for Java applications. This final point, of course, remains to be seen.

Shareable Applications

Reliance on the Swing interface library and JOS comes at some cost. Collaboration-transparency systems generally allow the sharing of any application written for a particular operating system/windowing toolkit (e.g., X or Macintosh). Due to the constraints outlined previously, Flexible JAMM's target platform for *transparent* sharing is not the entire set of Java applications, but rather the subset of serializable, Swing-based Java applications. For the time being, the set of legacy single-user applications that meet these constraints is fairly small. However, we can expect this set to grow as Swing becomes part of the core Java library in Version 1.2. It is important to note, however, that Flexible JAMM is not a commercial system, but demonstrates the approach to flexible collaboration transparency outlined in Section 5.2. These techniques are applicable to future collaboration-transparency systems.

5.3.5 Single- to Multi-user Replacement Classes

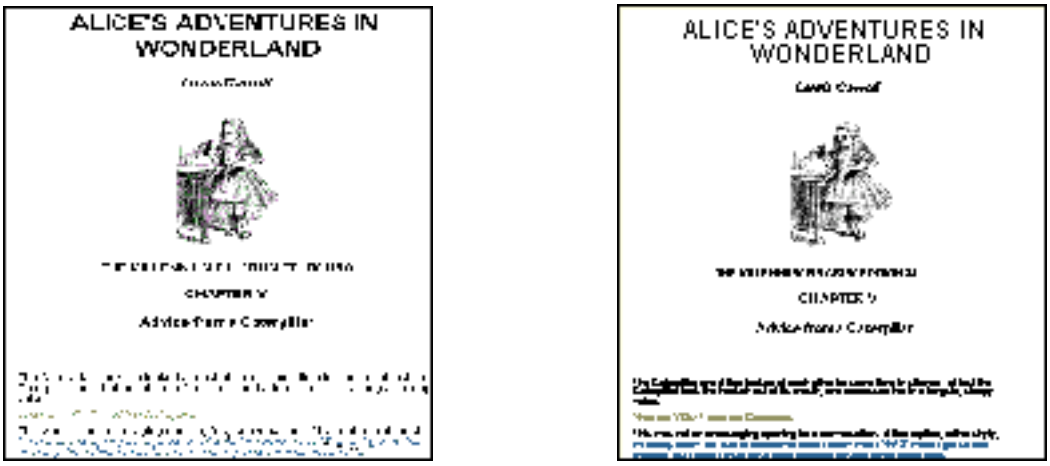
This section describes the two classes I have implemented to replace instances of Swing classes. Although other classes were considered, these two appeared to offer the greatest benefit in support of collaboration.

Multi-user Radar Pane

Flexible JAMM replaces each instance of `com.sun.java.swing.JScrollPane` by a multi-user radar pane (see Figures 5.2 and 5.4), called `edu.vatech.cs.ui.JRadarPane`. The radar pane provides several benefits to collaboration: location-relaxed WYSIWIS, detailed group awareness, and immediate local response to scroll position changes. The radar pane class can be used in a single-user application as well, where it's overall view of an embedded component facilitates navigation.

The miniature radar view image is obtained by having the embedded component draw itself to a special implementation of `java.awt.Graphics,` the class that defines drawing routines in the AWT. The `ScaledGraphics` class scales the coordinates, images, and font sizes relative to the size of the radar view. As opposed to drawing at full size then scaling the resulting image, on-the-fly scaling has two advantages. First, on-the-fly scaling uses less memory because it draws directly to the final miniature bitmap image, whereas the other approach draws first to an intermediate, full-size bitmap image, which is then scaled to the final miniature. Second, on-the-fly scaling preserves the order of drawing operations. That is, foreground items are drawn over top of background elements. In contrast, full-size scaling cannot distinguish foreground from background elements and therefore results in less-prominent foreground elements than on-the-fly scaling. Thus, images produced by on-the-fly scaling are perceived as more clear than full-size scaling. For example, compare the scaled documents shown in Figure 5.8.

On-the-fly scaling is another reason why I chose to use Swing over the AWT. Swing interface components draw themselves through an implementation of `Graphics` and can thus be scaled on the fly via `ScaledGraphics.` In contrast, only a full size image of an AWT component can be obtained because they are drawn by the native-platform user interface library. Although that image can then be scaled, the result is not as clear as that created by scaling the individual draw operations generated by a Swing component.



Scaled after drawing Scaled while drawing

Figure 5.8: Normal image scaling versus Flexible JAMM's on-the-fly scaling while drawing. Both images are miniatures of the document seen in Figure 5.2 on page 79. The image on the left was created by obtaining a bitmap image of the document and scaling it. Java's bitmap scaling approach cannot distinguish foreground from background pixels, and the resulting image loses foreground pixels. For example, only scattered dots remain of the smallest text in the left image. The image on the right was drawn by scaling each graphics operation as it was executed. Thus, each line of text was rendered by scaling the font as the text was drawn. Because this approach maintains the order in which items are drawn, items in the foreground are wholly drawn. For example, distinct words are recognizable in the smallest text of right image (although too small to read), whereas distinct words cannot be discerned in the same text of the left image.

Multi-user Text Editing

Flexible JAMM also substitutes each instance of `com.sun.java.swing.text.PlainDocument` (the default data model of a Swing text component) with a multi-user version, called `edu.vatech.cs.-ui.SharedDocument`. This provides significant groupware benefits: independent concurrent work, and immediate local response to edits.

To provide concurrent editing, Flexible JAMM's text data model uses atomic operations (insert and delete) to update replicas. Flexible JAMM uses an operational transformation algorithm similar to the REDUCE (REal-time Distributed Unconstrained Cooperative Editing) approach described by Sun *et al.* (1998), discussed in Section 5.3.7.

5.3.6 Employing Semantic Knowledge of User Input Events

Flexible JAMM intercepts and distributes low-level input events before they are delivered to the application. In addition, Flexible JAMM intercepts higher-level actions, such as scrollbar position changes and text edits. Thus, Flexible JAMM is aware of the semantic result of certain low-level input events. This section describes how Flexible JAMM uses semantic knowledge of input events to provide key groupware benefits: relaxed WYSIWIS, detailed awareness information, flexible floor control, concurrent work, and fast local application response.

Relaxed WYSIWIS

Replication and event semantics allow a flexible collaboration-transparency system to relax WYSIWIS by providing participants with independent views of the shared application. Instead of automatically distributing all inputs, in some cases Flexible JAMM does not distribute a given input event, but only applies it locally.

For example, the result of an input event within a scrollbar is to move the position of the scrollbar and in turn a viewport. In this case, a flexible collaboration-transparency system need not distribute the mouse event to all replicas, but only apply it locally because each replica is allowed to have an independent scroll position. Hence, the flexible collaboration-transparency system provides location-relaxed WYSIWIS.

Flexible JAMM allows the collaborators to turn off view relaxation, falling back to the traditional replicated approach of distributing *all* input events. Turning off this feature prevents replica inconsistency, but loses the advantage of relaxed WYSIWIS.

Awareness Information

To allow participants to coordinate their work, it is necessary to provide information about each other's location and activities in the workspace (Section 2.2.4).

Continuing the previous example, after the scroll position is changed locally, the system should send a message to remote participants indicating the new local viewport position. In response to a viewport change notification, a remote replica will update its location-awareness interface elements, such as the viewport indicators in the radar view (Figure 5.2). Therefore, in addition to telepointers, Flexible JAMM provides group awareness information by indicating each participant's viewport position in the radar view.

Explicit and Implicit Floor Control

To avoid potential conflicts between non-atomic events such as mouse drags (see Section 4.1.1), Flexible JAMM uses a floor control mechanism for events that occur in some, but not all, interface components. Where floor control is used, Flexible JAMM provides mechanisms for both **explicit** floor requests, where participants explicitly grab and release control of the application, and **implicit** floor requests, where the system automatically makes floor requests as collaborators use the application. Implicit control provides seamless floor passing as work flows among the participants.

Figure 5.9 shows the user interface to Flexible JAMM's floor control mechanism. The control holder is indicated by showing his or her name in the control panel and by having an arrow-shaped telepointer versus hand-shaped telepointers for non-controlling participants (Figure 5.2). Control is implicitly granted to the first participant to press a mouse button, who may be beginning a potentially non-atomic series of inputs (i.e., a mouse drag). That participant holds control until he or she releases the mouse button and for a short time (5 seconds) after. The delayed release allows the same user to maintain control if they continue to generate inputs, which is often the case when selecting an item from a series of cascading menus. Control may also be explicitly requested by pressing the "Take Control" button. Control requests are granted unless the current control holder checks the "Keep Control" check-box.

Figure 5.9: Flexible JAMM's input control panel indicating that control is currently held by the collaborator named Janice. Input control may be explicitly requested by pressing the "Take Control" button, or implicitly when a participant presses a mouse button in the shared application.

Concurrent Work

Although implicit control passing is easier to use than explicit control requesting/granting, it still allows only one participant at a time to work in the shared application. However, this application-wide locking is not required when an event is generated within a collaboration-aware interface object, such as the scrollbar of the Flexible JAMM radar pane and the multi-user text editing component. In those cases, Flexible JAMM does not impose floor control; all participants may simultaneously change their independent scroll positions and edit the text, as described previously.

Fast Local Response

In contrast to previous replicated collaboration-transparency systems, like JAMM and Dialogo (Lauwers, 1990), Flexible JAMM does not automatically forward all user input events; some are only applied locally. As described previously, key and mouse events that occur within a text area are not broadcast, but are only applied locally. Similarly, inputs that occur within a scrollbar of a radar pane are also only applied locally. In both cases, any resulting state change is sent some time after the input event. Thus, the local replica of the shared application responds immediately to the input and remote replicas are updated asynchronously.

The states of the replicas are not consistent until after all updates have arrived at each replica. We can be sure that the final states are consistent because Flexible JAMM only delays events that are not time dependent (discussed in Section 3.2.5). In the cases of scrollbar position and text entry inputs, the replicas will be consistent because the result does not depend on when the events were generated.

5.3.7 Distributed Concurrent Editing

To enable concurrent editing without locking, Flexible JAMM uses an operational transformation algorithm, several of which were reviewed by Sun and Ellis (1998). This section describes operational transformation algorithms in general and Flexible JAMM's approach in particular.

Operational Transformation

Operational transformation algorithms require no locking, yet preserve the users' intentions where conflicts occur (Ellis & Gibbs, 1989; Sun & Ellis, 1998). Each replica updates its copy of the document by transforming incoming remotely generated atomic operations to be consistent with its own set of local operations.

Operational transformation is distinct from serialized atomic transaction approaches used by distributed databases to allow access by multiple simultaneous clients (see Coulouris *et al.* (1994) for a general discussion of transaction approaches to shared data). Operational transformation applies all transactions to the shared data, transforming them to *preserve* the users' intentions. However, potentially conflicting transactions generated on a distributed database may be aborted to maintain the database's integrity, which is counter to the user's intention. Furthermore, distributed database, operating, file, and time-sharing systems traditionally provide the illusion of individual access to shared resources. That is, concurrent users are hidden from each other, which is counter to the goal of collaborative computing.

To apply operational transformation to text editing, changes to the document are made via two atomic operations: (1) `insert(int position, String newText)` inserts `newText` into the existing text at the offset specified by non-negative `position` and (2) `remove(int position, int length)` removes the number of characters specified by `length` beginning at `position`. An instance of either operation is called an edit. For any edit, there is an inverse. For example, the inverse of `insert(4, "insertion")` is `remove(4, 9)`. The inverse of a remove operation depends on the content of the text when the remove was generated. For example, if the document contained the text `"01234556789"`, the inverse of `remove(3, 2)` would be `insert(3, "34")`.

When a user generates an edit, the system creates a message containing the edit information, applies the edit locally, adds the edit to a queue of *unacknowledged* edits, which are used to transform an incoming edit that was generated concurrent to a local edit. The system then distributes the edit

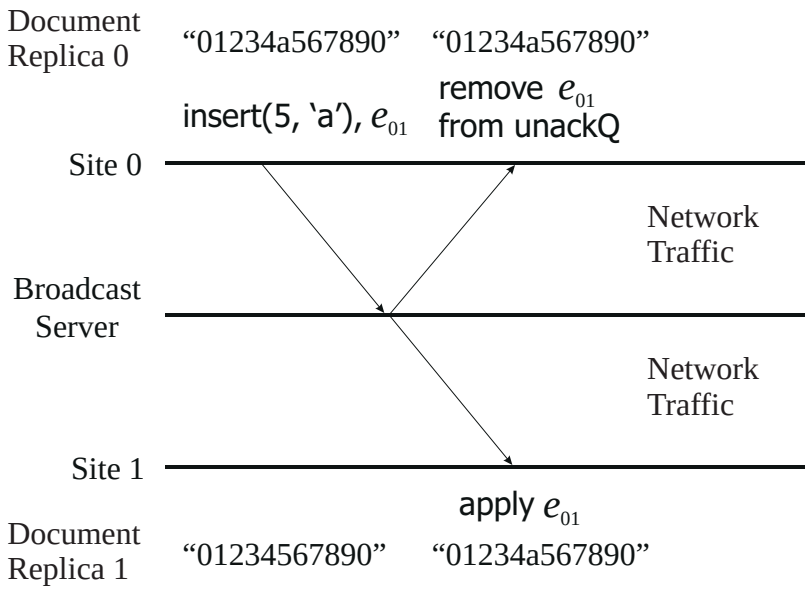


Figure 5.10: Simple case of concurrent editing: no conflicting edits. the locally generated edit does not conflict with any remotely generated edits. Arrows indicate message transmission; horizontal lines indicate the message send and arrival times at each site. In this example, the user at site 0 inserts the character ‘a’ at position 5. Edit operations are labeled e_{ij}, where i is the site number and j is the edit number. For example, the first edit generated at site 0 is labeled e_{01}. e_{01} is added to a queue of *unacknowledged* edits and sent to the broadcast server. When e_{01} arrives back at site 0, it is considered *acknowledged* and removed from the unacknowledged queue. When e_{01} arrives at site 1, it is applied to document replica 1, resulting in the same text at both sites.

message to all replicas of the shared document, including the local replica. Messages from all sites arrive in the same order at all sites (Section 5.3.8).

When each replica receives an edit message, it performs the following steps. If the incoming edit matches the first edit in the unacknowledged queue, then this edit was generated locally and has already been applied to the document. In that case, the edit is removed from the unacknowledged queue and processing is complete. Figure 5.10 illustrates this case. Otherwise, the edit was generated remotely. In that case, the system performs the following steps. If the unacknowledged queue is empty, then the incoming edit does not conflict with any local edit and can safely be applied to the document. If the unacknowledged queue was not empty, then the edits were generated concurrently and must be specially handled, as described next.

The more difficult case to handle is when edits are generated concurrently at different sites. In that case, each replica updates its copy of the document by transforming an incoming remote edit to be consistent with its own set of local operations. To do this, each edit message must contain information about what state the document was in when the edit was generated. One way to provide this information is by tagging each edit message with a **vector clock** timestamp, which represents a notion of global time among the replicas (Coulouris *et al.*, 1994; Singhal *et al.*, 1994). Briefly, vector clocks are implemented in the following way. The size of the vector is n, where n is the number of participating sites. Site indices in the vector are numbered from 0 to $n-1$. Each time a site sends or receives a message, it increments the value of its clock value in the vector. All outgoing messages are tagged with the current values of the vector (after incrementing the local clock value). When a site receives a message, it again increments the local clock value in the vector. The site then updates the values of each remote clock in its vector to match the values in the message's vector timestamp.

This system of vector clocks allows a site to determine what state the document was in when an operation was generated. Figure 5.11 illustrates how Flexible JAMM uses vector clocks to adjust incoming edits to match the state of the local replica. The adjustment preserves the intention of the edit in the context of the document state in which the edit was generated. Sun refers to this characteristic of operational transformation algorithms as **intention preservation**. For example, in Figure 5.11, the user at site 1 deleted the 'r' character. Therefore, at site 0, the operation is adjusted so that the 'r' character is deleted there also. Note that intention preservation is a stronger requirement than *consistency,* which requires that all replicas contain the same data. Consistency may be satisfied by simply erasing the entire document contents at all sites, but that would not preserve the intentions of the users.

In the example seen in Figure 5.11, the edits did not conflict. However, it is possible for users to concurrently generate conflicting edits. For example, one user may be inserting a character into a word while another user is deleting the same word. In that case, the system is unable to adjust edits to conform to both users intentions. However, consistency can still be maintained, by applying one of the edits before the other on all replicas. The order can be arbitrarily determined, for example by assigning precedence according to the site indices, as long as the order is the same on all sites. Figure 5.12 illustrates this approach. When user edits conflict in this way, the users must coordinate their efforts.

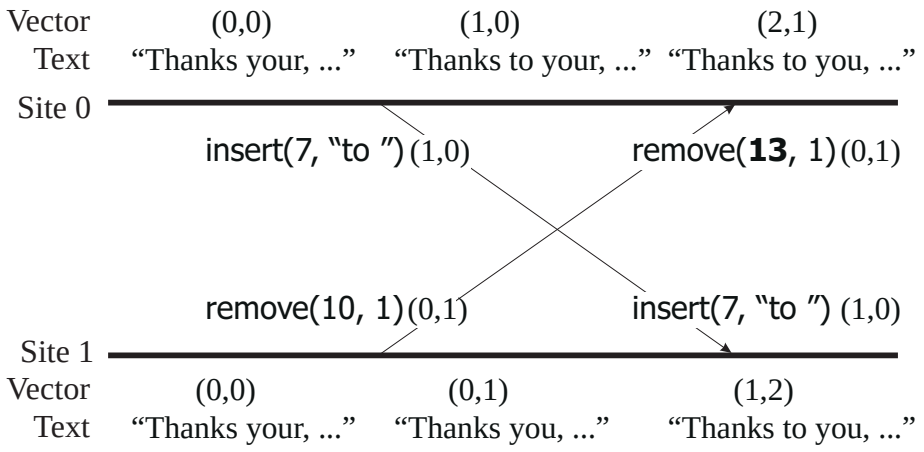


Figure 5.11: Users at sites 0 and 1 are collaboratively changing the replicated text from "Thanks your, ..." to "Thanks to you, ..." Each site has concurrently generated and distributed an operation (the broadcast server seen in Figure 5.10 has been removed from this diagram for clarity): site 0 inserts "to " at position 7, and site 1 deletes the 'r' character at position 10. When site 1's remove message arrives at site 0 with the timestamp of (0,1), site 0 can determine that the operation was generated before site 1 had received site 0's insert (otherwise, the timestamp would have been $(1, x)$). Therefore, site 0 adjusts the position of the remove from 10 to 13 (in bold) and applies the operation, correctly removing the 'r' character. When site 0's insert arrives at site 1, site 1 can determine that the operation was generated before site 0 had received site 1's remove. However, in this case, there is no need to adjust the position of the insert because the previous remove does not affect the position of the incoming insert.

Flexible JAMM's Operation Transformation Approach

Flexible JAMM's approach is similar to that of the REDUCE system (Sun *et al.*, 1998). This approach differs from others in that remotely generated operations are not transformed directly. Instead, the local replica maintains a queue of previously applied operations (both locally and remotely generated) and their inverses. The inverse operations are applied to the local replica to bring the document back to the state in which the remote operation was generated. Then the remotely generated operation is applied to the document. Next, the local operations are modified to preserve their intention and then re-applied. Figure 5.13 illustrates how Flexible JAMM actually handles the general operational-transformation case seen in Figure 5.11.

An operation is only kept in its queue until it is clear that all sites have received and applied that operation. This is determined as follows. Each operation, o, was generated by a particular site,

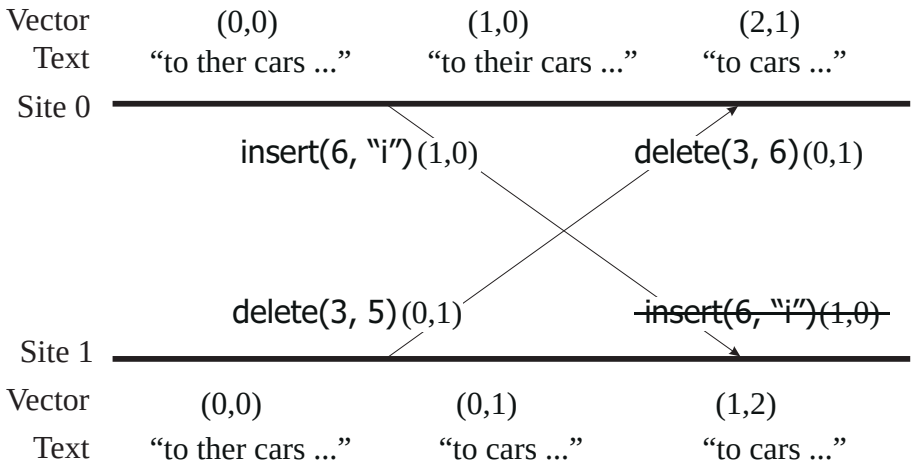


Figure 5.12: Handling conflicting concurrent edits. Site 0 operations are given precedence over site 1. Therefore, the insert is applied before the delete. The result is that on site 0, the remotely generated delete is adjusted to include the inserted character, while on site 1 the insert can be safely dropped.

s. When a new operation arrives, if its clock value for s is equal to or greater than s's clock value for an operation in the queue, then the queued operation has been received and applied at the other site. Therefore, the new operation was made in the context of the queued operation and the system will only undo previous operations that were added to the queue *after* the queued operation. Thus, when all sites have received an operation, it can be removed from the queue. In case a replica is not generating operations that carry its vector clock information, each Flexible JAMM site sends periodic messages with that site's current vector clock. Thus, the operation queues do not grow continuously.

Undo-Do-Redo versus Direct Transformation

The approaches used by Flexible JAMM and REDUCE (Sun *et al.*, 1998) use more processing time than other operational transformation approaches, such as Ellis and Gibb's (1989) distributed OPerational Transformation (dOPT) algorithm. When a remotely generated operation arrives, Flexible JAMM uses the following processing sequence: previously applied operations are undone, the new operation is applied, the previous operations are adjusted, and the previous operations are re-applied. In contrast, other approaches transform the incoming operation and apply it directly.

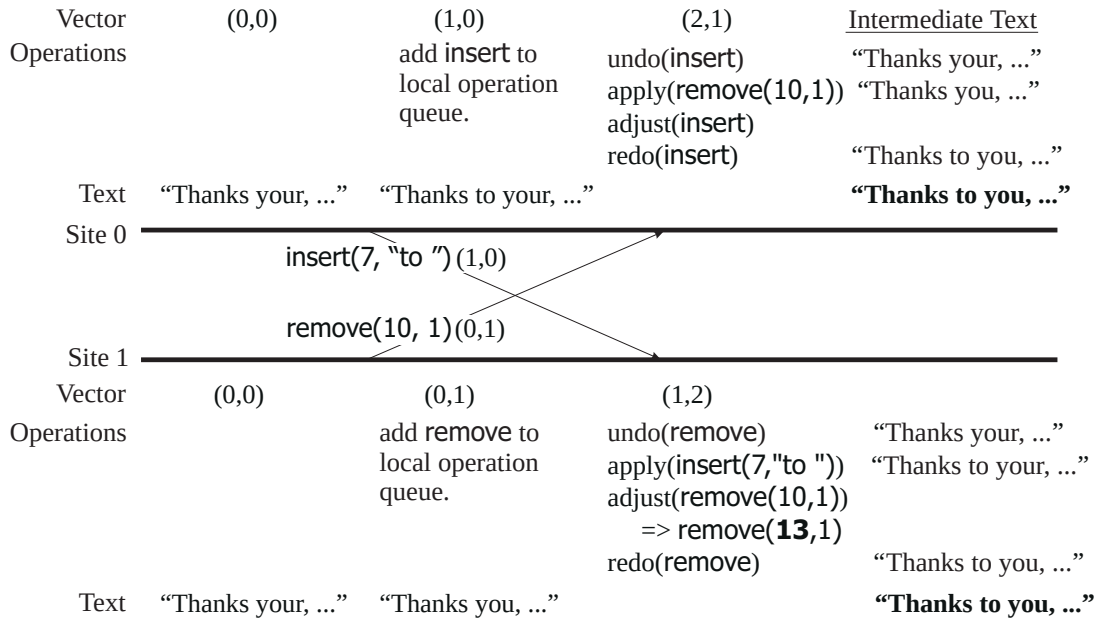


Figure 5.13: Flexible JAMM's undo-do-redo transformation approach. When the remotely generated insert arrives at site 1, the document state is returned to that in which the insert was generated at site 0 as determined by the vector clock stamp, (1,0). This is accomplished by undoing the operations in the local operation queue that were applied after that, according to each operation's vector clock stamp. The insert is applied to the document, then the local operations are adjusted to preserve the user's intention. Thus, remove(10,1) becomes remove(13,1). Finally, the local operations are re-applied to the application. A similar sequence occurs at site 0 when the remove operation arrives, resulting in the same text at each site (on the right, in **bold** typeface).

However, Flexible JAMM's processing time is less than the typical inter-arrival time of edits, which is bound by the user's typing speed. A user with an average typing speed of 35 five-character words per minute will generate an edit operation at an average rate of approximately one every 343 milliseconds. Thus, in a session of three simultaneous participants, the inter-arrival time of remotely generated operations will be approximately 343ms/2 = 171.5ms. As the number of participants simultaneously generating operations grows, the inter-arrival time decreases, which requires the local replica to process incoming operations more frequently. Although it is slower than other approaches, Flexible JAMM's undo-do-adjust-redo process is not perceivable in the typical-usage scenario of only a few simultaneously editing participants. In laboratory trials, up to nine users of the text editor seen in Figure 5.4 on Intel Pentium 133 MHz-based computers running Java 1.1.5

have simultaneously edited a shared document without perceivable decrease in the application's response to locally generated edits.

Flexible JAMM Concurrent Editor Implementation

Flexible JAMM's implementation of the previously described algorithm can be used in collaboration-aware applications, and is available from the Flexible JAMM web site (Begole, 1997). The implementation is contained in a package named `edu.vatech.cs.ui.collab`. Table 5.1 describes each class in that package.

5.3.8 Flexible JAMM Communications Layer

Flexible JAMM's primary communications requirement is that messages generated from all sites be received by all sites in the same order, which is referred to as **total ordering**. In recent years, many multicast protocols have emerged (Obraczka, 1998), but few provide total ordering of message delivery, and there are currently no Java implementations of total-ordered multicast protocols. Therefore, Flexible JAMM uses a straight-forward communications layer where a replica sends a message to a central site that in turn unicasts the message to each replica, including the originator.

Communications protocols may be **synchronous**, where the process sending a message stops processing, or "blocks," until a response is received from the message target, or **asynchronous**, where the sender continues processing immediately after sending a message. Remote service packages, sometimes called "middle-ware," such as Java Remote Method Invocation (RMI) and remote procedure call (RPC) use a synchronous approach. Although these packages are convenient to use, the processing delay imposed by the synchronous communications can severely impact the performance of a real-time collaborative system. In a semi-replicated real time groupware system that used a synchronous communications model, Graham *et al.* (1996) found that clients can spend up to 95% of the time blocked, waiting for responses. In many cases, such as sending update information among replicated shared data, no response is necessary. In those cases, an asynchronous protocol is more efficient than a synchronous one. It is possible to impose synchronicity at the application layer using an asynchronous protocol, and asynchronicity can be implemented using a synchronous protocol. However, in both cases, it is complex to implement one with the other.

In summary, an asynchronous protocol allows the sender to resume processing immediately after sending a message, and relieves the system of half the network messages when no acknowledgment

Table 5.1: Flexible JAMM's concurrent editor classes are contained in a package named `edu.vatech.cs.ui.collab`. The list is in order of relevance to a developer. Interfaces are listed in *italic* typeface.

Class	Description
`SharedDocument`	`SharedDocument` extends `com.sun.java.swing.text.PlainDocument`. To start a concurrent editing session, the application should create a `SharedDocument` and set it as the document of a Swing editor component, `com.sun.java.swing.text.JTextComponent`. The `SharedDocument` must also be assigned a global session identifier and a site number.
SharedDocument-Listener	One or more *SharedDocumentListener* objects can be associated with a `SharedDocument`. Whenever the local user generates an edit, all registered *SharedDocumentListener* objects are notified and given an EditCommand. The EditCommand should be sent to *all* replicas (including the originating one) and arrive in the same order at each.
`EditCommand`	`EditCommand` encapsulates the information required to perform either an insert or a remove operation and its inverse, including the document instance to which the edit applies and the vector clock stamp when the edit occurred. `EditCommand` object vector clocks can be compared to determine the order in which they should be applied relative to each other. `EditCommand` is serializable, allowing easy message creation. When a remotely generated `EditCommand` arrives, it should be applied to the local copy of the `SharedDocument` via the method `SharedDocument.applyEdit(EditCommand)`.
EditPrefilter	One or more *EditPrefilter* objects can be added to a `SharedDocument`. Each *EditPrefilter* is notified prior to the application of an edit and can block the application of the edit. For example, Flexible JAMM uses this to prevent multiple edits due to cut and paste operations.
`Adjuster`	`Adjuster` modifies `EditCommand` objects to be consistent with the edits previously applied to the local replica of the document. There are two places in the distribution architecture at which `Adjuster` objects can exist: either (1) one `Adjuster` for each replica (this is the default) or (2) one `Adjuster` at the central message dispatcher.
`GlobalClock`	A vector clock implementation. `GlobalClock` contains and manages the local vector clock.
`EventIdentifier`	An `EventIdentifier` object is associated with each `EditCommand` object. `EventIdentifier` contains the vector clock stamp when the `EditCommand` occurred.
`EditCommand-Vector`	Allows fast processing of queues (vectors) of `EditCommand` objects.

Table 5.2: Flexible JAMM's message sending classes are contained in a package named `edu.vatech.cs.collab.message`. The list is in order of relevance to a developer. Interfaces are listed in *italic* typeface.

Class	Description
MessageQueue	The *MessageQueue* interface defines a method to add a byte array message to a queue.
UnicastSender	UnicastSender is an implementation of *MessageQueue*. UnicastSender writes newly added messages to an OutputStream, which is typically a network socket or inter-process pipe.
MulticastSender	MulticastSender is an implementation of *MessageQueue*. MulticastSender writes newly added messages to a set of *MessageQueue* objects (i.e., UnicastSender objects).
MessageOutputStream	To send a message, an application constructs a MessageOutputStream with a *MessageQueue* instance. Primitive data types can be written directly to a MessageOutputStream. The first data written must be an integer which identifies the message type.
BufferQueue	BufferQueue is an implementation of *MessageQueue*. BufferQueue does not send the messages added to it, and is used to temporarily store messages for later transmission while a latecomer is brought up to date.
ByteArrayList	A linked list of byte arrays used by *MessageQueue* implementations to queue messages.

is required. In Flexible JAMM's case, most messages contain user input events or other information that require no response. Therefore, an asynchronous approach is preferred.

Flexible JAMM's communications layer consists of message-sending and message-receiving classes. The message-sending classes, listed in Table 5.2, are contained in a package named `edu.vatech.cs.collab.message`. Messages are sent by writing data to a `MessageOutputStream`, to which any Java primitive data type can be written directly, and entire objects can be written by first converting the object to a byte array using JOS. After the data are written, the application must call `flush()`, which adds a message to an outgoing queue. The application can resume processing as soon as the message is added to the outgoing queue. When there are messages in the outgoing queue, a separate thread writes each message to the outgoing stream, typically a network socket or an inter-process pipe. Flexible JAMM has two sender implementations, `UnicastSender` and `MulticastSender`. Each application replica uses a `UnicastSender` to send messages to the central message server, which uses a `MulticastSender` to send messages to all replicas.

Flexible JAMM's message-receiving classes, listed in Table 5.3, are contained in a package named `edu.vatech.cs.collab.acceptor`. On the receiving end of a network or inter-process connection,

Table 5.3: Flexible JAMM's message receiving classes are contained in a package named `edu.vatech.cs.collab.acceptor`. Interfaces are listed in *italic* typeface.

Class	Description
`Acceptor`	An `Acceptor` receives messages from a network socket or inter-process connection. `Acceptor` dispatches messages to multiple *`Handler`* objects.
`Handler`	*`Handler`* objects receive messages. A *`Handler`* implementation parses the message and invokes an action.

one instance of an `Acceptor` receives each message, determines the message type, and dispatches it to each `Handler` object that has registered to receive messages of that type. Multiple `Handler` objects can receive the same message. For example, in the Flexible JAMM server, two `Handler` objects receive participant-related information messages: one echoes the information to all replicas, while the other logs the information so that it can be delivered to latecomers.

5.4 Summary

This chapter presents a general approach to flexible collaboration transparency that supports key groupware principles that have previously been seen only in collaboration-aware applications: concurrent work, relaxed WYSIWIS, detailed group awareness, and efficient network usage.

The chapter first enumerates the desired features of a flexible collaboration-transparency system. The technical specifications are: no change to the application source code or underlying platform API (i.e., transparency), any performance cost incurred while sharing an application is not incurred when *not* sharing an application, and minimal network bandwidth usage. In addition, to provide flexible support for collaboration, the system should satisfy the following usability specifications: a user is not required to anticipate the need to share an application, new collaborators can join an ongoing session, collaborators can work simultaneously, "turn taking" is automated in situations where simultaneous work is not possible, location-relaxed WYSIWIS, and detailed group workspace awareness information.

Next, a new general approach to collaboration transparency that satisfies those requirements is described. The new approach uses a replicated architecture where selected single-user objects, such as a scrollable panel, are replaced at run time with multi-user versions, such as a multi-user radar panel. The multi-user replacement must implement the same object interface as the original

so that the replacement appears to the application to be an instance of the original. Therefore, the replacement is transparent to the single-user application and its developers.

The first characteristic of this approach is replication, which provides relaxed WYSIWIS and concurrent work, in addition to network efficiency. Each replica's interface state can differ slightly and operations can be concurrently applied to multiple replicas. The second characteristic of the new approach, replacing single-user objects with collaboration-aware equivalents, provides the system with knowledge of input event semantics. With this, the system can optimize its network usage by minimizing the size and number of messages distributed to replicas. In some cases the system can safely modify, delay, or drop an input event. In contrast, conventional collaboration-transparency systems must distribute all input events.

To implement the new approach, the underlying platform must provide the following capabilities: process migration, run-time object replacement, dynamic binding, and access to user input events. Process migration is necessary to provide a copy of the shared application in its current state to newcomers. Two forms of process migration are possible: (1) event logging and replay and (2) image copy. Event logging is possible on most platforms, but can require approximately an order of magnitude greater time than image copy. Image copy requires support from the platform to provide the process' address space, current execution state, and connections to system resources. Few current application platforms support image copy. Object replacement must occur after the application has started to support unanticipated sharing. Run-time object replacement is supported on some object-oriented application platforms, such as Java. Dynamic binding is necessary for flexible collaboration transparency so that one object can take the place of another without modifying the application. Because the application source code is not modified, the substitution is *transparent* to the shared application and its developers. Lastly, a collaboration-transparency system must be able to intercept local user input events and introduce remotely generated events. Event interception is needed for both centralized and replicated implementations.

Section 5.3 describes a Java-based prototype implementation of the new approach, called Flexible JAMM. The section describes the Java platform's support of the previously described requirements. Process migration and run-time object replacement are provided by the Java Object Serialization (JOS) library. Dynamic binding is inherent in Java. There are two difficulties using the standard Java user interface library (the AWT) to intercept and introduce user events: (1) input events may be received by the application before they can be intercepted at the Java layer and (2) it is not possible

to introduce remotely generated events into the local event queue. In an earlier prototype, called JAMM, we extended the AWT to avoid these problems by implementing pre-peer event interception and introduction. However, this fix required modifications to platform-dependent portions of the AWT, thereby forfeiting platform independence.

Flexible JAMM is able to preserve platform independence by constraining the set of shareable applications to those that are implemented with the Swing user interface library. Swing is implemented entirely in Java. Therefore, it is possible to intercept and introduce input events at the (post-peer) platform-independent Java layer. Swing has additional features that make it preferable to the AWT for flexible collaboration transparency: *layers* allow straightforward telepointer implementation, text components have separate interface and data model objects, the text data model facilitates concurrent editing by providing synchronized insert and remove operations and notification of text changes, and Swing components can be scaled as the component is drawn.

Section 5.3.5 describes the two primary multi-user classes that replace single-user classes when an application is shared: multi-user radar pane, and multi-user text document. Flexible JAMM's radar view scales components while they are drawn as opposed to drawing at full size then scaling the resulting bitmap image. On-the-fly scaling has two advantages: (1) smaller memory requirement and (2) foreground elements remain prominent.

Section 5.3.6 describes how Flexible JAMM uses knowledge of event semantics. Flexible JAMM provides location-relaxed WYSIWIS, concurrent editing, and fast local response by not distributing events that occur in collaboration-aware components. In contrast, previous collaboration-transparency systems distributed all user events because they were unable to determine which could be safely modified, delayed, or dropped. Event semantics also allow Flexible JAMM to provide awareness information and to automate turn taking for non-atomic events (i.e., mouse drags) that occur within components that are not collaboration-aware.

Section 5.3.7 describes Flexible JAMM's operational transformation algorithm for distributed concurrent editing. Operational transformation algorithms require no locking, yet largely preserve the users' intentions where conflicts occur. Each replica updates its copy of the document by transforming incoming remote operations to be consistent with its own set of local operations. Examples are given of handling cases of conflicting and nonconflicting concurrent edits. Flexible JAMM's operational transformation implementation is less efficient than other approaches, but is not perceivably slower in typical-usage scenarios of only a few simultaneously editing collaborators.

Section 5.3.8 describes Flexible JAMM's communications package. Messages are sent from each participant to a central host (the host on which the application was initially shared) and multicast to all participants. The central distribution imposes a total ordering on messages so that each application replica receives the same messages in the same order. Flexible JAMM uses asynchronous message sending, which allows the sender to continue processing before the message is actually transmitted. Message handling is optimized by requiring a message handler to declare the specific message types it handles.

Flexible JAMM implements a new approach to flexible collaboration transparency, demonstrating that it is possible to bring collaboration transparency closer to the advantages afforded by collaboration awareness. The aim of this approach is to allow a collaboration to flow naturally within the range of tightly to loosely coupled collaboration styles.

Chapter 6

Evaluation of Flexible Collaboration Transparency

Chapter 4 criticized conventional collaboration-transparency systems because they impose a tightly coupled style of collaboration. However, when collaborators are working closely together, a conventional collaboration-transparency system should be adequate. The approach to flexible collaboration transparency presented in Chapter 5 aims to provide equivalent support for tightly coupled tasks and to extend the conventional concept of collaboration transparency by additionally supporting loosely coupled tasks. By supporting both ends of the range of collaboration styles, a flexible collaboration-transparency system should allow a collaboration to flow naturally between the two.

This chapter evaluates how well the flexible collaboration-transparency system, Flexible JAMM, achieves these goals from both the user and developer points of view. This chapter consists of two parts. The first part contains a formal usability evaluation that compared Flexible JAMM versus a representative conventional collaboration-transparency system, Microsoft NetMeeting (Microsoft Corp., 1998). The second part of this chapter contains an informal evaluation from the perspective of developers using Flexible JAMM as a simple groupware toolkit.

6.1 Usability Evaluation

Flexible JAMM provides features previously seen only in collaboration-aware applications: concurrent work, relaxed WYSIWIS and detailed group awareness. Each of these features has been shown to be useful in the context of an application designed to support collaborative work (Ellis & Gibbs, 1989; Gutwin & Greenberg, 1998; Stefik *et al.*, 1987). By introducing new interface elements (i.e., radar-views and concurrent editing) into a legacy, single-user application, it was possible that users' unfamiliarity with the new interface items would offset the advantages provided by a flexible collaboration-transparency system. It is important to be sure that Flexible JAMM does not detract from the capabilities commonly provided by conventional collaboration-transparency systems. Therefore, I conducted a study to evaluate whether this flexible collaboration-transparency system did indeed provide better support for loosely coupled collaborative tasks and did not adversely impact the performance of tightly coupled collaborative tasks.

6.1.1 Procedure and Methods

The study consisted of two independent variables: (1) the type of collaboration-transparency system (flexible versus conventional) and (2) the type of task (loosely versus tightly coupled).

Collaboration-transparency Software The study compares the flexible collaboration-transparency system, Flexible JAMM version 1.0, versus a representative conventional collaboration-transparency system, Microsoft NetMeeting version 2.0. NetMeeting was used because it is freely available and is perhaps the most commonly used collaboration-transparency system today.

Flexible JAMM and NetMeeting have different target platforms. Flexible JAMM can share any serializable Swing-based Java application, and NetMeeting can share any Windows-based application. Since Java applications can be run on Windows, NetMeeting can share the same applications that Flexible JAMM can. For the evaluation, I used an application that can be shared by both, a Java-based text editor, called Notepad, which is shown in Figure 6.1 being shared via Flexible JAMM.

Tasks The tasks in this study involved editing text. I was interested in evaluating the effectiveness of the two collaborative software systems for tightly versus loosely coupled collaborations. The tightly coupled task, called **Copy Edit,** was modeled after a real-world scenario in which a

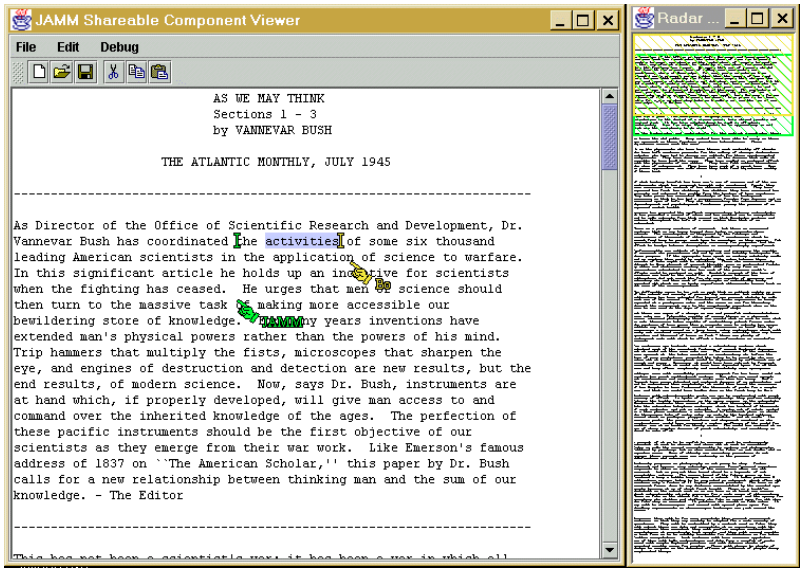


Figure 6.1: The experiment's shared text editor application, Notepad, shared via Flexible JAMM.

manuscript editor guides an author in real time to make changes to an article. This is a text-editing task analogous to the equipment-assembly task of Chapanis' (1975) communication study in which a **source** guided a **seeker** to construct a piece of equipment. In the current study's text-editing task, participants shared an electronic copy of a document that contained errors. One participant assumed the role of a manuscript editor and was given a hard copy of the seven-page document with ten corrections highlighted by ***bold-italic*** type. The other participant, in the role of the author, was the only one allowed to make the changes to the text. The manuscript "editor" was allowed to use the application to guide the "author," but was not permitted to directly change the text.

The scenario used to evaluate the software in a loosely coupled collaborative task modeled coauthors writing independently in an article. In this task, called **Text Entry,** the participants were given a two-paragraph article to enter into the editing software. The paragraphs had the same number of words (107) and approximately the same number of characters (601-611). The participants were instructed to each enter one paragraph and decided between themselves who entered which. For both tasks, participants were instructed to "work as quickly and accurately as you can."

Table 6.1: Counterbalanced ordering of task type nested in software type. Task types: C = Copy Edit, T = Text Entry. Software types: FJ = Flexible JAMM, NM = NetMeeting. Thus, NM-T means NetMeeting was used to perform the Text Entry task.

Group Number	**Order** 1	2	3	4
1	FJ-C	FJ-T	NM-C	NM-T
2	FJ-C	FJ-T	NM-T	NM-C
3	FJ-T	FJ-C	NM-C	NM-T
4	FJ-T	FJ-C	NM-T	NM-C
5	NM-C	NM-T	FJ-C	FJ-T
6	NM-C	NM-T	FJ-T	FJ-C
7	NM-T	NM-C	FJ-C	FJ-T
8	NM-T	NM-C	FJ-T	FJ-C

Experiment Design

This study used a two-factor factorial in a randomized complete block design. The task type was one factor and the collaboration-transparency system was the other. Each pair of participants was considered a *block* and received all four treatment combinations. This within-subjects design allowed participants to the software and helped offset the notoriously high variability of groups because it assumes that each block (pair of participants) is heterogeneous (Lentner & Bishop, 1993). The order of the tasks was nested within the order of the software in a counterbalanced design shown in Table 6.1. Each order combination was randomly assigned to one group.

Participants

In preparation for this experiment, I conducted a pilot study with four pairs of participants to refine the tasks and questionnaires as well as determine the number of participants needed to ensure statistical significance in the actual experiment.

Eight pairs (thirteen male and three female) of computer science undergraduate students with a minimum typing proficiency of thirty-five words-per-minute participated in the actual study. A ninth pair (two males) also completed the experiment, but were not included in the analysis because they had been scheduled merely as a backup in case one of the other groups did not complete the study. The ninth pair's data are consistent with the analyzed results. All participants were each

paid $7.50 US. Participants were allowed to choose a partner or were paired with another by the experimenter. Some of the participants had heard of NetMeeting, but none had previously used Flexible JAMM or NetMeeting for application sharing.

Procedure

Each session took approximately one hour. Participants signed a consent form and were given a copy of it. They each then filled out a short demographic information form and took a short typing speed test. Next, they were shown a brief (approximately 1.5 minutes) instruction video, which demonstrated using the first collaboration-transparency system they would use to share the text editor application. The instruction included the features of the collaboration-transparency system they would need to know. For Flexible JAMM, the video defined telepointers, showed how to use the radar view, and demonstrated concurrent text entry. For NetMeeting, the video defined floor control, explained how to pass control, and demonstrated passing control between two users. Participants then practiced using the text editor shared via the collaboration-transparency system until they felt comfortable with it (typically, less than one minute). They then used the software to perform the Copy Edit and Text Entry tasks, filling out the same questionnaire after each.

Before being introduced to the next system, participants rested for five minutes during which a computer-animated cartoon was shown. Then they followed the same procedure for the second collaboration-transparency system. Following that, the participants filled out a questionnaire that asked which system they preferred overall and for each type of task. Finally, the experimenter briefly interviewed the participants about observed critical usability incidents, their impressions about both systems, and alternative interface details for Flexible JAMM.

Equipment and Layout

Typical laboratory setups for the evaluation of single-user software is inadequate to capture a collaborative session because more than one subject and computer are involved. Fortunately, I found that equipment commonly used for single-user software studies can be configured to provide sufficient, though not ideal, observation and recording of collaborative sessions.

The layout of the experiment site is shown in Figure 6.2. To use synchronous collaborative software, participants must be able to communicate. To avoid potentially confounding effects due to computer-mediated communication, such as audio and video conferencing software, participants

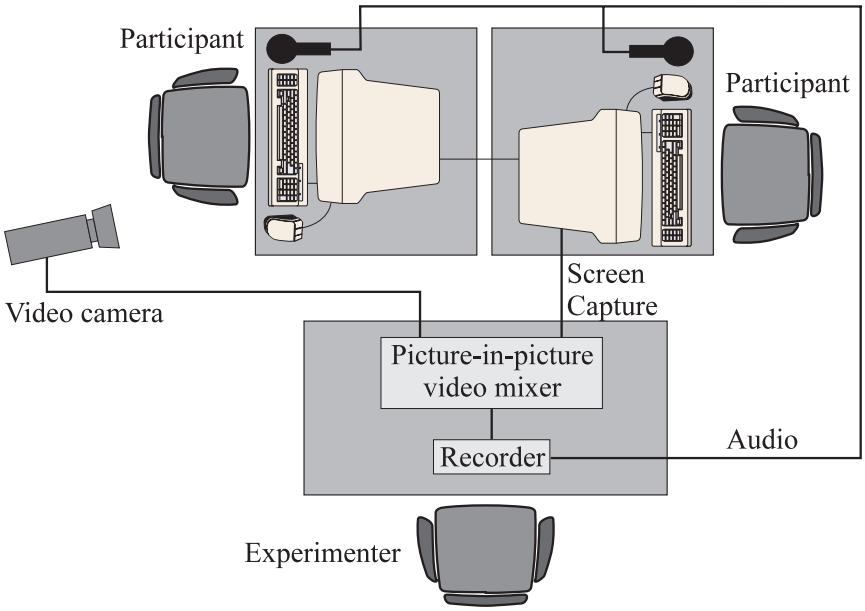


Figure 6.2: The layout of the experiment site.

were allowed to speak directly to each other. Participants were seated across from each other in the same room. The computer monitors were partially sunk into the desktop giving participants' a clear view of each other's face, but preventing each from seeing the other's screen.

The audio of each participant was recorded. Two video signals were recorded. First, the screen content of one participant was captured with a scan converter. Capturing just one screen was adequate, even though in one condition each participant can have a different view of the shared application. The second recorded video signal came from a camera behind the participant whose screen was not directly recorded. The camera's field of view encompassed the uncaptured screen and both participants. The two signals were mixed and recorded in a single "picture-in-picture" video signal such that the screen capture was full size and the camera view was one-quarter size. Figure 6.3 shows a sample of the recorded video. During recording, I dynamically moved and resized the camera view so that it did not obstruct the activity in the software seen in the screen capture.

I found that the screen capture and audio were the primary media needed to review a session. Generally, the camera view was only detailed enough to distinguish body gestures made by participants and was useful only to complete the record.

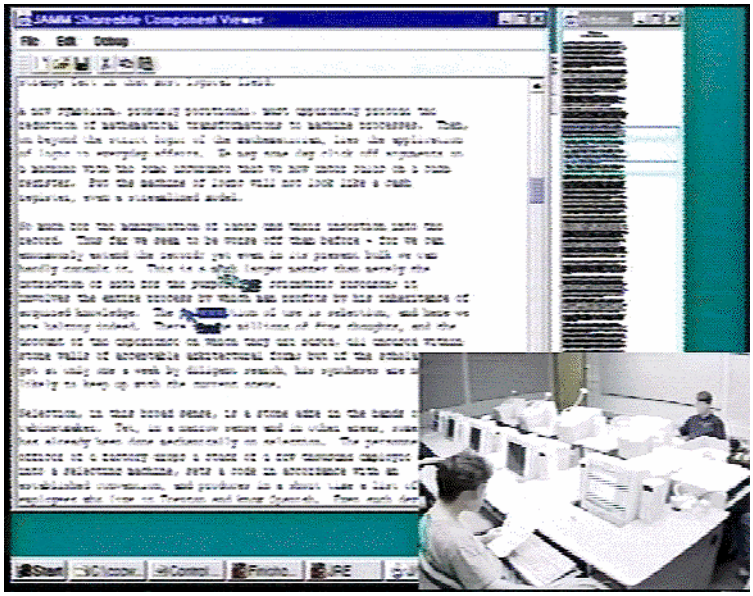


Figure 6.3: Screen capture and camera video signals merged into one "picture-in-picture" recording. The screen capture was recorded at full size and the camera was recorded at one-quarter size in the bottom right quadrant. The poor quality of the screen capture was due to converting the 800x600 pixel screen to analog video. The poor resolution was sufficient, however, because the evaluator knew the text contents (an excerpt from V. Bush (1945))

6.1.2 Results

The focus of the evaluation is the interaction of task type and collaboration-transparency system. I performed a two-way analysis of variance (ANOVA) to compare completion times, accuracy, and perceived effort. I used an alpha value of 0.05 to determine significance. Appendix A contains the raw data, which I summarize and interpret in this section.

Completion Time

The interaction of task and system is significant for completion times ($F(1,7) = 15.56$, $p = 0.0007$). The results show that participants completed the Text Entry task significantly faster using Flexible JAMM than using NetMeeting. The results are summarized in Figure 6.4 and Table 6.2. To determine the amount of difference, I conducted pre-planned comparisons of the least squares means (Tukey adjusted for multiple comparisons) of each combination of system and task. For Text Edit,

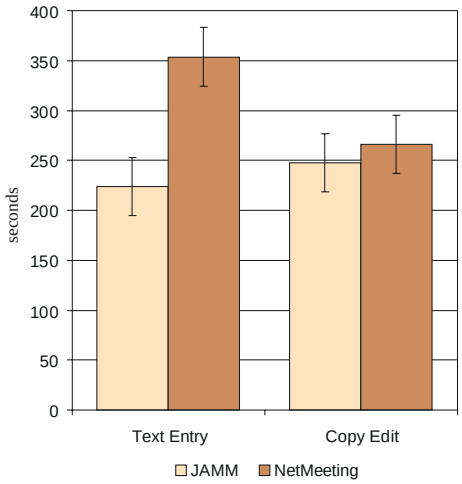


Figure 6.4: Least squares mean seconds to complete task. Error bars indicate 95% confidence interval of actual mean.

the difference in completion time using Flexible JAMM versus NetMeeting is between 74.15 and 185.34 seconds at 95% confidence. For Copy Edit, there is no significant difference between the two systems.

To confirm that the results were not confounded by transfer effects due to ordering in this within-subjects design, I also performed a between-subjects analysis using only the data from the first system used by each group. The results of the between-subjects analysis correspond to those of the within-subjects analysis with $p = 0.0001$ for Text Entry and $p = 0.364$ for Copy Edit.

For the most part, these results match the expected results. Participants completed the Text Entry task more quickly using Flexible JAMM than using NetMeeting because Flexible JAMM allows simultaneous text entry. The mean time for Flexible JAMM is not half that of NetMeeting because of several factors in addition to experimental variance. First, at the beginning of the task, up to 25 seconds was spent as the collaborators coordinated who would do what. Second, participants also spent time after completing each task deciding that they really were done. Finally, the completion time of concurrent text entry is bound by the slower typist. That is, the task is not complete until the slower typist is finished.

For the Copy Edit task, I expected completion times to not be substantially dependent on the system, but perhaps slightly longer using Flexible JAMM. In Flexible JAMM, each participant had to

Table 6.2: Least squares mean seconds to complete task. Significant difference is indicated by **bold** values for p.

	Task	
	Text Entry	Copy Edit
Flexible JAMM	223.75	247.50
NetMeeting	353.50	266.00
p	**0.0001**	0.7905

95% confidence interval = $\bar{x} \pm 29.33$

Table 6.3: Least squares mean number of errors per task and system. No significant difference was observed.

	Task	
	Text Entry	Copy Edit
Flexible JAMM	2.25	0.50
NetMeeting	1.25	0.50
p	0.1519	1.0000

95% confidence interval = $\bar{x} \pm 0.989$

scroll through the text independently, as opposed to NetMeeting where both scrolled simultaneously. In fact, however, the observed mean completion times for the Copy Edit task were slightly faster for Flexible JAMM. Factors that contributed to this result are discussed in Section 6.1.5 after the remaining results of the study are presented.

Accuracy

To measure accuracy, I counted the number of errors in the resulting text after the completion of each task. For this measure, there was no significant interaction of task and system ($F(1, 7) = 1.11$, $p = 0.3051$). The number of errors is the same regardless of the system used. The results are summarized in Table 6.3.

For Copy Edit, the results match expectations. However, for the Text Entry task I had expected the accuracy to be better using NetMeeting because only one participant can enter text at a time allowing the other to proofread for errors. In practice, only one group thought of and used this error-checking strategy while using NetMeeting.

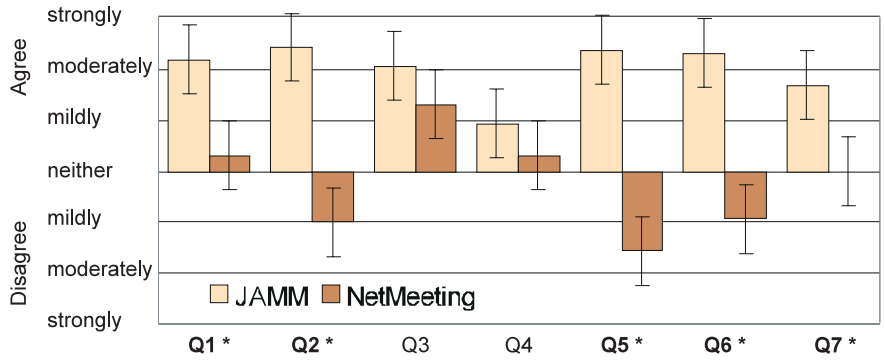


Figure 6.5: Mean responses to questions after the Text Entry task. Error bars indicate 95% confidence interval of the actual mean. Asterisks indicate significant difference in the responses for that question for Flexible JAMM versus NetMeeting.

Again, as a check against within-subjects transfer effects, I conducted a between-subjects analysis. The between-subjects results correspond to those of the within-subjects analysis with $p = 0.280$ for Text Entry and $p = 0.659$ for Copy Edit.

User Perception

Following each task, participants were asked a set of questions to measure their opinion about aspects of the collaboration-transparency system's support of collaboration for that task. The questions were formed as statements to which the participants marked the degree to which they disagreed or agreed (strongly, moderately, mildly) or neither. The seven questions asked participants their perception of (1) satisfaction with the collaborative software, (2) ability to work simultaneously, (3) ease of controlling the shared application, (4) ease of indicating text locations to partner, (5) ease of simultaneously editing text, (6) ability to have different scroll positions, and (7) ease of knowing partner's scroll position.

The responses for the Text Entry task are summarized in Figure 6.5. For all questions, the responses were significantly positive using Flexible JAMM, whereas only question 3 received a significantly positive response using NetMeeting. There are significant differences ($p < 0.001$) in the responses between the two systems for all but questions 3 and 4. These responses generally correspond to the observed use of each collaboration-transparency system in the performance of the task. The lack of distinction in question 4 is because participants generally did not indicate text locations to each other, although I had expected that they would to point out errors to each other.

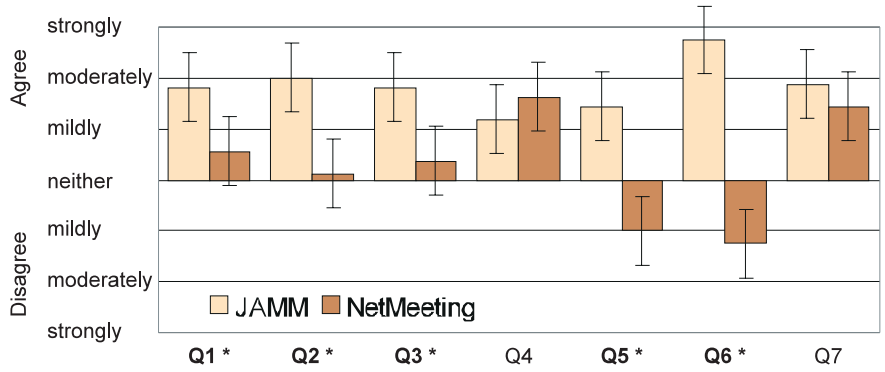


Figure 6.6: Mean responses to questions after the Copy Edit task. Error bars indicate 95% confidence interval of the actual mean. Asterisks indicate significant difference in the responses for that question for Flexible JAMM versus NetMeeting.

The significant difference in response to question 7 was unexpected. Since users share the same view in NetMeeting, they should implicitly know their partner's view position. The difference is possibly due to the implicit nature of this knowledge using NetMeeting, whereas with Flexible JAMM, view position information was made explicit.

The responses for the Copy Edit task are summarized in Figure 6.6. Participants responded significantly positively for all questions using Flexible JAMM and for questions 4 and 7 using NetMeeting. There are significant differences in the responses between the two systems for questions 2, 3, 5, and 6 ($p < 0.003$). Note that the unexpected difference in response to question 7 for the Text Entry task, described in the previous paragraph, does not show for the Copy Edit task.

I also performed a between-subjects analysis of the question responses. Unlike the objective measures of completion time and accuracy, the results of the between-subjects analysis did not exactly correspond to the within-subjects analysis for these question responses. For the Text Entry task, significant difference was seen for questions 2, 5, and 6 ($p < 0.005$), whereas only question 6 showed significant difference ($p = 0.0001$) for the Copy Edit task. The lack of distinction in the between-subjects analysis is likely due to the participants not having a base from which to make judgments. Gutwin and Greenberg (1998) observed a similar lack of distinction in a between-subjects study, which was a primary reason I chose to use a within-subjects design for this study.

Overall Preference

At the completion of all tasks using both systems, participants were asked which system they preferred overall as well as which system better supported collaboration overall, better supported the Copy Edit task, and better supported the Text Entry task. Respondents overwhelmingly selected Flexible JAMM ($\chi^2 = 16$, $p < 0.001$) for all except for support of Copy Edit, where neither system was selected over the other ($\chi^2 = 0.29$, $p = 0.9$). Table 6.4 displays the results.

Table 6.4: Preference scores for Flexible JAMM versus NetMeeting.

Question	Flexible JAMM	NetMeeting
Prefer overall	16	0
better supported:		
collaboration	16	0
Text Entry	16	0
Copy Edit	8	6

The majority of comments supported the preference ratings. However, one participant commented, "... I've really not had too much prior experience with dual typing, so being introduced to it makes it seem very extravagant." He seems to feel that concurrent editing provided by Flexible JAMM is a more novel capability than the sequential editing provided by NetMeeting. This participant was also unfamiliar with NetMeeting. Therefore, he ought to consider sequential editing equally novel. This was the only comment that indicated participants might have been influenced by the novelty of either system. Therefore, the effect due to novelty was probably slight.

6.1.3 Breakdown Analysis

I performed a **breakdown analysis** of the evaluation sessions by reviewing the audio and video record of each. Winograd and Flores (1986) describe a **breakdown** as an occurrence where a person becomes aware of items in the environment as having properties of their own, as opposed to implicitly existing as part of the environment. When using a computer to accomplish a task, for example, the user is unaware of the system unless it behaves unexpectedly. At the point of such a breakdown, the user must stop and reconsider how the system responds to their actions and how to use it. A breakdown may be as severe as a **critical incident,** which Hix and Hartson (1993) define

as having "a significant effect, either positive of negative, on task performance or user satisfaction." However, even when the effect is not significant, a breakdown identifies potential usability problems by indicating cases where the user's model of the system does not match reality.

Scrivener *et al.* (1996) have argued that analysis of breakdowns is especially effective for evaluating synchronous collaborative software. Breakdowns during cooperative work are easier to identify than during individual work because a breakdown is often naturally accompanied by communication among collaborators. In contrast, it is difficult to identify points of breakdown when evaluating single-user systems because the user may not give any observable indication that a breakdown occurred.

This evaluation followed the methodology of Scrivener *et al.*. Under their methodology, there are four categories of interactions in which breakdowns can occur: user and task, user and tool, user and environment, and user and user. The breakdowns reported here were between user and tool. For each task in this experiment, participants used two tools simultaneously: a collaboration-transparency system, Flexible JAMM or NetMeeting, and the text editor, Notepad. User-tool breakdowns therefore could occur between the user and three software tools: the collaboration-transparency system, the Notepad, or the combination of Notepad shared with the collaboration-transparency system. The breakdowns observed in each are detailed next.

Breakdowns Between Users and Flexible JAMM

Viewport Mismatch When using Flexible JAMM, each participant has a different view of the text. There was one instance during the performance of the Copy Edit task where the collaborators did not initially realize that their viewports were not aligned, although they quickly recovered. Table 6.5 contains the transcript of the breakdown. This is the sort of occurrence that I expected would lead to slower completion times in the Copy Edit task using Flexible JAMM than using NetMeeting. However, the completion-time results, described in Section 6.1.2, indicate that this sort of problem did not have a significantly adverse impact.

The radar views in Flexible JAMM show the position and extent of each participant's view (Section 5.3.5). The group transcribed in Table 6.5 used the radar view information to recover and align their views, as did the other groups who did not experience a view-misalignment breakdown. I do not consider view misalignment to be a serious problem when using Flexible JAMM for the following reasons: the collaborators recovered quickly; there was only one occurrence of such a

Table 6.5: Transcript of a breakdown when the collaborators views of the text were not aligned.

Task: Copy Edit System: JAMM Group: 4 Date: April 2, 1998

Time	Subject	Transcript	Notes
30:07	Editor	Change ... correct the word "violently."	Points at the word.
	Author	I don't see where you are.	Viewports are not aligned.
	Editor	Go ... yeah.	The author moves her viewport in line with the editor's.
	Author	Okay, I see	

breakdown out of eight groups; and the Copy Edit task completion time was slightly faster using Flexible JAMM than NetMeeting (Section 6.1.2). Nevertheless, a possible improvement to Flexible JAMM's radar view would be to allow a participant to synchronize their view position with another.

Text Highlighting Not Propagated Flexible JAMM provides independent text highlighting. That is, the text highlighted by one participant is not highlighted on other participants' replicas of the shared text. One group performing the Copy Edit task under Flexible JAMM experienced a breakdown in this regard. No transcript is available because the audio was not recorded for this session, but the breakdown was recorded in the experimenter's observation notes. The "editor" highlighted a word and asked the "author" if he could see the highlight, which he could not. This had only minor impact on the task performance, however. The "editor's" telepointer and telecaret pointed to the highlighted word, and the "author" correctly inferred which word was intended.

In comments and post-experiment interviews, subjects suggested that Flexible JAMM should propagate user-specific highlights. By propagating highlights in each user's specific color, awareness of remote user activities would be enhanced.

Breakdowns Between Users and NetMeeting

NetMeeting Display Update Two groups experienced minor breakdowns when they noticed a delay between the time they performed an edit and when the application displayed the edit when using NetMeeting. Note that the sessions were conducted on an ethernet local area network.

The breakdowns underscore the point that conventional collaboration-transparency systems are inappropriate for collaborations spanning the Internet (see Section 4.1.5).

Floor Control The faster completion time in the Text Edit task (Section 6.1.2) is primarily because Flexible JAMM allows simultaneous text entry whereas NetMeeting does not. However we also saw somewhat faster times for Flexible JAMM in the Copy Edit task, where simultaneous text entry cannot improve performance speed. Part of this can be explained by the awkwardness of passing control of the application in NetMeeting, whereas it is not necessary to pass control when entering text in Flexible JAMM.

Gaining control in NetMeeting is not difficult, but has an unexpected consequence. To gain control, the requester needs to click the mouse or press a key once. The unexpected consequence is that the control-gaining event is consumed by NetMeeting and not applied to the shared application. As a result, many times when a participant took control with a key press that they intended to enter in the text, they were surprised to see that it was not entered, as reflected in the following comments.

> I made mistakes typing the first letter, because I would forget that I had to click a key first, then start typing.

> I consistently forgot the extra step of taking control when I wanted to type material into the location indicated by my "editor."

Another problem with NetMeeting's floor control was that it did not indicate clearly who was in control. NetMeeting provides two mechanisms to indicate which collaborator has control. The first is to attach the initials of the hostname and user of the collaborator in control (e.g., "B.P." for host "Basil" and user "Pumpkin"). However, there is typically a lag of more than 5 seconds between when control changes and the displayed initials are updated, by which time control may have changed again. Because of this discontinuity, users had difficulty understanding that the initials indicated control. Participants may not have realized that the initials signified their experiment partner because the initials did not correspond to the actual name of the subject in the experiment. In actual use, the initials would likely correspond to the hostnames and login names of collaborators. The second indication of control is by displaying the words "In Control" next to the controller's name in NetMeeting's participant list. However, the list is displayed in a separate window, which is often obscured. The following comments indicate that NetMeeting's control indications are inadequate.

> While it was easy to take control, there was no strong indicator of who had control.

> Taking control in NetMeeting was kind of annoying. It wasn't always entirely clear who had control.

The problems experienced with NetMeeting's explicit floor control affected the participants' feeling of involvement. Users commented that because they had to take turns using NetMeeting, they did not feel like they were working together. The following comments about NetMeeting floor control were made before the participants had seen an alternative in Flexible JAMM.

> You can't really work together. Yes, it is real-time collaboration, but our working together required taking turns, which does not increase the efficiency of our work.

> Sitting and watching wasn't all that great.

Thus, even though passing control is fairly simple, interface difficulties and explicit turn taking diminished the benefits of collaboration. In contrast, participants commented that they felt more involved using Flexible JAMM because no floor control was required, as reflected in the following comment.

> Also, the lack of a "controller" [in Flexible JAMM] kept me involved instead of feeling passive during the editing.

Breakdowns Between Users and the Combination of Notepad with JAMM

View snapped to Caret Location There was one problem unique to the combination of using Notepad via Flexible JAMM. Whenever an edit is made in the text, the text area component (`com.sun.java.swing.JTextArea`) moves the scroll position so that the user's text caret lies within the view. This ensures that the user can see the text he or she is editing. If a user does not realize that the caret had not been in view, this sudden viewport change can be surprising. In single-user mode, a user may realize that the sudden change was the result of their immediately prior key press. In multi-user mode, however, users are less likely to make such an inference because an edit may have been generated remotely.

The following scenario illustrates the problem. User A scrolls the text past her local insert caret and is reading some text. User B enters text somewhere in the document. User A's copy of the Notepad application snaps the view to include her text caret. Note that User A's view does not necessarily show the text inserted remotely by User B because her caret may not have been near User B's insertion. It will be difficult to infer why the view suddenly shifted.

Table 6.6: Transcript of a breakdown when the scroll position unexpectedly moved.

Task: Copy Edit System: JAMM Group: 7 Date: April 2, 1998

Time	Subject	Transcript	Notes
36:20	Editor	Do you see where my cursor's blinking?	Near the top of the document.
	Author	Yeah, "their old."	
	Editor	Type in "professional" there.	
	Author	[types]	Scrollpane view jumps to bottom of text.
	Editor	That's ... not right	
	Author	Why did I go all the way to the bottom?	
	Editor	Maybe that's where your cursor was and you didn't click.	
	Author	Oh.	Moves scroll position and caret
	Editor	That's the spot	

Three groups experienced breakdowns similar to this scenario while performing the Text Entry task with Flexible JAMM. Two groups were able to infer the problem and quickly recover from it. In another case, however, the experimenter told the subject how to continue.

In these incidents, the collaborators did not discuss the problem between themselves, most likely because only one of the collaborators witnessed the problem. Table 6.6 contains a transcript of a similar situation, where the view snapping was caused by a local edit, not a remote edit as in the previous scenario. Here, the "author's" text caret is not located within his scrollpane view. As the "author" begins to enter text, his view snaps back to the insertion point. The other collaborator realizes the cause of the behavior, and they quickly recover.

This view shifting problem seriously disrupted the collaborators' work. I addressed this problem in version 1.0.1 of Flexible JAMM by overriding the view snapping behavior of the Swing text entry component. Now, the view only changes as the result of *locally generated* edits, rather than any edit. Like the other modifications I have made to the Swing library, this change is transparent to the application developer, since no application source code was modified.

Breakdowns between Users and Notepad

Notepad is a sample application included with the Swing class library as a demonstration application. While it demonstrates functionality characteristic of a simple text editor, it is not complete. Some of its problems are reported here. Fixing problems in Notepad is beyond the scope of this work.

No Word Wrap Notepad does not perform automatic word wrapping. To prevent excessive attention to this during the experiment, participants were told that the format of the text (i.e., line length and number of lines) they entered was not important to the task.

Misbehaving Paging Keys A critical problem with Notepad is that it responds incorrectly to page-up and page-down keys. It does not merely shift the view by one page of text, but jumps all the way to the top or bottom of the text. This problem initiated the view snapping breakdowns described previously in this section.

6.1.4 Participant Comments

Participants commented on problems for which there was no apparent breakdown. In some cases, the suggested remedies that would improve the system.

Flexible JAMM Telepointers

A few participants commented that Flexible JAMM's remote telepointers occasionally obstructed their view of the text. Figure 6.7 shows telepointers obstructing text. A local mouse pointer could also obstruct a user's view, but in that case the user can move the mouse and position the cursor out of the way. However, a user has no control of remote collaborators' telepointer positions.

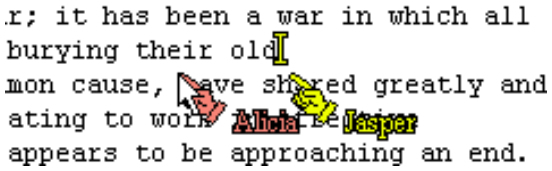


Figure 6.7: Telepointers obstruct the view of the underlying shared application.

Participants and researchers proposed various solutions: smaller pointers, translucent pointers, and pointers that fade and disappear after an idle period. The version of Flexible JAMM used for the experiment (1.0) did hide the telepointers after an idle period of one minute, but that was too long for these tasks. In Flexible JAMM 1.0.1, this period was reduced to a default of 15 seconds, based on participant's comments and the suggestion of an independent study conducted by three graduate students in computer science. Furthermore, version 1.0.1 allows each collaborator to adjust this period and other telepointer-related options, as shown in Figure 6.8. Participants liked having the telepointers, and did not suggest telepointers be removed completely. Nevertheless, participants may choose not to display telepointers in Flexible JAMM version 1.0.1.

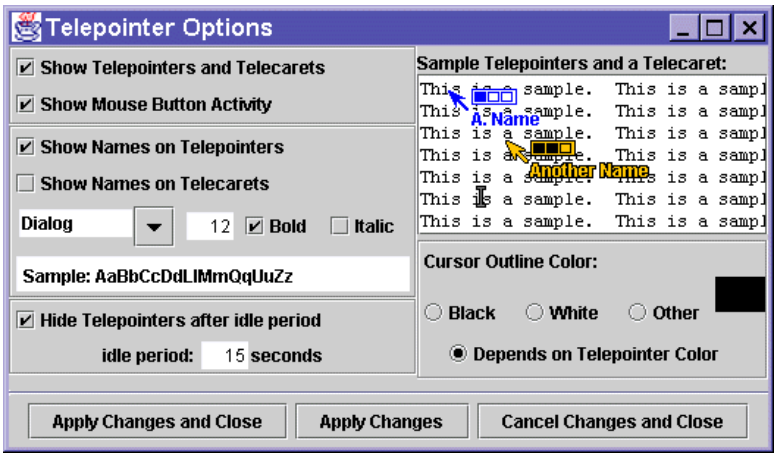


Figure 6.8: Users may select several options for displaying telepointers. The upper right region shows a sample of how telepointers will be displayed using the currently selected parameters.

Another factor that contributed to view obstruction is that Flexible JAMM currently has only two telepointer shapes of fixed size: (1) an arrow shape with a bounding rectangle of 20x20 pixels and (2) a hand shape bound within 22x23 pixels. Flexible JAMM's arrow telepointer is the same size and shape as the standard arrow cursor on the X Window System. This size is appropriate for a screen resolution of 1024x768 pixels, an increasingly common resolution. However, the equipment used in the experiment to capture and record a screen, called a "scan converter," allowed a maximum screen resolution of only 800x600 pixels. Flexible JAMM's telepointers are larger than the mouse cursor at that resolution on the evaluation machines (10x18 pixels). To see this difference, compare

the arrow-shaped local mouse cursor (10x18 pixels) in Figure 6.7 to the hand-shaped telepointer under it (22x23 pixels). Therefore, another improvement to Flexible JAMM's telepointers would be to adjust the default telepointer size depending on the screen resolution.

Flexible JAMM Participant List

Subjects suggested that Flexible JAMM should display a list of the participants in a collaboration. Version 1.0.1 of Flexible JAMM displays currently present collaborators, as shown in Figure 6.9.

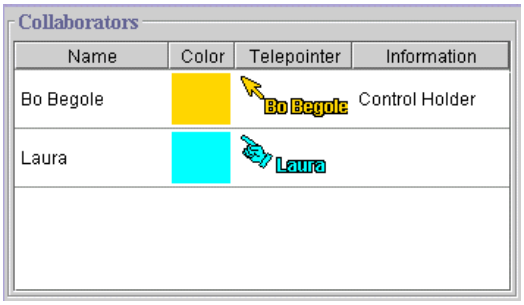


Figure 6.9: The currently present collaborators are listed. The list includes each collaborator's name, color, telepointer shape, and other information (e.g., "Left the session" or "Control Holder" as in this example).

NetMeeting Floor Control Notification Obstructs View

Some users commented that a feature of NetMeeting's floor control mechanism obstructed their view of the application. In NetMeeting, when a non-controlling participant attempts to move their mouse, a message window is displayed next to the mouse. Figure 6.10 illustrates the problem. This window disappears after 3 seconds, but during that period it obstructs the view of the shared application. Participants were unable to communicate about the underlying shared text while the message was displayed. This was a considerable problem because the mouse position was usually the locus of attention.

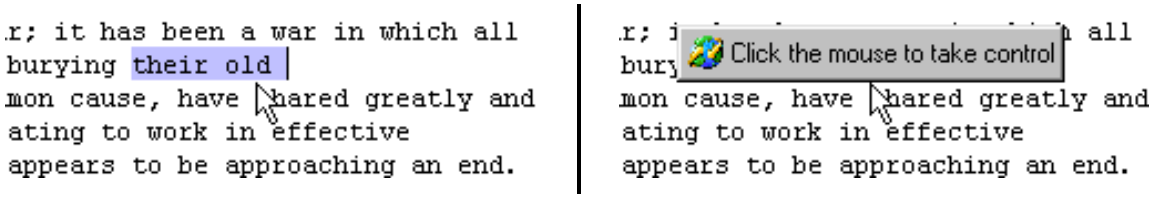


Figure 6.10: The controlling collaborator on the left has highlighted some text. When the other collaborator, on the right, attempted to move her mouse, a message window popped up at the mouse position, obstructing the highlighted text.

6.1.5 Discussion of Results

Analysis of the measured results was largely consistent with expectations. The participants completed the Text Entry task using Flexible JAMM more quickly than when using NetMeeting. This is not surprising because Flexible JAMM allows simultaneous text entry, whereas NetMeeting does not. A more interesting result was that the mean completion time for the Copy Edit task was slightly faster when using Flexible JAMM. I had expected participants to be slowed by unfamiliar multi-user interface elements in Flexible JAMM, such as having to align their views as in the breakdown described in Section 6.1.3.

Part of the explanation for slightly faster times using Flexible JAMM is that users were slowed by floor control problems when using NetMeeting (Section 6.1.3), whereas they did not need to explicitly pass control in Flexible JAMM. Participants rated NetMeeting significantly lower than Flexible JAMM in response to question 3 (ease of controlling the shared application) for the Copy Edit task, which required frequent control changes (see Figure 6.6).

Users commented that when they had to take turns in NetMeeting, they did not feel like they were working together. Despite the fact that passing control is fairly simple in NetMeeting, interface difficulties and explicit turn taking diminished the benefits of collaboration. In contrast, participants commented that they felt more involved using Flexible JAMM because no floor control was required.

The study also showed that even in the tightly coupled Copy Edit task, relaxed WYSIWIS is sometimes useful. In a few sessions, as soon as it was clear that their partner understood a specified change, the "editor" would move to the next error while the "author" made the change. The results support the aim of providing flexible support of multiple styles of collaboration in a collaboration-transparency system.

6.2 Developer Evaluation

Collaboration-transparency systems allow collaborative use of any application developed for a particular platform, such as X, Macintosh or Microsoft Windows. These are widely used platforms for which many legacy applications exist. Flexible JAMM's target platform is more limited than typical systems, being constrained to serializable, Swing-based applets as described in Section 5.3.4. For the time being, the set of legacy, single-user applications that meet these constraints is fairly small. However, I expect the set of shareable applications to grow as Swing becomes part of the core Java library (it is currently available as a free, but separate, package). I emphasize, however, that Flexible JAMM is not intended as a commercial system, but demonstrates the approach to flexible collaboration transparency outlined in Chapter 5.

Furthermore, Flexible JAMM is not only intended as a collaboration-transparency system, but also serves as a simple-to-use groupware toolkit. A developer who is cognizant of Flexible JAMM's capabilities might more easily build a single-user application that is meant to be shared than to build a multi-user application explicitly. For example, a text editing application that includes scrollable panels and text areas will become multi-user aware at the time of sharing. Thus, incorporating multi-user radar panels and concurrent text components in a collaborative application has the same development cost as including single-user scroll panels and text components in a single-user application.

Unlike conventional groupware toolkits, where developers must make explicit calls to the toolkit functionality, the Flexible JAMM developer need only ensure that the application conforms to two constraints: (1) it must be serializable and (2) use Swing (see Section 5.3.4 for more about Swing and serialization). This section informally reviews developer impressions of the difficulty of these requirements.

6.2.1 Procedure

I gave three experienced Java developers the original Java source code of a physics simulation applet. Figure 6.11 shows a screen-captured image of the test applet, by Serge G. Vtorov (1998), which simulates the wave-interference properties of light. The source code consisted of ten files with a total of 702 lines. The developers' task was to modify the source code so that it was serializable and used the Swing graphical user interface library.

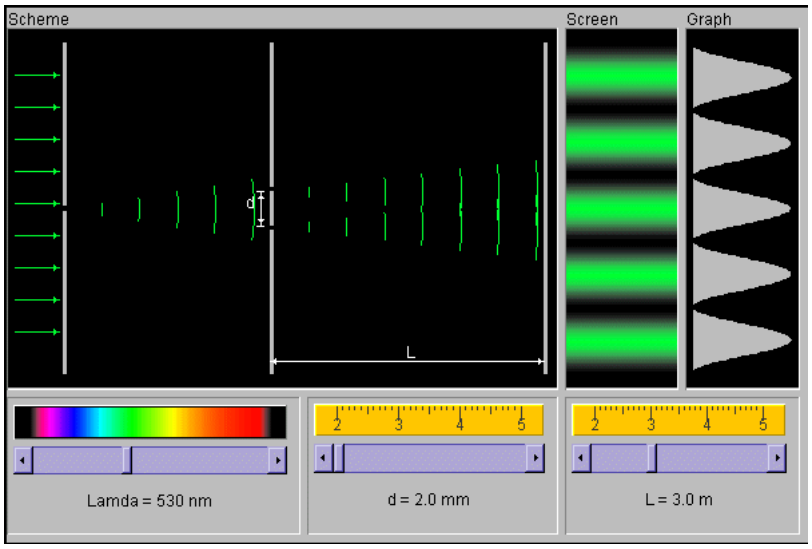


Figure 6.11: A single-user physics simulation of the wave-interference properties of light. The simulation recreates the well-known "two-slit" experiment.

I gave the developers a set of instructions that specified how to modify an arbitrary, existing Java application to allow it to be shared under Flexible JAMM (Begole, 1998). The instructions consist of two parts: (1) how to convert the application to Swing and (2) how to make the application serializable. The Swing section consisted of ten instructions in four subsections. The serialization section consisted of seven instructions. The instructions are in a Hyper-Text Markup Language (HTML) document that has a printed length of approximately four and a half pages. The instructions are available at the Flexible JAMM web site (Begole, 1997).

6.2.2 Results

Among the most important results were the developers' suggestions for improvements to the conversion instructions. I incorporated each developer's suggestions into the instructions before the next developer began work. Thus, the instructions were iteratively refined during this evaluation.

The average time required to make the conversion was approximately two hours. Developers indicated that the conversion was straightforward. Two developers complained of having to fix "bugs" that existed in the original applet and were only brought to light using Swing. Two developers felt that novice programmers might encounter difficulties.

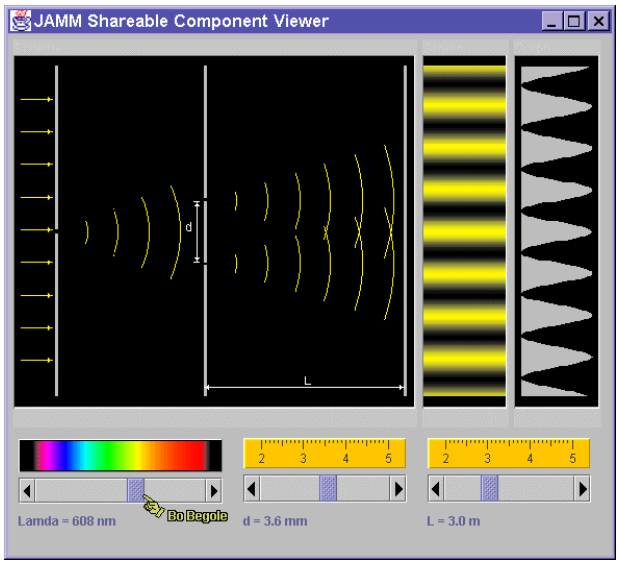


Figure 6.12: The single-user physics simulation seen in Figure 6.11 shared via Flexible JAMM. To be shared, the applet was converted to be serializable and use the Swing user interface library.

All conversions met the criteria of being serializable and using Swing. However, the first conversion did not run correctly after having been serialized and reconstructed because a critical association between two of the objects was lost. I modified the instructions to have developers look for and correct such cases. The second and third conversions, which used the modified instructions, worked correctly.

The first two conversions contained an error regarding drawing text in Swing. Notice that the column labels, seen at the top of the applet in Figure 6.11, are not displayed in the converted applet, shown in Figure 6.12. This is because, unlike AWT, Swing does not implicitly set the text drawing color to a component's foreground color. The labels in the converted applet are unseen because they are drawn in the same color as the background. After the first conversion, I included steps to correct this error in the conversion instructions. The second developer said that he noted the instruction, but did not see an error on his platform's (Free BSD) implementation of Java, and therefore did not include the fix. The third conversion worked correctly.

Flexible JAMM is available for download (Begole, 1997) on the World Wide Web. Version 1.0.x was downloaded 423 times in the two-month period between Jun 25, 1998 and Sep 2, 1998.

A few of the downloaders have sent feedback, though most have not. Andreas Ruff, a graduate student in computer science at the University of Stuttgart, Germany, provided some comments from a developer's perspective. He developed a simple applet and demonstrated sharing it under Flexible JAMM to colleagues. In addition to making a suggestion to allow hiding telepointers, he gave the following report after his demonstration (Ruff, 1998).

> Everyone was quite astonished [at] how easy it is to develop applications/applets for JAMM. I showed my (very) simple ButtonTest-Example and shared it with JAMM. No one thought that it was possible to share it without programming special "administration-" or "sharing-information."

6.2.3 Discussion of Developer Evaluation

A secondary goal of Flexible JAMM is to act as a simple-to-use groupware toolkit. Beyond the development of the application, there are only two requirements to allow the application to be shared under Flexible JAMM: (1) the application must use the Swing interface library and (2) all classes in the application must be serializable.

Experienced Java developers had little difficulty modifying an existing Java applet to be shared via Flexible JAMM. Following a four-and-one-half page set of instructions, developers were able to convert an applet of ten files in an average period of approximately two hours. In contrast, the instructions to create a shareable application using GroupKit 5.0, which gives developers greater control of the distributed aspects of the application than Flexible JAMM does, consist of approximately 39 printed pages.

6.3 Summary

This chapter reports evaluations of Flexible JAMM's effectiveness in supporting the two communities it aims to benefit: users and developers.

The first part of this chapter describes an empirical usability study of users performing a loosely coupled task, Text Entry, and a tightly coupled task, Copy Edit, using Flexible JAMM versus using a representative conventional collaboration-transparency system, NetMeeting. Eight pairs of computer science students participated. The study sought to confirm that users perform faster using Flexible JAMM for the loosely coupled task because Flexible allows simultaneous text entry, whereas NetMeeting does not. The study also hoped to confirm that there was no difference for the tightly

coupled task. However, there was concern that learning and using new interface elements (radar view, telepointers and telecarets) would adversely impact performance times when using Flexible JAMM.

Analysis of the measured results matched expectations in terms of task completion time: participants were able to complete the Text Entry task more quickly using Flexible JAMM than using NetMeeting. Participants completed the Copy Edit task slightly faster using Flexible JAMM than using NetMeeting, but the difference was not significant. The number of errors was not significantly different for either task using either system. Users rated Flexible JAMM significantly positively for both tasks in a series of seven questions, whereas NetMeeting was rated significantly positively for only one question in the Text Entry task and two in the Copy Edit task. The responses were significantly different between Flexible JAMM and NetMeeting in five of the seven questions for Text Entry and four questions for Copy Edit. Participants greatly preferred Flexible JAMM overall and commented that they felt involved in the collaboration, whereas with NetMeeting, the explicit turn taking made them feel like they were not working together.

Breakdown analysis and participant comments identified several areas needing improvement in each collaboration-transparency system. Problem areas in Flexible JAMM included viewport misalignment, no remote text highlighting, unexpected viewport shifting, telepointer obstruction, and lack of a participant listing. I incorporated participant-suggested solutions to these problems into version 1.0.1 of Flexible JAMM following the study. For NetMeeting, participants identified problems in the following areas: display update lag, lost key presses, explicit floor passing, weak indication of who has control, and the floor control notification obstructs the view.

The second part of this chapter contains an informal evaluation from the perspective of developers using Flexible JAMM as a simple groupware toolkit. Three developers were asked to modify an existing applet to meet the two requirements of sharing it via Flexible JAMM: (1) the application must use the Swing interface library and (2) all classes in the application must be serializable. Following a set of instructions, the developers were able to convert the applet of ten files in an average of approximately two hours. They indicated that the conversion was not difficult. The instructions were iteratively refined during this evaluation as each developer suggested improvements. There was a major error in the first conversion. After updating the instructions to correct this error, subsequent conversions worked correctly.

The two evaluations confirm that Flexible JAMM contributes to the field of collaborative computing in ways that benefit both users and developers. The usability study in Section 6.1 demonstrates that it is possible to bring collaboration transparency closer to the advantages afforded by collaboration awareness. By supporting the range of tightly to loosely coupled collaboration styles, Flexible JAMM allows a collaboration to flow naturally between the two. This capability enhances the collaborative use of the growing body of legacy, single-user applications.

The developer-oriented evaluation in Section 6.2 illustrates that Flexible JAMM also serves as a simple-to-use groupware toolkit. Multi-user scroll panels and concurrent text components can be no more difficult to include in a collaboration-aware application than single-user scroll panels and text components are to include in a single-user application. Future groupware toolkits can and should include multi-user interface components that are easy for developers to include in collaborative applications.

Chapter 7

Conclusions

In this research, I have examined the usability and implementation differences between conventional collaboration-transparency systems and collaboration-aware applications and toolkits. I examined how the use of particular distribution architectures affects support for collaboration. The claim behind this research was that many of the benefits currently seen in collaboration-aware applications can be provided in a collaboration-transparency system. To that end, I designed an alternate approach to collaboration transparency, implemented a prototype system, and evaluated its effectiveness compared to a conventional collaboration-transparency system. This chapter summarizes the main results of this research, discusses their significance, and offers suggestions for future research.

7.1 Summary of Results

This dissertation makes several contributions.

- *Critique of conventional collaboration-transparency systems*
  Previously, researchers have made scattered complaints about conventional collaboration transparency, but no thorough critique exists. My analysis finds that conventional collaboration-transparency implementations do not use network resources efficiently and they impose an inflexible, tightly coupled style of collaboration because they do not adequately support important groupware principles: concurrent work, relaxed WYSIWIS, group awareness, and inherently collaborative tasks.

- *Application replication is feasible for collaboration transparency.*
  Replication has advantages over centralization that are important for Internet-wide collaboration: lower network bandwidth requirement, and faster application response. In addition, some degree of replication is necessary to support groupware usability requirements: concurrent work, and independent views. Previously, however, replication was deemed impractical for collaboration transparency. Although it remains true that consistency cannot be guaranteed when replicating some applications (i.e., those with time-dependent user interactions), this dissertation demonstrates that replication is viable for many applications.

- *A system resource can be shared via replicated proxies of the centralized resource.*
  In a replicated environment, each copy executes on a different host, which may provide different responses to requests for system resources, such as files, sockets, and time. This dissertation presents an approach that ensures each replica will receive the same data for the same request.

- *Multi-user scroll panels, telepointers, and concurrent text components can be as easy to include in a groupware application as single-user versions are to include in a single-user application.*
  Few groupware toolkits provide multi-user versions of standard interface components, and those that do so require additional programming over the single-user version to update remote users. However, the object-replacement used by Flexible JAMM demonstrates that it is possible to include multi-user versions of some standard single-user components with no additional programming by the toolkit user.

- *Many of the features of collaboration-aware applications can be provided* transparently *in shared single-user applications.*
  This is the primary claim investigated in this dissertation and is supported by the above results. Section 5.2 describes a general design for a new approach to collaboration transparency, and Section 5.3 describes an implementation of that approach, called Flexible JAMM. Flexible JAMM demonstrates that a collaboration-transparency system can promote some single-user, collaboration-unaware applications to the level of support for collaboration previously provided only in applications specifically designed for collaborative use.

- *Flexible collaboration transparency requires the application platform to have four capabilities.*
  There are four requirements of an application platform needed to implement the replicated, object-replacement approach to flexible collaboration transparency (Section 5.2.3): process

migration, run-time object replacement, dynamic binding, and the ability to intercept and introduce low-level user input events.

The work done for this dissertation makes several secondary contributions. A prototype flexible collaboration-transparency system, Flexible JAMM (Section 5.3), is freely available on the Internet (Begole, 1997). Flexible JAMM includes many reusable multi-user components: telepointers, participant information components, a multi-user scrollable panel, and a multi-user concurrent text editor. These components are also used in a collaboration-aware interactive visualization system called Sieve (Isenhour *et al.*, 1997). The development of these components contributed to the design and development of a collaborative learning environment, the LiNC (Learning in Network Communities) virtual school.

Finally, the evaluation of flexible versus conventional collaboration transparency (Section 6.1) corroborates the results of previous studies of group awareness interface techniques (Gutwin *et al.*, 1996b; Gutwin & Greenberg, 1998). Additionally, the evaluation substantiates previous claims that implicitly passing control among collaborators is preferred over explicit control passing in some types of collaboration (Crowley *et al.*, 1990; Lauwers, 1990; McKinlay *et al.*, 1994).

7.2 Discussion

Because it contains features of both transparency and awareness, flexible collaboration transparency has implications for how we view the capabilities of each. First, the target platform for *transparent* sharing is shifted from the operating/windowing system level to the interface library level. Second, groupware toolkits can provide easy-to-use multi-user interface components.

7.2.1 Transparency

Collaboration-transparency systems generally allow the sharing of any application written for a particular platform (e.g., X or Windows). Due to the constraints outlined in Section 5.3, Flexible JAMM is only able to *transparently* share serializable, Swing-based Java applications. We saw in Section 6.2 that converting an existing AWT-based Java application to Swing and serialization is straightforward. Such conversion, however, breaks the notion of collaboration *transparency* because the application source code is modified.

Flexible JAMM is not a collaboration-transparency system at the windowing system level, as are conventional collaboration-transparency systems, but at the interface library level. Although this clearly limits the set of shareable applications, the boundary is not delimited by hardware or operating/windowing systems, by which the sets of applications that are shareable under conventional systems are constrained.

7.2.2 Multi-user Interface Components in Groupware Toolkits

In addition to demonstrating a new approach to collaboration transparency, Flexible JAMM has implications for collaboration-aware groupware toolkits. Flexible JAMM may be viewed as a simple-to-use groupware toolkit. A developer who is cognizant of Flexible JAMM's capabilities might more easily build a single-user application that is meant to be shared than to build a multi-user application explicitly. For example, a text editing application that includes scrollable panels and text areas will become multi-user aware at the time of sharing.

Groupware toolkits can provide multi-user interface components that are no more difficult to include in a collaborative application than to include single-user interface components in a traditional single-user application. Currently, the few groupware toolkits that do provide multi-user interface components require the toolkit user to explicitly send notifications of local changes to remote participants. Therefore, considered as a groupware toolkit, Flexible JAMM advances the state of the art by requiring no additional programming by the toolkit user to include standard multi-user interface components.

7.3 Directions for Future Work

This research has brought to light several areas that deserve further exploration.

7.3.1 Multi-user Interface Components

An obvious extension to Flexible JAMM would be to create and integrate more multi-user interface components. For example, multi-user menus that only drop down on the replica on which they are invoked, but that notify remote users of the activity.

7.3.2 Alternatives to Object Replacement

The object-replacement strategy works well, but has two limitations: (1) objects cannot be replaced after the point of sharing an application and (2) subclasses of replaceable classes cannot be replaced.

Flexible JAMM replaces the target objects via a mechanism provided by JOS at the point when the application is initially shared. Unfortunately, no such mechanism is available after this initial point of sharing. Therefore, if the shared application later creates a new instance of a replaceable object, such as a scrollable panel, Flexible JAMM is unable to replace it with a multi-user extension. The other potential problem would arise if the application developer has subclassed a replaceable class. Because the application presumably depends on the specialized behavior of the subclass, it is not safe to replace it with a multi-user–aware object that does not contain the same specialized behavior. In both of these cases, Flexible JAMM does not replace the target object, thus falling back to the sharing behavior of conventional collaboration-transparency systems and losing the advantages of flexible collaboration transparency.

There are viable alternatives to full object replacement. Many classes of the Swing interface library expose enough information to provide multi-user capabilities externally without replacing the object. For example, the Swing text components notify registered observers of each text insertion and deletion, making it possible to attach an operational transformation algorithm to an existing text object. Additionally, Swing allows a system to attach observers to a scrollable panel which are notified when the scroll position changes. It is possible to use this information to allow independent scroll positions and provide location awareness in an external radar view. Attaching observers is possible in Swing, but not in other interface class libraries, such as the Abstract Window Toolkit (AWT) because it does not expose sufficient information. Full object replacement remains a viable strategy for such a library.

7.3.3 Barriers to Adoption of Synchronous Collaborative Software

There are many examples of successful asynchronous collaborative applications that are commonly used by casual computer users: electronic mail, discussion groups (e.g., Usenet), and publishing and reading documents on the WWW. In contrast, there are relatively few synchronous collaborative systems that are widely used. Although the Internet's population has expanded greatly, it remains a place where workers are relatively isolated. Given that people commonly carry out synchronous

collaborations in "real life," we might expect synchronous collaborative applications to be used more commonly than we see currently.

Past researchers (Ellis *et al.*, 1991; Grudin, 1988) have investigated the adoption of groupware systems within an organization, and have identified several factors that affect the adoption of groupware: critical mass of users, cost–benefit imbalances, organizational politics, and collaborative application usability. Groupware continues to be an important area of research and, in addition to the previously-identified barriers to groupware adoption, the following are some areas that I believe need investigation before we see a dramatic increase in the use of synchronous collaborative systems.

Network Connection Persistence

Computer users outside of academic and scientific research communities have only recently acquired wide-area network access to each other via the Internet. In the past, corporate networks were connected for only brief periods of time to transfer batches of data. Such aperiodic connection permits asynchronous software, such as email and discussion groups, but does not allow synchronous interactions. Today, business users typically have persistent connections. Since synchronous interaction has become feasible, we have seen the growing use of some synchronous collaborative systems, such as multi-player games (e.g., MUDs and Diablo), and text chat systems (e.g., Internet Relay Chat (IRC), and I Seek You (ICQ)). As network connections become more and more persistent, people may become more accustomed to the notion that real-time computer-based interactions are possible, and they will seek systems that provide such capabilities.

Concurrent with the trend toward connection persistence, there is a conflicting trend in the use of portable devices (i.e., laptops and personal digital assistants (PDA)) that follow the previous model of periodic connection for batch transfer. So, there is a tension for new software systems to take advantage of persistent connections while supporting mobile, aperiodically-connected devices. Dourish (1998) has proposed one programming paradigm that supports various degrees of connection. This and other approaches need to be studied.

Collaborator Management

Potential collaborators need to be able to find each other in order to form groups. There are many competing formats for maintaining individual user information. A well-known example of one such format is the `\etc\passwd` file used by all UNIX systems. Although this format is simple

and prevalent, it is not used by many systems that require similar user information, such as e-mail servers. In fact, even among systems that provide capabilities similar to each other, different formats are used. For example, two instant messaging systems, America OnLine's (AOL) Instant Messanger (AIM) and ICQ, use different formats for participant information (both of those technologies are now owned by AOL, which may lead to consolidation in the future).

These competing user information formats add to the startup cost of using groupware systems because each user's information must be entered into each new system. Standards for user information, such as X.500 (ISO, 1989), and access mechanisms, such as the Lightweight Directory Access Protocol (LDAP) (Wahl *et al.*, 1997), could ease the adoption of groupware systems.

Coordination and Session Management

Real-time collaborations may be scheduled or opportunistic. In either case, potential collaborators need to be able to form, find, and join collaborative sessions. Currently, each real-time collaborative system uses a different session coordination mechanism. Standard session registration and lookup mechanisms, similar to that provided by X.500 and LDAP for individuals and groups, could ease the integrated use of groupware systems from different vendors.

Suggestions to Improve System Support for Groupware

Presently, using a synchronous collaborative system, like Microsoft NetMeeting, requires users to make several explicit actions. They must start the collaborative system, register and announce the session, coordinate the joining of other users, and use potentially unfamiliar multi-user interfaces. Such startup requirements discourage the use of synchronous collaborative systems.

This and prior work in collaboration transparency brings to light several limitations in existing operating/windowing systems with respect to supporting collaborative software. In addition to suggestions made by prior researchers (Lauwers *et al.*, 1990; Minenko, 1996), the following features should be included in operating/windowing systems to support the implementation and use of collaborative software. Currently, collaboration transparency must be implemented as a layer between the operating/windowing system and the shared application. However, with the following capabilities, it would be possible to build application sharing into the operating/windowing system itself. Enabling collaboration at the operating/windowing system level would facilitate users' transitions between individual and cooperative work.

- *Support image copy for process migration.*

  Some means of process migration must be used to give newcomers a copy of the shared application in its current state. Event logging and replay can usually be used, but can require considerably more time than obtaining and transferring an image of the process (i.e., address space, call stack, and program counter). Image copy requires support from the system to obtain process information and reconstruct it at the receiving end.

- *Allow run-time substitution of one object with another.*

  The platform should provide a mechanism to replace one object with another at run time. This way, the replacement can occur after some point when the user decides to share an application. Such run-time replacement maintains transparency because no source code is modified.

- *Use dynamic binding.*

  User interface classes should be programmed to use dynamic binding so that a replacement object will respond to invocations after it has replaced some object.

- *Support input from multiple sources.*

  Current operating/windowing systems assume no more than one simultaneous pointing device and character input device. A mouse input event does not need to specify the source of the event because there is only one possible source. However, in multi-user applications, and multi-input environments in general, the application must be able to determine the source of the input and act accordingly. To support multiple inputs, operating/windowing systems should

  - provide the source of an input event, including the host and a unique device identifier.
  - allow multiple simultaneous screen pointers (e.g., mouse cursors and text-entry carets).

7.3.4 Application Session Recording

In addition to applications in cooperative work, intercepting input events may be used to record the use of an application. There are two areas where such a recording might be useful: (1) regression testing newer versions of a system and (2) recording usability evaluations.

Regression Testing

Event logging and replay could be applied to regression testing of newer versions of a system. An initial session can be recorded by logging the input events. Later, the events can be replayed on newer versions of the system, to ensure that some revision does not break another portion. When necessary, the event log could be edited by adding or removing event sequences. Thus, the application interface could change from one version to the next. Requirements and feasibility of such an approach to regression testing need to be explored.

Recording Usability Evaluation Sessions

Logging events could be used to record a session of application use in a usability evaluation. Evaluator annotations (audio or text) could be recorded, and later replayed synchronously along with the replay of user events. Recording a session in this way has three advantages over video recording. First, less hardware is needed to record a session. Currently, a digital-to-analog scan converter is physically attached to the computer video output. Second, the input logs could be used to analyze patterns of use (Kelley *et al.*, 1998; Hammontree *et al.*, 1992). Third, the image quality will be higher than video taping. For example, compare the video image of an 800x600 (the highest resolution that can be satisfactorily reproduced) display shown in Figure 6.3 on page 116 with a digital image capture of the same application shown in Figure 6.1 on page 112.

Downsides of this approach need to be explored. One disadvantage is that it may not be possible to view a session in reverse, whereas with video tape, reverse viewing is possible. However, by saving snapshots of the application state at various points, an evaluator would be able to "rewind" the session to a previous point. Other tradeoffs of this approach need to be identified.

7.4 Concluding Remarks

Flexible collaboration transparency blurs the distinction between transparency and awareness by providing many of the advantages of each approach: no extra development cost to share existing applications, lower network usage than conventional systems, concurrent work within multi-user components, independent views of shared data, and detailed group awareness information.

This dissertation demonstrates that it is possible to bring collaboration transparency closer to the advantages afforded by collaboration awareness. By supporting the range of tightly to loosely

coupled collaboration styles, flexible collaboration transparency aims to allow a collaboration to flow naturally between the two. This capability enhances the collaborative use of the growing body of legacy, single-user applications. Furthermore, the prototype system demonstrates that collaboration-aware toolkits can include multi-user versions of some standard single-user components that require no collaboration-specific programming by the toolkit user. Thus, the results of this research advance the state of the art in both collaboration-transparency systems and collaboration-aware toolkits.

References

Abdel-Wahab, Hussein, & Feit, M. 1991. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. *Pages 159–167 of: Proceedings of IEEE TriComm.*

Abdel-Wahab, Hussein, & Jeffay, Kevin. 1994. Issues, Problems and Solutions in Sharing X Clients on Multiple Displays. *Internetworking - Research and Practice*, **5**(1), 1–15.

Abdel-Wahab, Hussein, Kvande, B., Nanjangud, S., Kim, O., & Favreau, J.P. 1996. Using Java for Multimedia Collaborative Applications. *In: PROMS'96.*

Ackerman, Mark S., & Starr, Brian. 1996. Social Activity Indicators for Groupware. *IEEE Computer*, **29**(6), 37–42.

AIM. 1998 (Apr. 7,). AOL Instant Messenger. *America On-Line, Vienna, Virginia.* <http://www.-aol.com/aim/home.html>. last accessed: Jun 25, 1998.

Arnold, Ken, & Gosling, James. 1996. *The Java Programming Language.* The Java Series. Reading, MA, USA: Addison-Wesley.

Baecker, Ronald M. 1993. *Readings in Groupware and Computer-Supported Cooperative Work.* San Francisco: Morgan Kaufmann Publishers.

Baecker, Ronald M., Nastos, Dimitrios, Posner, Ilona R., & Mawby, Kelly L. 1993. The User-Centred Iterative Design of Collaborative Writing Software. *Pages 399–405 of: Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems.* Meetings and Collaborative Writing.

Beca, Lukasz, Cheng, Gang, Fox, Geoffrey C., Jurga, Tomasz, Olszewski, Konrad, Podgorny, Marek, Sokolowski, Piotr, & Walczak, Krzysztof. 1997. Web Technologies for Collaborative Visualization and Simulation. *Proc. of the 8th SIAM Conference on Parallel Processing for Scientific Computing.*

Begole, James "Bo". 1997 (Dec. 5,). Flexible JAMM (Java Applets Made Multiuser). *Virginia Polytechnic Inst. and State University, Department of Computer Science.* <http://simon.cs.vt.edu/jamm>. last accessed: Oct. 22, 1998.

Begole, James "Bo". 1998 (July 20,). Converting an Application to Swing and Serialization. *Virginia Polytechnic Inst. and State University, Department of Computer Science.* <http://simon.cs.vt.edu/jamm/Jammify.html>. last accessed: Oct. 22, 1998.

Begole, James "Bo", & Shaffer, Clifford A. 1997 (Feb.). *Internet Based Real-Time Multiuser Simulation: Ppong!* Technical Report TR-97-01. Virginia Polytechnic Inst. and State University, Department of Computer Science.

Begole, James "Bo", Struble, Craig A., & Shaffer, Clifford A. 1997a. Leveraging Java Applets: Toward Collaboration Transparency in Java. *IEEE Internet Computing*, **1**(2), 57–64.

Begole, James "Bo", Struble, Craig A., Shaffer, Clifford A., & Smith, Randall B. 1997b. Transparent Sharing of Java Applets: A Replicated Approach. *Pages 55–64 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-97).* New York: ACM Press.

Begole, James "Bo", Isenhour, Philip L., & Shaffer, Clifford A. 1998a (May). *JavaBeans as a Framework for Collaborative Software.* Technical Report TR-98-13. Virginia Polytechnic Inst. and State University, Department of Computer Science.

Begole, James "Bo", Rosson, Mary Beth, & Shaffer, Clifford A. 1998b. Supporting Worker Independence in Collaboration Transparency. *Pages 133–142 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-98).* New York: ACM Press.

Bhola, Sumeer, Banavar, Guruduth, & Ahamad, Mustaque. 1998. Responsiveness and Consistency Tradeoffs in Interactive Groupware. *Pages 79–88 of: Proceedings of the ACM conference on Computer-Supported Cooperative Work (CSCW'98).* New York: ACM Press.

Bly, Sarah A., Harrison, Steve R., & Irwin, Susan. 1993. Media Spaces: Bringing People Together in a Video, Audio and Computing Environment. *Communications of the ACM*, **36**(1), 28–47.

Brown, N., & Kindell, C. 1996. *Distributed Component Object Model Protocol - DCOM/1.0.*

Burridge, Rich. 1998 (Apr.). *Java Shared Data Toolkit User Guide.* User Guide. Sun Microsystems, Inc., Mountain View, CA. <http://java.sun.com/products/java-media/jsdt/jsdt-userguide-1.4.pdf>. last accessed: Oct. 22, 1998.

Bush, Vannevar. 1945. As we may think. *Atlantic Monthly, 176, July 1945*, 101–108. Reprinted in Greif (1988).

Buxton, William A. S. 1992 (May). Telepresence: Integrating Shared Task and Person Spaces. *Pages 123–129 of: Proceedings of Graphics Interface '92.*

Cederqvist, Per, *et al.* 1993 (Nov.). *Version Management with CVS.* Signum Support AB, Box 2044, S-580 02 Linkoping, Sweden. <http://www.loria.fr/~molli/cvs/doc/cvs_toc.html>. last accessed: Oct. 22, 1998.

Chabert, Annie, Grossman, Ed, Jackson, Larry S., Pietrowiz, Stephen R., & Seguin, Chris. 1998. Java Object-sharing in Habanero. *Communications of the ACM*, **41**(6), 69–76.

Chapanis, Alphonse. 1975. Interactive Human Communication. *Scientific American*, **232**(3), 36–42.

Chung, Goopeel, & Dewan, Prasun. 1996. A Mechanism for Supporting Client Migration in a Shared Window System. *Pages 11–20 of: Proceedings of the ACM Symposium on User Interface Software and Technology.* Papers: CSCW and Agents.

Chung, Goopeel, Jeffay, Kevin, & Abdel-Wahab, Hussein. 1993. Accommodating Latecomers in Shared Window Systems. *Computer*, **26**(1), 72–74.

Clemente, Peter C. 1998. *State of the Net: The New Frontier.* New York, NY: McGraw-Hill.

Coulouris, George, Dollimore, Jean, & Kindberg, Tim. 1994. *Distributed Systems: Concepts and Design.* Second edn. Reading, MA, USA: Addison-Wesley.

Crowley, Terrence, Milazzo, Paul, Baker, Ellie, Forsdick, Harry, & Tomlinson, Raymond. 1990. MMConf: An Infrastructure for Building Shared Multimedia Applications. *Pages 329–342 of:*

Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work. Systems Infrastructure for CSCW. Los Angeles, California: ACM Press.

Data Connection, Ltd. 1998 (Aug. 18). Data Connection Conferencing - DC-WebShare. *Data Connection, Ltd.* <http://www.datcon.co.uk/conf/webshare.htm>. last accessed: Oct. 22, 1998.

Dewan, Prasun, & Choudhary, Rajiv. 1991a. Flexible User Interface Coupling in a Collaborative System. *Pages 41–48 of: Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*. Group Use of Computing.

Dewan, Prasun, & Choudhary, Rajiv. 1991b. Primitives for Programming Multi-User Interfaces. *Pages 69–78 of: Proceedings of the ACM Symposium on User Interface Software and Technology*. CSCW.

Dewan, Prasun, & Choudhary, Rajiv. 1995. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer-Human Interaction*, **2**(1), 1–39.

Dourish, Paul. 1996a. Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit. *Pages 268–277 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Dourish, Paul. 1996b. *Open Implementation and Flexibility in CSCW Toolkits*. Ph.D. thesis, University of London, Department of Computer Science.

Dourish, Paul. 1998. Using Metalevel Techniques in a Flexible Toolkit for CSCW Applications. *ACM Transactions on Computer Human Interaction (TOCHI)*, **5**(2), 109–155.

Downes, Tony. 1997 (Apr. 28). Data Connection Adds ITU T.120 MCS/GCC Support to DC-Share for UNIX, Enhancing Application Sharing Interoperability with Microsoft NetMeeting 2.0,. <http://www.datcon.co.uk/press/dcunix2.htm>. last accessed: Sep. 19, 1998.

Dunwoody, J. Craig, & Linton, Mark A. 1988 (Mar.). A Dynamic Profile of Window System Usage. *Pages 90–99 of: Proceedings of the 2nd IEEE Conference on Computer Workstations*.

Dutta-Roy, Amitava. 1998. Virtual Meetings with Desktop Conferencing. *IEEE Spectrum*, **35**(7), 47–56.

Edwards, W. Keith. 1994. Session Management for Collaborative Applications. *Pages 323–330 of: Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*. Technologies for Sharing II.

Edwards, W. Keith. 1996. Policies and Roles in Collaborative Applications. *Pages 11–20 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Ellis, Clarence A., & Gibbs, S. J. 1989. Concurrency Control in Group Systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **18**(2), 399–407.

Ellis, Clarence A., Gibbs, S. J., & Rein, G. L. 1991. Groupware: Some Issues and Experiences. *Communications of the ACM*, **34**(1), 38–58.

Engelbart, Douglas C. 1963. A Conceptual Framework for the Augmentation of Man's Intellect. *Pages 1–29 of:* Howerton, P. (ed), *Vistas in Information Handling*, vol. 1. Washington, DC: Spartan Books. Reprinted in Greif (1988).

Engelbart, Douglas C. 1975. NLS Teleconferencing Features: The Journal, and Shared-Screen Telephoning. *In: Proceedings of the Fall COMPCON*.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1996. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading: Addison Wesley.

Garfinkel, D., Welti, B.C., & Yip, T.W. 1994. HP SharedX: A Tool for Real-Time Collaboration. *HP Journal*, Apr., 23–36.

Gates, Bill, Myhrvold, Nathan, & Rinearson, Peter. 1995. *The Road Ahead*. New York: Viking.

Gaver, William W. 1993. Sound Support for Collaboration. *Pages 355–362 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

Gettys, Jim, Karlton, Philip L., & McGregor, Scott. 1990. The X Window System, Version 11. *Software–Practice and Experience*, **20**(S2), S2/35–S2/67.

Gosling, James, Joy, Bill, & Steele, Guy. 1997. *The Java Language Specification*. The Java Series. Addison-Wesley.

Graham, T. C. Nicholas, Urnes, Tore, & Nejabi, Roy. 1996 (Nov.). Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. *Pages 1–10 of: Proceedings 9th Annual ACM Symposium on User-Interface Software and Technology*. ACM SIGCHI.

Greenberg, Saul, & Marwood, David. 1994. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. *Pages 207–217 of: Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*. Technologies for Sharing I.

Greenberg, Saul, & Roseman, Mark. 1996. GroupWeb: A WWW Browser as Real Time Groupware. *Pages 271–272 of: Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*. SHORT PAPERS: Working Together Near and Far, vol. 2.

Greenberg, Saul, & Roseman, Mark. 1998. Groupware Toolkits for Synchronous Work. *In:* Beaudouin-Lafon, Michel (ed), *Computer-Supported Cooperative Work, Trends in Software Series*. John Wiley & Sons. (in press).

Greenberg, Saul, Gutwin, Carl, & Roseman, Mark. 1996 (Nov.). Semantic Telepointers for Groupware. *In: Proceedings of OzCHI '96 Sixth Australian Conference on Computer-Human Interaction*.

Greif, Irene. 1988. *Computer-supported Cooperative Work: A Book of Readings*. San Mateo, California: Morgan Kaufmann Publishers Inc.

Grudin, Jonathan. 1988. Why CSCW Applications Fail. *Pages 85–93 of: Proceedings of the Conference on CSCW*.

Grudin, Jonathan. 1994. Computer-Supported Cooperative Work: History and Focus. *Computer*, **27**(5), 19–26.

Gupta, Aloke, & Hwu, W. W. 1993. An Execution Profiler for Window-oriented Applications. *Software — Practice and Experience*, **23**(5), 487–510.

Gupta, Amar, & Toong, H. D. (eds). 1985. *Insights into Personal Computers*. New York: IEEE Press.

Gutwin, C., Greenberg, S., & Roseman, M. 1996a. Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation. *Pages 281–98 of: People and Computers XI (Proceedings of the HCI'96)*. Springer-Verlag.

Gutwin, Carl. 1997. *Workspace Awareness in Real-Time Distributed Groupware.* Ph.D. thesis, Department of Computer Science, The University of Calgary, Alberta, Canada.

Gutwin, Carl, & Greenberg, Saul. 1998. Effects of Awareness Support on Groupware Usability. *In: Proceedings of the 1998 ACM SIGCHI Conference on Human Factors in Computer Systems.* New York: ACM Press.

Gutwin, Carl, Roseman, Mark, & Greenberg, Saul. 1996b. A Usability Study of Awareness Widgets in a Shared Workspace Groupware System. *Pages 258–267 of: Proceedings of the ACM 1996 Conference on Computer Supported Work (CSCW'96).* New York: ACM Press.

Hamilton, Graham. 1997 (July 24). JavaBeans™1.01. *JavaSoft.* Application Programming Interface Specification. Sun Microsystems, Inc., Mountain View, CA. <http://splash.javasoft.-com/beans/docs/beans.101.pdf>. last access: Oct. 22, 1998.

Hammontree, Monty L., Hendrickson, Jeffrey J., & Hensley, Billy W. 1992. Integrated Data Capture and Analysis Tools for Research and Testing on Graphical User Interfaces. *Pages 431–432 of: Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems.* Demonstration: Analysis Tools/Multimedia Help.

Harrison, Steve, & Minneman, Scott. 1993. The Media Space: A Research Project into the Use of Video as a Design Medium. *Pages 785–791 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration.* Morgan-Kaufmann Publishers.

Hill, Ralph D., Brinck, Tom, Rohall, Steven L., Patterson, John F., & Wilner, Wayne. 1994. The Rendezvous Architecture and Language for Constructing Multiuser Applications. *ACM Transactions on Computer-Human Interaction*, **1**(2), 81–125.

Hix, Deborah, & Hartson, H. Rex. 1993. *Developing User Interfaces: Ensuring Usability Through Product and Process.* New York, New York: John Wiley & Sons, Inc.

Isenhour, Philip L. 1998 (Feb.). *Sieve: A Java-Based Framework for Collaborative Component Composition.* Master of Science thesis, Virginia Polytechnic Inst. and State University, Department of Computer Science.

Isenhour, Philip L., Begole, James "Bo", Heagy, Winfield S., & Shaffer, Clifford A. 1997 (Oct. 22–24). Sieve: A Java-Based Collaborative Visualization Environment. *Pages 13–16 of: IEEE Visualization '97 Late Breaking Hot Topics Proceeding.*

ISO. 1989 (Dec.). *Information Processing Systems - Open Systems Interconnection - The Directory - Authentication Framework.* Tech. rept. International Organization for Standardization and International Electrotechnical Committee. International Standard 9594-8.

Java Foundation Classes. 1998. Java Foundation Classes. *JavaSoft.* <http://java.sun.com/products/jfc>. last accessed: Mar. 23, 1998.

Jones, Oliver. 1993. Multidisplay Software in X: A Survey of Architectures. *The X Resource*, **6**(1), 97–113.

Kaplan, Karen. 1998. Compaq, IBM Lead PC Sales in 1997. *Los Angeles Times*, Jan. 26,, D–2.

Karsenty, A., & Beaudouin-Lafon, M. 1993. An Algorithm for Distributed Groupware Applications. *Pages 195–202 of:* Werner, Robert (ed), *Proceedings of the 13th International Conference on Distributed Computing Systems.* Pittsburgh, PA: IEEE Computer Society Press.

Kelley, C.L., Bosman, E.A., Charness, N.H., & Mottram, M.A. 1998. Reliability Assessment of a User Proficiency Measurement Technique. *International Journal of Human-Computer Interaction*, **10**(1), 33–49.

Kelly, Kevin, & Reiss, Spencer. 1998. One Huge Computer. *Wired*, **6**(8).

Lauwers, J. C. 1990. *Collaboration Transparency in Desktop Teleconferencing Environments.* Ph.D. thesis, Computer Systems Laboratory, Stanford, CA.

Lauwers, J. Chris, & Lantz, Keith A. 1990. Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems. *Pages 303–311 of: Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems.* CSCW - Computer Support for Real Time Collaborative Work.

Lauwers, J. Chris, Joseph, Thomas A., Lantz, Keith A., & Romanow, Allyn L. 1990. Replicated Architectures for Shared Window Systems: A Critique. *Pages 249–260 of: OIS90.* Computer Mediated Work Environments. ACM Press.

Lee, Jang Ho, Prakash, Atul, Jaeger, Trent, & Wu, Gwobaw. 1996. Supporting Multi-User, Multi-Applet Workspaces in CBE. *Pages 344–353 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Lentner, Marvin, & Bishop, Thomas. 1993. *Experimental Design and Analysis*. second edn. Blacksburg, Virginia, USA: Valley Book Company.

Liao, Tie. 1998 (Sept. 2,). Light-weight Reliable Multicast Protocol, *INRIA (French National Institute for Research in Computer Science and Control)*. <http://monet.inria.fr/lrmp/lrmp.html>. last accessed: Sep. 15, 1998.

Loney, Matt. 1998 (June 5). Timbuktu talks to NetMeeting. <http://www.zdnet.co.uk/news/1998/-22/ns-4618.html>. last accessed: Sep. 19, 1998.

Mantei, Marilyn M. 1993. Observation of Executives Using a Computer Supported Meeting Environment. *Pages 695–708 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

Mantei, Marilyn M., Baecker, Ronald M., Sellen, Abigail J., Buxton, William A. S., Milligan, Thomas, & Wellman, Barry. 1993. Experiences in the Use of a Media Space. *Pages 803–808 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

McGarvey, Joe. 1998. Residential PC Penetration Nears 50 Percent. *Inter@ctive Week*, Mar. 10,.

McKinlay, Andy, Procter, Rob, Masting, Oliver, Woodburn, Robin, & Arnott, John. 1994. Studies of Turn-Taking in Computer-Mediated Communication. *Interacting with Computers*, **6**(2), 151–171.

Microsoft Corp. 1998. Microsoft NetMeeting. *Microsoft Corp.* <http://www.microsoft.com/-netmeeting/>. last accessed: Oct. 22, 1998.

Miller, Stewart S. 1997. *IPv6: The Next Generation Internet Protocol*. Butterworth Heinemann.

Minenko, W. 1998 (Mar.). *The Application Sharing Technology,* The X Advisor*, 1(1), Discovery Publishing Group, June 1995, republished in Motif Developer*. <http://www.motifzone.com/-tmd/articles/XpleXer/XpleXer.html>. last accessed: Oct. 22, 1998.

Minenko, Wladimir. 1996. *Advanced Design of Efficient Application Sharing Systems under X Window*. Ph.D. thesis, Universität Ulm,.

Mintzberg, Henry. 1993. A Typology of Organizational Structure. *Pages 177–180 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

Mitchell, Alex. 1996. *Communication and Shared Understanding in Collaborative Writing*. M.Phil. thesis, Department of Computer Science, University of Toronto.

Mitchell, Alex, Posner, Ilona, & Baecker, Ronald. 1995. Learning to Write Together Using Groupware. *Pages 288–295 of: Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*. Papers: Technology at Work, vol. 1.

MLA. 1998 (July 15,). Citing Sources from the World Wide Web*, Modern Language Association of America*. <http://www.mla.org/main_stl.htm#sources>. last accessed: Oct. 22, 1998.

Mowbray, T. J., & Zahavi, R. 1995. *The Essential CORBA: Systems Integration using Distributed Objects*. Canada: Wiley.

Nichols, David A., Curtis, Pavel, Dixon, Michael, & Lamping, John. 1995. High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. *Pages 111–120 of: Proceedings of the ACM Symposium on User Interface Software and Technology*. Distributed User Interfaces.

NIST. 1998. Java Collaborative Environment, *National Institute of Standards and Technology*. <http://snad.ncsl.nist.gov/madvtg/Java/Java.html>. last accessed: Sep. 15, 1998.

Nua Ltd. 1998 (July). *How Many Online?* <http://www.nua.ie/surveys/how_many_online/>. "... here is an 'educated guess' as to how many are online worldwide as of July 1998. And the number is 130 million." last accessed: Oct. 22, 1998.

Obraczka, Katia. 1998. Multicast Transport Protocols: a Survey and Taxonomy. *IEEE Communications Magazine*, **36**(1), 94–102.

Patterson, J.F., Day, M., & Kucan, J. 1996 (Nov.). Notification Servers for Synchronous Groupware. *Pages 122–129 of: Proceedings of CSCW '96*.

Patterson, John F. 1991. Comparing the Programming Demands of Single-User and Multi-User Applications. *Pages 87–94 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST'91)*. CSCW.

PictureTel Corporation. 1997 (Apr. 28). PictureTel announces interoperability with Microsoft NetMeeting 2.0 and enhancements to provide users of NetMeeting with hardware-accelerated videoconferencing. *PictureTel Corp.* <http://www.picturetel.com/press34.htm>. last accessed: Sep. 19, 1998.

Pinsonneault, Alain, & Kraemer, Kenneth L. 1993. The Impact of Technological Support on Groups: An Assessment of the Empirical Research. *Pages 754–773 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

Posner, I.R., & Baecker, R.M. 1993. How People Write Together. *Pages 239–250 of:* Baecker, R.M. (ed), *Readings in Groupware and Computer-supported Cooperative Work*. Morgan Kaufmann.

Prakash, Atul, & Shim, Hyong Sop. 1994. DistView: Support for Building Efficient Collaborative Applications using Replicated Active Objects. *Pages 153–164 of: Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*. Collaborative Editing and Reviewing.

Proshare, Intel. 1998. Intel Proshare. *Intel Corporation*. <http://www.intel.com/proshare/>. last accessed: Mar. 23, 1998.

Read, William, Croft, Molly, Rossignac, Jarek, Kehoe, Colleen, & Pitkow, Jim. 1998 (June). *Graphic, Visualization, & Usability Center's 9th WWW User Survey*. <http://www.gvu.gatech.edu/user_surveys/survey-1998-04/>. last accessed: Oct. 22, 1998.

Reinhard, Walter, Schweitzer, Jean, Volksen, Gerd, & Weber, Michael. 1994. CSCW Tools: Concepts and Architectures. *Computer*, **27**(5), 28–36.

Reuters. 1998 (Apr.). *More Net deals for Saudi prince*. <http://www.news.com/News/Item/-0,4,21624,00.html>. News item 0,4,21624,00, "There are currently no Saudi-based Internet access providers." last accessed: Oct. 22, 1998.

Roseman, Mark, & Greenberg, Saul. 1996a. Building Real Time Groupware with GroupKit, A Groupware Toolkit. *ACM Transactions on Computer Human Interaction (TOCHI)*, **3**(1), 66–106.

Roseman, Mark, & Greenberg, Saul. 1996b. TeamRooms: Network Places for Collaboration. *Pages 325–333 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Ruff, Andreas. 1998 (June 8). personal email communication.

Schuckmann, Christian, Kirchner, Lutz, Schümmer, Jan, & Haake, Jörg M. 1996. Designing Object-Oriented Synchronous Groupware With COAST. *Pages 30–38 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Scrivener, Stephen A. R., Urquijo, Silvia Ponguta, & Palmen, Hilary K. 1996. The Use of Breakdown Analysis in Synchronous CSCW System Design. *In:* Thomas, Peter (ed), *CSCW Requirements and Evalution*. London: Springer Verlag.

Seifert, Rich. 1998. *Gigabit Ethernet; Technology and Applications for High-Speed LANs*. Peachpit Press.

Short, John, Williams, Ederyn, & Christie, Bruce. 1993. Visual Communication and Social Interaction. *Pages 153–164 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

Singhal, Mukesh, Shivaratri, Niranjan G., Singhai, Mukesh, & Shivaratro, Niranjan. 1994. *Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems*. NY: McGraw-Hill.

Smart, Julian. 1998 (Sept. 25,). wxWindows. *Anthemion Software*. <http://web.ukonline.co.uk/-julian.smart/% -wxwin/>. last accessed: Nov. 12, 1998.

Smith, Gareth. 1996a. Cooperative Virtual Environments: Lessons from 2D Multi User Interfaces. *Pages 390–398 of: Proceedings of the ACM 1996 Conference on Computer Supported Work*. New York: ACM Press.

Smith, R. B., O'Shea, T., O'Malley, C., Scanlon, E., & Taylor, J. 1989. Preliminary Experiments with a Distributed, Multi-Media, Problem Solving Environment. *Pages 19–34 of: Proceedings of the First European Conference on Computer Supported Cooperative Work (EC-CSCW '89).* Human Sciences and Empirical Methods. Gatwick, London, UK: Computer Sciences House, Slough, UK.

Smith, Randall B. 1992. What You See Is What I Think You See. *SIGCUE Outlook*, **21**(3), 18–23. Presented at The Spring '92 ACM Conference on Computer-Supported Collaborative Learning.

Smith, Randall B. 1996b (Apr. 24,). *Kansas: A Large, Flat, Mutli-user Virtual World for Interactive Simulations.* Presentation to the Virginia Tech Computer Science Colloquium Series.

Smith, Randall B., Wolczko, Mario, & Ungar, David. 1997. From Kansas to Oz: Collaborative Debugging When a Shared World Breaks. *Communications of the ACM*, **40**(4), 72–78.

Snyder, Seth. 1996 (Mar. 24). The Virtual Meeting for Macintosh. <http://www.rtz.com/-tvmfm.html>. last accessed: Sep. 19, 1998.

Stefik, M., Bobrow, D. G., Foster, G., Lanning, S., & Tatar, D. 1987. WYSIWIS Revised: Early Experiences with Multiuser Interfaces. *ACM Transactions on Office Information Systems*, **5**(2), 147–167.

Streitz, Norbert A., Geissler, Jorg, Haake, Jorg M., & Hol, Jeroen. 1994. DOLPHIN: Integrated Meeting Support Across Local and Remote Desktop Environments and LiveBoards. *Pages 345–358 of: Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work.* Supporting Meetings.

Sun, Chengsheng, Jia, Xiaohua, Zhang, Yanchun, Yang, Yun, & Chen, David. 1998. Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems. *Transactions on Computer-Human Interactions*, **5**(1), 63–108.

Sun, Chengzheng, & Ellis, Clarence (Skip). 1998. Operational Transformation in Real-Time Group Editors: Issues Algorithms, and Achievements. *Pages 59–68 of: Proceedings of the ACM conference on Computer-Supported Cooperative Work (CSCW'98).* New York: ACM Press.

Sun Microsystems, Inc. 1998. *ShowMe SharedApp.* <http://www.sun.com/products-n-solutions/-sw/-ShowMe/products/ShowMe_SharedApp.html>. last accessed: Mar. 23, 1998.

Tanenbaum, Andrew S. 1996. *Computer Networks*. Prentice-Hall International, Inc.

Tang, John C. 1991. Findings from Observational Studies of Collaborative Work. *International Journal of Man-Machine Studies*, **34**(2), 143–160. Reprinted in Baecker (1993).

Thibodeau, Patrick. 1998. Online Population Esitmates Differ. *COMPUTERWORLD*, June 29,. "According to one estimate [by Matrix Information and Directory Services, Inc.] released today, the number of Internet users worldwide reached 102 million by January of this year – almost double the number of users, at 57 million, for the same period last year. ".

Timbuktu Pro. 1998. *Timbuktu Pro Remote Control Software, Netopia, Inc.* <http://www.netopia.-com/software/tb2/>. last accessed: Oct. 22, 1998.

Turner, Nick. 1998. Will Windows 98 Debut With Bang Or Whimper? *Investor's Business Daily*, May 20,, A8.

Viller, Stephen. 1993. The Group Facilitator: A CSCW Perspective. *Pages 145–152 of:* Baecker, Ronald M. (ed), *Readings in Groupware and Computer Supported Cooperative Work: Assisting Human-Human Collaboration*. Morgan-Kaufmann Publishers.

Vtorov, Serge G. 1998 (Feb. 26). Interference*, Mississippi State University*. <http://webphysics.-ph.msstate.edu/javamirror/interf/interference.html>. last accessed: Aug. 25, 1998.

Wahl, M., Howes, T., & Kille, S. 1997 (Dec.). *RFC 2251: Lightweight Directory Access Protocol (v3)*. Status: PROPOSED STANDARD.

Winograd, T, & Flores, F. 1986. *Understanding Computers and Cognition: A New Foundation for Design*. Norwood, New Jersey: Ablex.

Wolf, K. H., Froitzheim, K., & Schulthess, P. 1996. Multimedia Application Sharing in a Heterogeneous Environment. *Pages 57–64 of: The Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)*. New York: ACM Press.

Wollrath, Ann, Riggs, Roger, & Waldo, Jim. 1996. A Distributed Object Model for the Java System. *Computing Systems*, **9**(4), 265–290.

X/TeleScreen. 1998. X/TeleScreen. *VisualTek, Inc.* <http://www.visualtek.com/xtelescreen/>. last accessed: Oct. 22, 1998.

Sources on the WWW are cited according to the Modern Language Association guidelines (MLA, 1998).

Appendix A

Evaluation Data

The following data were obtained during the evaluation of Flexible JAMM versus NetMeeting. See Section 6.1 for interpretation of these data.

A.1 Performance Measures

Table A.1 lists the time in seconds for each group to complete each task. The times for each system (FJ = Flexible JAMM, NM = NetMeeting) are in adjacent columns to facilitate direct comparison. Table A.2 lists the number of errors made by each group for each task.

Table A.1: Seconds to complete task by system.

Group Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	162	280	247	202
2	241	344	260	368
3	199	303	227	260
4	257	406	255	286
5	242	427	279	255
6	243	407	215	242
7	285	366	251	218
8	161	295	246	297
Mean	223.75	353.50	247.50	266.00
Std. Dev.	45.07	56.87	19.63	51.96

Table A.2: Number of errors in task by system.

Group Number	**Text Entry** FJ	NM	**CopyEdit** FJ	NM
1	0	1	0	0
2	6	5	0	0
3	4	1	1	0
4	1	0	0	0
5	3	2	1	1
6	4	0	1	2
7	0	0	1	1
8	1	0	0	0
Mean	2.25	1.25	0.50	0.50
Std. Dev.	2.31	1.67	0.53	0.76

A.2 Perception Measures

Tables A.3–A.9 list each participant's response to each of seven post-task statements. After each task, participants were given the following instructions followed by seven statements.

> Please rate the collaboration software you used to complete the previous task. Please read each statement carefully and circle one choice for the degree to which you disagree or agree with the statement. The scale items are equally spaced. The scale is balanced, having an equal number of items for disagree and agree. The ends of the scale are typically extreme and therefore less frequently used. However, if you feel strongly about a question, please indicate so on the scale.

Participants marked their responses on the following scale. For analysis, the scale was scored ranging from 1 for *Strongly Disagree* to 7 for *Strongly Agree*.

```
   |-------------|-------------|-------------|-------------|-------------|-------------|
Strongly     Moderately      Mildly       Neither        Mildly      Moderately     Strongly
Disagree      Disagree      Disagree     Disagree        Agree         Agree         Agree
                                         Nor Agree
```

Table A.3: Responses to post-task statement 1: *Overall, I am satisfied with the collaboration software.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	6	2	6	1
2	6	4	6	5
3	7	7	7	6
4	7	6	7	7
5	6	5	6	6
6	6	6	6	3
7	7	5	5	6
8	6	2	5	6
9	7	1	7	2
10	6	1	6	5
11	5	3	5	6
12	5	5	6	6
13	5	2	3	2
14	7	6	6	6
15	7	5	6	5
16	6	6	6	4
Mean	6.19	4.31	5.81	4.56
Std. Dev.	0.75	2.02	0.98	1.82

Table A.4: Responses to post-task statement 2: *My partner and I could effectively work at the same time.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	5	1	7	3
2	6	2	6	5
3	7	5	7	6
4	7	5	7	7
5	6	5	6	1
6	6	1	6	3
7	7	1	5	4
8	7	1	4	6
9	7	1	7	1
10	7	1	7	1
11	6	6	6	3
12	6	2	6	5
13	6	3	4	3
14	7	3	5	7
15	6	5	6	5
16	7	6	7	6
Mean	6.44	3.00	6.00	4.12
Std. Dev.	0.63	2.00	1.03	2.06

Table A.5: Responses to post-task statement 3: *It was easy to gain control of the application to enter text.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	6	2	6	2
2	6	4	4	4
3	7	6	4	7
4	7	7	7	7
5	6	6	6	3
6	6	6	6	2
7	6	6	6	3
8	7	6	7	5
9	4	6	4	6
10	7	2	7	2
11	6	6	5	3
12	3	5	6	6
13	6	5	6	5
14	7	6	7	6
15	6	5	5	3
16	7	6	7	6
Mean	6.06	5.31	5.81	4.37
Std. Dev.	1.12	1.40	1.11	1.82

Table A.6: Responses to post-task statement 4: *I could effectively point out specific locations in the text to my partner.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	4	2	4	4
2	5	5	6	6
3	7	6	7	7
4	4	4	6	7
5	6	4	6	6
6	4	5	5	6
7	4	5	5	6
8	5	4	5	6
9	4	5	6	3
10	6	2	6	3
11	4	4	6	5
12	6	4	6	6
13	4	5	3	6
14	6	6	2	7
15	4	4	5	6
16	6	4	5	6
Mean	4.94	4.31	5.19	5.62
Std. Dev.	1.06	1.14	1.28	1.26

Table A.7: Responses to post-task statement 5: *It was easy for us to both simultaneously make changes to the text.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	7	1	4	1
2	6	2	4	4
3	7	4	4	4
4	7	4	7	7
5	6	1	6	1
6	6	1	4	4
7	7	1	7	4
8	7	2	4	2
9	7	1	6	1
10	7	1	6	1
11	6	1	6	3
12	2	4	6	1
13	6	1	4	4
14	7	6	7	4
15	7	4	6	1
16	7	5	6	6
Mean	6.37	2.44	5.44	3.00
Std. Dev.	1.26	1.93	1.21	1.75

Table A.8: Responses to post-task statement 6: *We could both view different portions of the text simultaneously.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	7	4	7	1
2	4	2	6	2
3	7	4	7	4
4	4	4	7	4
5	6	4	6	1
6	6	6	7	1
7	6	4	7	4
8	6	2	6	4
9	7	1	7	1
10	7	1	7	1
11	7	4	7	2
12	7	2	6	6
13	6	4	7	1
14	7	3	7	7
15	7	3	7	1
16	7	1	7	4
Mean	6.31	3.06	6.75	2.75
Std. Dev.	1.01	1.44	0.45	1.98

Table A.9: Responses to post-task statement 7: *It was easy to know what portion of the text my partner could see.*

Participant Number	**Text Entry**		**CopyEdit**	
	FJ	NM	FJ	NM
1	4	4	7	6
2	5	4	6	6
3	4	4	7	7
4	4	4	6	6
5	6	1	5	6
6	6	1	6	6
7	6	4	6	5
8	5	2	5	3
9	7	7	6	7
10	7	7	6	7
11	5	4	7	3
12	7	2	6	2
13	5	4	3	7
14	6	6	6	7
15	7	4	6	6
16	7	6	6	3
Mean	5.69	4.00	5.87	5.44
Std. Dev.	1.14	1.86	0.96	1.71

VITA

James Michael Allen Begole, "Bo"
Date of Birth: June 26, 1963
Place of Birth: McPherson, Kansas

Education

Doctor of Philosophy, Computer Science, Virginia Tech, Blacksburg, VA, December 1998.
Dissertation: *Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems*

Master of Science, Computer Science, Virginia Tech, Blacksburg, VA, July 1994.
Thesis: *The Effectiveness of Online Interactive Tutorial versus Online Help and Printed Manual in Project GeoSim's User Assistance System*

Bachelor of Science, *summa cum laude*, Virginia Commonwealth University, Richmond, VA, May 1992.

Research Interests

computer-aided education and training, computer-supported cooperative work, concurrency, distributed systems, human-computer interaction, interactive visualization, networks, object-oriented design and development, user-interface technologies.

Experience

Instructor, Dept. of Computer Science, Virginia Tech, Aug. – Dec. 1998.

Graduate Research Trainee, Dept. of Computer Science, Virginia Tech, Aug. 1997 – Aug. 1998.

Research Intern, SunLabs/JavaSoft, Cupertino, CA, Aug. – Dec. 1996, May – Aug. 1997.

Graduate Research Assistant, Dept. of Computer Science, Virginia Tech, Jan. – Aug. 1996, Jan. – May 1997.

Senior Software Engineer, Research Development Corp., Herndon, VA, June 1995 – Jan. 1996.

Software Analyst, E-Systems, Melpar Division, Ashburn, VA, June 1994 – June 1995.

Graduate Research Assistant, Dept. of Computer Science, Virginia Tech, Aug. 1992 – June 1994.

System Administrator, Pembrooke Health, Richmond, VA, Mar. 1989 - July 1992.

Senior Arabic Translator/Liaison Official, Staff Sergeant, (MOS-98G), US Army, July 1981 – June 1989.

Refereed Publications

Begole, James "Bo," Struble, Craig A., & Shaffer, Clifford A. 1997. Leveraging Java Applets: Toward Collaboration Transparency in Java. *IEEE Internet Computing*, **1**(2), 57–64.

Begole, James "Bo," Struble, Craig A., Shaffer, Clifford A., & Smith, Randall B. 1997. Transparent Sharing of Java Applets: A Replicated Approach. *Pages 55–64 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-97).* New York: ACM Press.

Begole, James "Bo," Rosson, Mary Beth, & Shaffer, Clifford A. 1998. Supporting Worker Independence in Collaboration Transparency. *Pages 133–142 of: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-98).* New York: ACM Press.

Begole, James "Bo," & Shaffer, Clifford A. 1998 (Oct.). *Flexible Collaboration Transparency.* invited paper at *IEEE Workshop on Distributed Visualization Systems.*

Begole, James M.A. 1994 (June). *The Effectiveness of Online Interactive Tutorial versus Online Help and Printed Manual in Project GeoSim's User Assistance System.* Master of Science thesis, Virginia Polytechnic Inst. and State University, Department of Computer Science.

Begole, James M.A. 1998 (Dec.). *Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems.* Ph.D. thesis, Virginia Polytechnic Inst. and State University, Department of Computer Science.

Begole, James M.A., Shaffer, Clifford A., & Lattanzi, Mark. 1993 (Oct.). The Project GeoSim Graphical User Interface. *In: Proceedings of the 23rd Annual Virginia Computer Users Conference.*

Isenhour, Philip L., Begole, James "Bo," Heagy, Winfield S., & Shaffer, Clifford A. 1997a (Oct. 22–24). Sieve: A Java-Based Collaborative Visualization Environment. *Pages 13–16 of: IEEE Visualization '97 Late Breaking Hot Topics Proceeding.*

Shaffer, Clifford A., Begole, James M.A., Carstensen, L. William, Morrill, Robert W., & Fox, Edward A. 1994 (Feb.). GeoSim: A GIS-based Simulation Laboratory for Introductory Geography. *In: Proceedings of the 35th ADCIS Conference.*

Technical Reports

Begole, James "Bo," & Shaffer, Clifford A. 1997 (Feb.). *Internet Based Real-Time Multiuser Simulation: Ppong!* Technical Report TR-97-01. Virginia Polytechnic Inst. and State University, Department of Computer Science.

Begole, James "Bo," Isenhour, Philip L., & Shaffer, Clifford A. 1998 (May). *JavaBeans as a Framework for Collaborative Software.* Technical Report TR-98-13. Virginia Polytechnic Inst. and State University, Department of Computer Science.

Hines, David T., Begole, James M. A., Klipsch, Colin A., & Shaffer, Clifford A. 1994. *The GeoSim Interface Library (GIL): Programmer's Manual, Version1.0.1.* Technical Report TR-94-31. Virginia Polytechnic Inst. and State University, Department of Computer Science.

Isenhour, Philip L., Shaffer, Clifford A., Begole, James "Bo," Nielsen, Jeff, & Abrams, Marc. 1997. *Sluice: A Java-Based Framework for Collaborative Interactive Modular Visualization Environments.* Technical Report TR-97-17. Virginia Polytechnic Inst. and State University, Department of Computer Science.

Honors

1998 Upsilon Pi Epsilon/Microsoft scholarship recipient

Outstanding Undergraduate in Mathematical Sciences, VCU 1992

US Army Achievement, Commendation, and Meritorious Service Medals

Professional Memberships

Association for Computing Machinery

Institute of Electrical and Electronics Engineers Computer Society

Upsilon Pi Epsilon, Honor Society of the Computing Sciences

Phi Kappa Phi Honor Society