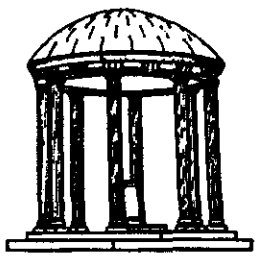
A Model, Architecture, and Operating System Support for Shared Workspace Cooperation

TR89-029

August, 1989

Sheng-Uei Guan



The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

A Model, Architecture, and Operating System Support for Shared Workspace Cooperation

by

Sheng-Uei Guan

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Chapel Hill

1989

Approved by:

Hussein M. Abdel-Wahab

Advisor: Hussein M. Abdel-Wahab

Peter Calingaert

Reader: Peter Calingaert

F. Donelson Smith

Reader: Don Smith

Sheng-Uei Guan. A Model, Architecture, and Operating System Support for Shared Workspace Cooperation (Under the direction of Hussein M. Abdel-Wahab.)

ABSTRACT

As more and more special-purpose *real-time user cooperation* tools are being built, the impact of these new applications to operating systems has just emerged. Instead of directly implementing an operating system for such applications, we try to identify user requirements for such cooperation and the support that designers of such tools seek. With this investigation, the desirable requirements and support can then be achieved through different approaches, e.g. operating system kernels, on-line libraries, or user module libraries. Rather than tackle the problem in its full generality, we focus on real-time cooperation with *shared workspaces*, which is the core of most real-time cooperation.

This dissertation describes a shared workspace model. Subtleties supporting this shared workspace have been studied. A *Remote Shared Workspaces* facility has been built as a research vehicle. Desirable features for shared workspace cooperation are investigated; the relevance of operating systems supporting them is also discussed. An architecture supporting dynamic groups formation and activities is presented. It supports remote cooperation and also cooperation with existing single-user applications. Operating system mechanisms are also described to support multi-user tools development and sharing of user privileges in a session, namely: *multi-user processes* and *shared capabilities-lists*. A system-call level programmers' interface has been specified.

We also introduce *jointly-owned objects*, found in real life and the computer world. The use of multi-user tools makes the existence of jointly-owned objects a necessity: a participant who joins a multi-user tool written by others knows that the user agent executed in his name is not a Trojan horse if the multi-user tool is jointly owned by all the participants. The concept of "jointly-owned" is generalized to "conditionally jointly-owned", which helps resolve conflicts among joint-owners. Graham and Denning's protection model is extended to incorporate these conditionally jointly-owned entities. *Authority-* and *quorum-based objects* are investigated as instances of conditionally jointly-owned objects.

ACKNOWLEDGEMENTS

My deepest appreciation goes first to my advisor, *Hussein M. Abdel-Wahab*, for numerous discussions, comments, and guidance. His insight has kept me pursuing a fruitful path. His practical knowledge with the system helped me solve difficulties encountered building the prototypes. His encouragement made three years' research a valuable experience.

I would also like to express my sincere gratitude to my committee members whose guidance and endurance were essential to the development of this study.

To *Peter Calingaert*, for much valuable advice, constructive comments on numerous versions of my draft, and generous giving of time in proofreading the final version.

To *Jay Nievergelt*, for initiating the *Remote Shared Workspaces* project and providing much helpful advice.

To *Don Smith*, for much valuable comments in reading my dissertation draft and the final version.

To *Dean Brock* and *Jan Prins*, for helping me realize an attainable goal.

I am also grateful to *John Smith* and *Michael Stumm* for making comments on my initial dissertation proposal, to *Richard Snodgrass* and *Keith Lantz* for making comments on my initial dissertation draft, to *Joey Hughey* for making comments on my initial dissertation proposal and assisting in the implementation of the *Remote Shared Workspaces* user agent, and to *Jih-Fang Wang* for implementing the *Remote Shared Workspaces* communications server.

I am also indebted to my parents, Horng-Tzou and Fong-Mei, for their constant encouragement. Finally, this work would not be possible without the support and love from my wife, *Julia Wang Feng*, and the constant joy from my son, *David Shaofang*.

(This work was partially supported by the Office of Naval Research, under contract N00014-86-K-0680, and under IBM Shared University Research Agreement #826.)

TABLE OF CONTENTS

LIST OF FUNCTION CALLS

LIST OF FIGURES

LIST OF EXAMPLES

CHAPTER 1

INTRODUCTION

1.1. Overview

With maturing network technology and readily available personal computers or workstations, real-time collaboration is experimented with frequently nowadays. Collaboration occurs among intra-machine, intra-LAN, or inter-network users. Although the distance varies, the basic user requirements are mostly the same.

Computer supported cooperative work (CSCW) [Greif88] touches a wide range of fields, including computer science, group communication [Goldberg75], office systems, psychology, and organizational design. Tools developed include electronic mail [Hiltz81, Vallee83], electronic bulletin boards [Essick85], computer conferencing [Greif82, Sarin84, Lantz86, Walters87, Hughes88, Sakata88, Abdel-Wahab88], meeting schedulers [Sarin85], group decision support systems [Bui86, Gray87, Kraemer88], brainstorming and group problem solving tools [Stefik86, Stefik87, Malone87, Malone87a], collaborative writing [Seliger85, Smith87, Fish88], hypertext systems [Trigg86, Delisle87, Conklin87, Trigg88], and project management tools [Tichy82, Sathi86, Perry87].

As more and more special-purpose cooperation tools are being built, the impact of these new applications on operating systems has just emerged. This statement is supported by the observation that a few recent operating systems are designed with an explicit goal of supporting cooperative work (see Sec. 1.4.4). Instead of going directly to implement an operating system for such applications, we try to ask ourselves first: how should operating systems be designed to support these CSCW applications? More specifically, how should operating systems be designed to support *real-time (on-line, synchronous)* cooperation?

Before attempting to answer these questions, someone may ask: what is the difference between real-time and *non-real-time* cooperation? Why is it so important to support real-time cooperation? Real-time cooperation differs from non-real-time

cooperation in two aspects. First, users are able to interact and coordinate with each other promptly. For example, there is a significant difference between contacting a person by phone versus by electronic mail. Second, users are able to share objects in a real-time fashion. For example, the process output of an editor can be shared. With these differences, we see that users can cooperate more effectively in a real-time cooperation. To answer the question how operating systems should be designed to support real-time cooperation, our approach is to first find out user requirements for such cooperation and to identify the support that designers of such tools seek. We also look for the infrastructure of cooperation tools. With this research, the desirable cooperation features and support can then be achieved through any one of several different approaches, e.g. operating system kernels, on-line libraries, and user module libraries. This research is rewarding because in many existing systems real-time user cooperation is usually poorly supported and cannot be achieved without substantial implementation effort.

A related and rewarding research topic is to investigate the use of existing single-user tools for real-time cooperation. To name a few examples, an editor can be used for co-writing a paper, a learning-guide can be used for group tutoring, a debugger can be used for co-debugging, etc. Since a large investment has already been made in developing and learning these tools, it would be fruitful for any general support to include this capability.

1.2. The Thesis

THESIS Shared workspace cooperation is the core of most real-time cooperation. Existing operating systems do not provide adequate support for user requirements in such cooperation, nor do they have enough support for builders of such cooperation tools. To support the above requirements either from the server/library or system-call level will serve the needs of users in such cooperation and reduce significantly the implementation effort of cooperation tool builders. Mechanisms can be developed to support these requirements.

GOALS The goals of this research are: to identify the desirable features and support for shared workspace cooperation, to develop architectures and operating system mechanisms to support them, and to extend the related protection model. We also look for mechanisms to support sharing of existing single-user applications.

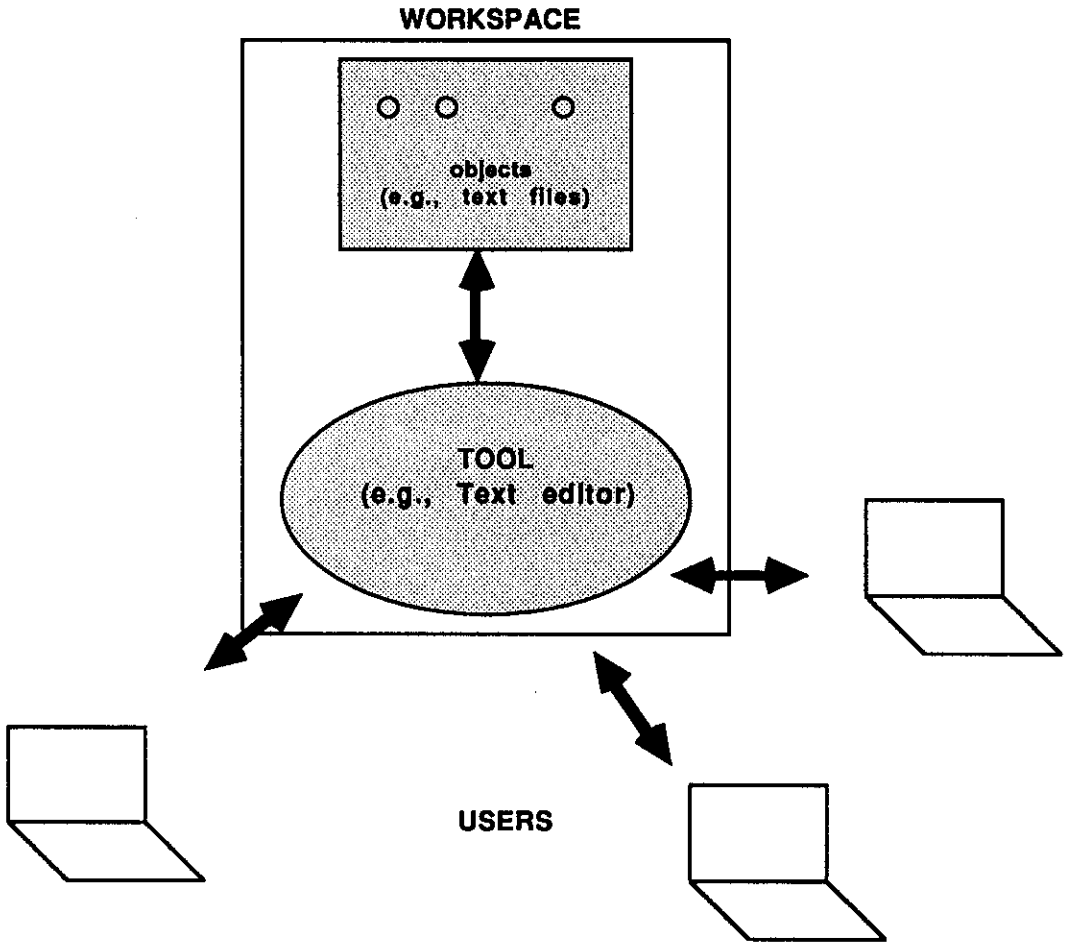


Fig. 1.1 Shared Workspace

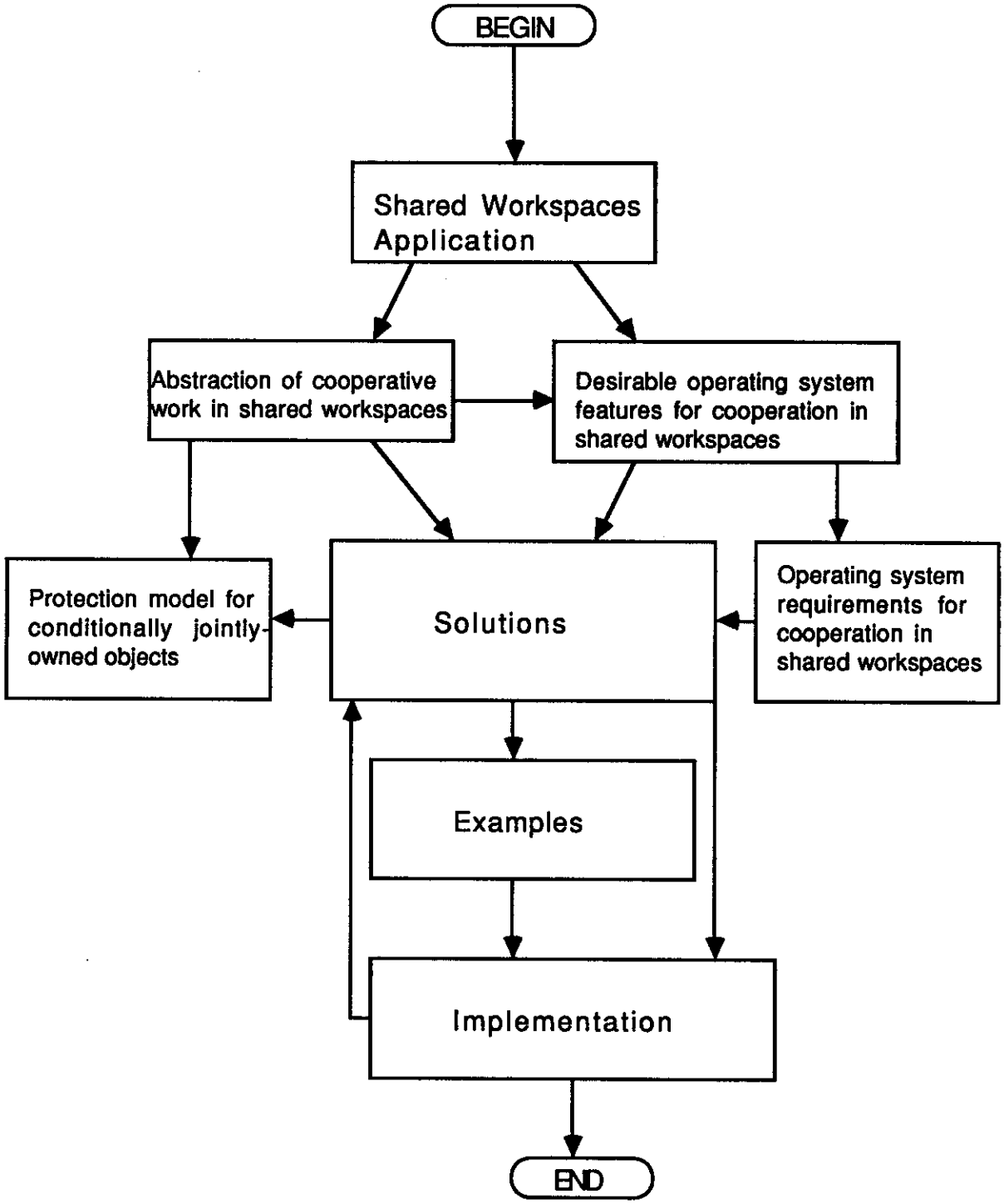


Fig. 1.2 Research Map

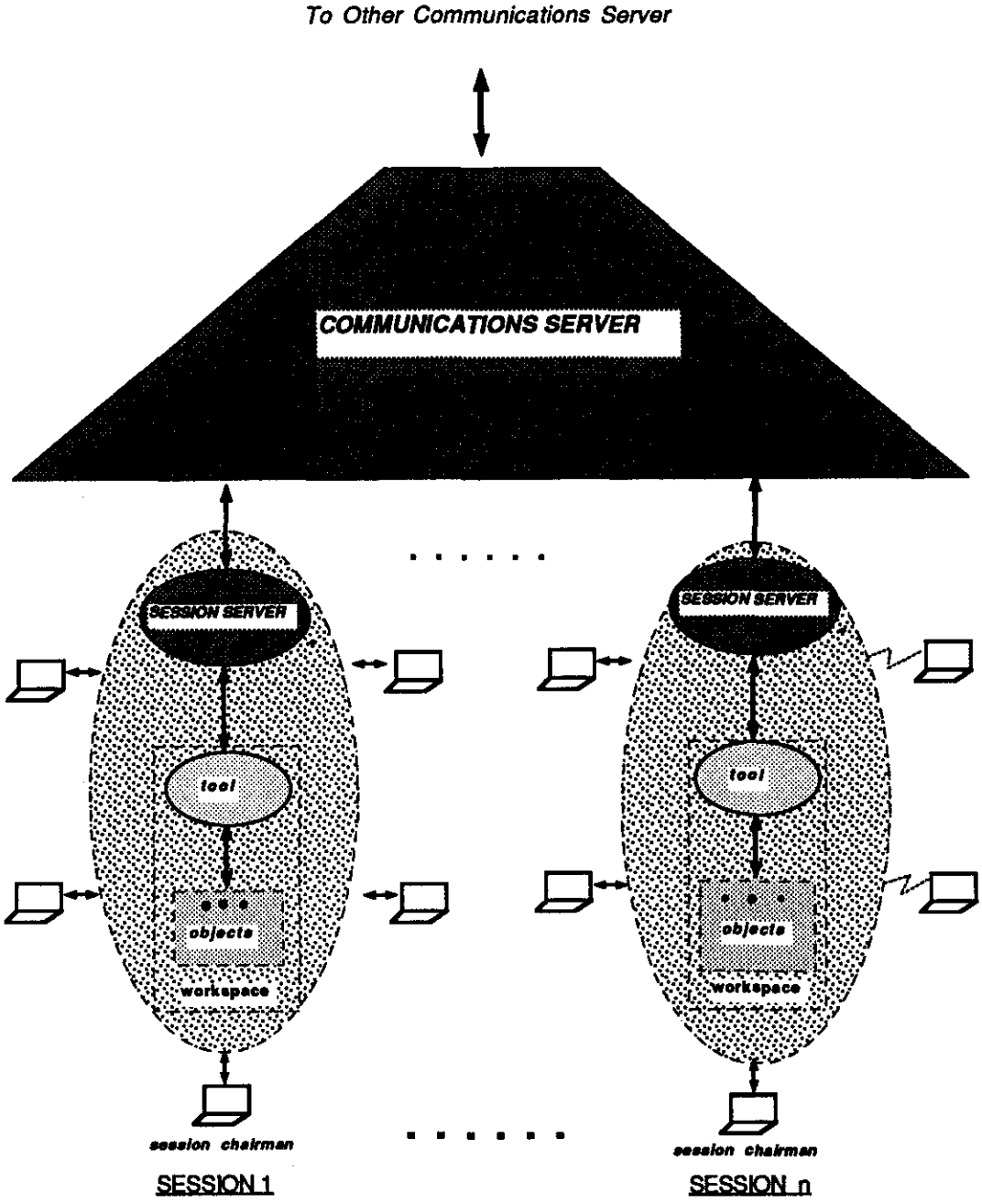


Fig. 1.3 Remote Shared Workspaces

Rather than tackle the problem in its full generality, we will focus on cooperation with *shared workspaces* (*Fig. 1.1*), which is the core of most real-time cooperation, especially in distributed applications (see Sec. 2.4 for examples supporting this claim). Our approach (*Fig. 1.2*) is to build and experiment with a general-purpose *Remote Shared Workspaces* (RSW) facility (*Fig. 1.3*) [Abdel-Wahab88] on top of an existing operating system, and derive an abstract view of shared workspace cooperation.

RSW is an application that achieves low-cost remotely shared workspaces based on widely available systems and single-user application programs. Issues for building such a distributed application are investigated. For example, how is remote real-time cooperation achieved when a user usually has no account on machines other than his own? What are the security problems incurred by using such cooperation tools? What effect is there on the users if a *single shared workspace* or *replicated shared workspaces* are maintained? What floor passing (chalk passing) scheme can be developed for a close cooperation using the existing technology? What are the issues or restrictions in sharing an existing single-user tool?

Using experience from building and experimenting with such shared workspace cooperation tools that operate over a network, desirable features and requirements are then identified either from a user's or a programmer's point of view. Next, we investigate whether to support these features from an application, an on-line library, or the operating-system level. Architectures or operating system mechanisms to support these features can be developed, and a programmers' interface at the level of system or library calls can be specified. Multi-user cooperation examples are designed using these system or library calls to show how the proposed mechanisms provide the desirable cooperative work features. The protection model is also investigated to see whether any generalization is needed to support shared workspace cooperation.

An initial implementation has been done to ascertain the realizability of the proposed mechanisms. The *Remote Shared Workspaces* prototype [Abdel-Wahab88] has been re-implemented with the proposed primitives.

RESEARCH OUTLINE

1. Offer abstract view of shared workspace cooperation and specify the related requirements.
2. Build general-purpose remote shared workspaces: investigate design aspects, architecture and operating system support.
3. Identify the desirable features and requirements for shared workspace cooperation.
4. Investigate the following, and operating system support therefor:
   a. dynamic groups formation and activities;
   b. mechanisms to support cooperation using existing single-user application;
   c. infrastructure of shared workspace cooperation tools;

d. mechanisms for sharing privileges in cooperation;
e. jointly-owned objects.

5. Develop the protection model for jointly-owned objects.

I should point out here that this is still a burgeoning field, so the major purpose of this research is investigation, identification and solutions. No guarantee for complete treatment can be made because the ever-changing user requirements and evolving technology.

1.3. Major Results

Four mechanisms have been developed, namely:

1. dynamic groups and shared viewing;
2. multi-user processes;
3. shared capabilities-lists;
4. conditionally jointly-owned objects.

These mechanisms are independent of each other, except for multi-user processes and conditionally jointly-owned objects. This will be explained shortly.

A general *Remote Shared Workspaces* (RSW) facility has been built as our research vehicle and reported in [Abdel-Wahab88]. A brief description is provided in Appendix A. RSW provides the large community of *UNIX* [Ritchie78] users linked by *Internet* [Postel81, Cerf83] with a general-purpose facility that effectively converts a single-user software tool into one that can be used for real-time collaboration by a group of mutually remote users. The prototype has been used between the departments of computer science at University of North Carolina — Chapel Hill (UNC) and North Carolina State University (about 25 miles apart). It has also been tested between Old Dominion University in Norfolk, Virginia, and UNC-Chapel Hill (about 180 miles apart).

A shared workspace model has also been developed. Subtleties of supporting this shared workspace have been studied. Requirements of shared workspaces have been specified. Activities and modes commonly seen in a real-time cooperation have been studied. An effective and graceful scheme for multi-user floor-passing has been developed. Design aspects of shared workspace cooperation tools have been studied. An architecture supporting dynamic groups formation and activities has also been developed. It can be applied to a distributed system and supports cooperation using existing single-user applications. The architecture has been specified in the details of library calls.

Operating system mechanisms have been investigated to support multi-user tool development and sharing of user privileges in a session, namely: *multi-user processes* and *shared capabilities-lists*. A multi-user process is a process jointly owned by several users. It runs under the union of these owners' privilege domains and provides a multi-user terminal interface to the joint-owners. A shared capabilities-list is a capabilities-list shared by multiple users, where each participant may *post* a capability to his private object and allow others to share access. This permits a dynamic and controlled sharing of workspace objects. The shared capabilities-list mechanism can be applied to a distributed system. A programmers' interface at the system-call level has been specified. Multi-user cooperation examples using these system calls have been designed to show how the proposed mechanisms provide the desirable cooperation features.

Jointly-owned objects are found in real life and the computer world. The use of multi-user tools makes the existence of jointly-owned objects a necessity: if a participant joins a multi-user tool written by others, how can he be sure that the user agent executed in his name is not a Trojan horse [Saltzer75]? This doubt can be removed by making the multi-user tool jointly owned by all the participants so that each one knows the multi-user tool cannot be replaced without his presence.

The concept of "jointly-owned" is generalized to "conditionally jointly-owned". Conditions can be imposed on the presence of joint-owners or users who have the rights to access or make a protection state change on an object. A jointly-owned object is then a conditionally jointly-owned object with a null condition. A mechanism realizing conditionally jointly-owned objects is presented. Conditionally jointly-owned objects can be useful in resolving conflicts among joint-owners or users. The same concept can be applied to subjects (i.e. processes). Graham and Denning's protection model [Graham72] has been extended to incorporate these conditionally jointly-owned entities.

Quorum- and *authority-based objects* are instances that have been investigated. Conditions such as a quorum or a list of users can be imposed on the presence of joint-owners or users who have the rights to access or make a protection state change of an object.

1.4. Related Work

1.4.1. Overview of Computer-Supported Cooperative Work

Electronic Mail and Bulletin Boards

Electronic mail systems are widely used nowadays; their functions are well-developed. They are used primarily for non-real-time communication. At the New Jersey Institute of Technology, Starr R. Hiltz and Murray Turoff designed the *Electronic Information Exchange System* (EIES) [Hiltz81]. This system provides four general-

purpose structures: *1.* messages — delivery to individuals or groups; *2.* conferences — asynchronous meetings, transcripts, and voting; *3.* notebooks — private to an individual or shared among a group of users; *4.* directory — a membership directory containing both individuals and defined groups with self-entered interest descriptions.

At the University of Illinois, a discussion-oriented, computer bulletin-board system called *Notefile* has been developed [Essick85]. It contains a number of notefiles. Each notefile contains many discussions. The applications of *Notefile* include problem report filing, mail processing, project workbooks, and automatic logs.

Computer Conferencing and Meeting Schedulers

At the Xerox Palo Alto Research Center (PARC), an experimental meeting room — *Colab*, provides computer support for collaborative activities in face-to-face meetings. Several prototypes have been built, such as *Boardnoter*, a multi-user interface that expresses many characteristics of a chalkboard in face-to-face meetings [Stefik86, Stefik87].

At Stanford University, an experimental prototype has been implemented to support integrated multimedia conferencing [Lantz86]. It offers a window-based computer conferencing system that permits existing applications to run in the context of a conference. At NEC, a computer-supported desk-to-desk conference system has been developed for users to conduct a meeting from their telephone-attached workstations [Sakata88]. Participants can jointly view and manipulate relevant multimedia information distributed through a local area network, and discuss the shared information over the telephones.

At the MIT Laboratory for Computer Science, several prototypes [Greif82, Sarin84, Sarin85, Seliger85] have been developed for office activities. *RTCAL* [Seliger85], a real-time meeting scheduler, supports scheduling of meetings by building a shared information workspace from participants' on-line calendars. Participants may speak over the phones and use their workstation displays as blackboards.

Group Decision Support

Decision support systems (DSS) are systems that try to improve the performance of information workers in organizations. Group decision support systems are DSS designed to help groups of senior management and professional groups reach consensus [Gray87]. Some commercial systems have already been built: Applied Future Inc.'s *CONSENSOR*, Decisions & Design Inc.'s *Decision Conference*, etc. [Gray87].

Brainstorming and Group Problem Solving

At Xerox PARC, a *Colab* tool *Cognoter* [Stefik87, Foster86] is used to prepare presentations collectively by a group of people. Its output is an annotated outline of

ideas and associated text. *Cognoter* organizes a meeting into three distinct phases — brainstorming, organizing, and evaluation, each of which emphasizes a different set of activities.

At MIT, a prototype called *Information Lens* [Malone87, Malone87a] is designed to include not only good user interfaces for supporting the problem-solving activities of individuals, but also good organizational interfaces for supporting the problem-solving activities of groups.

Collaborative Writing and Hypertext

At MIT, an editing system *CES* has been developed for co-authors working asynchronously on a shared document. Authors can work independently on different sections of the document [Seliger85].

Hypertext systems contain linked texts and figures that can be used for reading documents nonlinearly. At Tektronix Inc., a hypertext system, *Contexts* [Delisle87], has been implemented; it extends the existing hypertext technology to support collaborative writing. At Xerox Parc, two tools — *guided tours* and *tabletops*, implemented in the *NoteCards* [Trigg86] environment, allow authors to employ annotation, graphic layout, and ordered presentation when communicating to readers [Trigg88].

Project Management

The *Callisto* project [Sathi86] was initiated by the Digital Equipment Corporation to study and support the management of large projects. Research goals were established in the following four areas for this project: activity modeling, configuration management, activity scheduling, and project control.

1.4.2. Conceptual Work in Computer-Supported Cooperative Work

Many pioneers have worked in this field [Greif88]. Douglas Engelbart began to experiment with systems using computers to support collaboration in the late 1960's. *NLS* (oN-Line-System) [Engelbart68, Engelbart75], later called *AUGMENT* [Engelbart82, Engelbart84, Engelbart84a], supported useful functions for collaboration, as elaborated in the next section. The following two dissertations represent a comprehensive treatment of the related subject.

Sunil Sarin's dissertation [Sarin84] serves as a guideline for designing and implementing real-time conferences on distributed computer systems. He pointed out that a *real-time conference* allows a group of users, each at his or her own workstation, to conduct a problem-solving meeting by collectively viewing and manipulating a shared space of on-line application information while using a voice communication channel for discussion and negotiation. He proposed useful functions for real-time conferences

and evaluated different implementation techniques. He also provided criteria for choosing alternative techniques when designing a real-time conferencing system.

Gregg Foster claimed in his dissertation [Foster86a] that a real-time computer-based environment can be built from network-connected workstations and that such a system can enhance group work. *Colab* [Foster86a, Stefik86] is such a system built to provide the meeting participants with simultaneous and shared access to the meeting database. *Colab* and its tools explored the following properties of computer-based cooperation: the structure of the problem-solving process, the design of multi-user interfaces, social coordination, simultaneous activity, maintenance of consistent views of shared objects, and uses for digitally captured meetings [Foster86a]. Gregg Foster also investigated real-time software tools to support groups working together in the same room. Strict and relaxed versions of *WYSIWIS (what you see is what I see)* are discussed.

The research of Sunil Sarin and Gregg Foster is based on a broad area of real-time CSCW and touches the general concepts and issues; mine differs from theirs in that a narrower scope — shared workspace cooperation — is investigated; support from the operating system point of view is investigated.

1.4.3. Sharing Existing Single-User Applications for Cooperation

NLS was the first terminal-based conferencing system that allows integration with the existing software environment. Stanford's multimedia conferencing prototype [Lantz86] was the first window system-based conferencing system that achieves similar function. These systems allow existing application programs to be used without modification by users in a conference through use of a communication system. A similar system built at AT&T Bell Laboratories is the *Rapport* multimedia conferencing system [Ensor88]. It uses the *X* window system [Scheifler86] executing on *UNIX* [Ritchie74] as a standard input/output environment. Many programs based on this environment can be run by conferees without modification.

The *Remote Shared Workspaces* prototype [Abdel-Wahab88] offers similar functions. It is a general-purpose utility that achieves remote shared workspaces and converts any appropriate single-user tool into one that can be used for real-time collaboration. An example of an appropriate single-user tool is a text editor or a debugger. No special-purpose resources are required to use it. For example, the required windows can be created using any available window system such as the *Berkeley UNIX 4.3BSD* [Leffler89] *window* program that runs on any ASCII terminal, or the MIT *X* window system that runs on a variety of workstations.

All the prototypes mentioned above are application-level solutions to sharing existing single-user applications for cooperation. Instead of implementing every CSCW

application from scratch, this dissertation research goes a step further: it tries to identify the infrastructure of CSCW applications and investigates the support from operating system or library/server level, with a clear programmers' interface defined.

1.4.4. Operating System Functions for Cooperative Work

Operating systems evolved historically: *serial processing, batch, multiprogramming, timesharing, virtual memory, virtual machine* [Adair66, Creasy81], *networking* [Tanenbaum85], and *user interactive* [Beretta82]. Most computers can be networked today and cooperation among users occurs often. Cooperation tools have been built on top of existing operating systems that have meager real-time cooperation support. Even the user interactive operating systems with multiple windows and user-friendly environments are oriented toward single-user applications or non-real-time cooperation. As a result, real-time cooperation is usually poorly supported and cannot be achieved without substantial implementation effort.

Terminal Linking, Floor Passing, Shared File, Journal

Some operating systems, e.g. *NLS/AUGMENT, TENEX Link*, provide *terminal linking*, i.e. shared screen mode for multiple users [Engelbart68, Engelbart75]. Such systems do not work properly unless all linked terminals are of the same type. Tymshare's *AUGMENT* [Engelbart82, Engelbart84, Engelbart84a] aims toward supporting close collaboration among groups of workers. The support includes "virtual" terminal linking, floor-passing commands, shared file, and journal. Virtual terminal linking allows screen sharing across dissimilar terminal types. The shared file is a hypertext-like shared file system where files can be interlinked to create a shared network of information among collaborators. An authorship-change record is maintained for each statement in the file. The journal system supports a recorded history of dialogs having attributes similar to those provided to professionals, with libraries to store, catalog, and access them.

Group Programming

A distributed file system, *CFS* [Schroeder85] was designed to support group programming. It supports two jobs: to help each programmer manage a private file environment in which to work, and to help the group share consistent versions of the software being developed in parallel. A special feature is that all remote files in *CFS* are immutable and only remote files are shared. A new version is created whenever a remote file is modified.

WYSIWIS Conference

Another distributed system has been built [Suzuki86] to support real-time conferencing. The concept of *group network directory*, which corresponds to a group of nodes (LAN stations), is introduced to support WYSIWIS meetings. Any access via group network directory is broadcast to every node of the group and executed simultaneously.

The research described above represents pioneering efforts by operating system designers to support cooperative work. Cooperation support in these operating systems can be classified into two categories: *1.* linking of objects: object versions, linked file system, and group network directory; *2.* linking of terminals: support for converting a single-user tool for multi-user cooperation. Terminal linking and floor-passing provide a low-level support for collaboration. The converted tool is restrictive: only one user is active at a time, hence multi-user freedom in real-time collaboration cannot be achieved.

Take an editor as an example [Stumm88]. Shared viewing with a single-user editor is like several users sitting at the same terminal using the same editor, except that the network allows remote users. Only one user at a time can be using the editor. In contrast, a multi-user editor allows several users to use it simultaneously. As we can see, an important step is missing in the research described above, namely, *identifying the functional deficiencies of existing operating systems to support real-time cooperation.* In many cases, *ad hoc* approaches have been adopted instead.

1.4.5. Protection Model for Jointly-Owned Objects

Graham and Denning [Graham72] proposed a protection model based on one developed by Lampson [Lampson71] to permit the cooperation of mutually suspicious subsystems. They left the case of "jointly-owned" unsolved [Graham72, Harrison76, Linden76, Landwehr81]. As real-time cooperation becomes more frequent, the possibility of "jointly-owned" can no longer be ignored. Graham and Denning's work is extended for jointly-owned objects. Their model is chosen here for extension because it represents the widely used access matrix model [Landwehr81, Maekawa87]. The problem of conflicts among the joint-owners is solved with *presence conditions.*

1.5. Outline of Dissertation

Chapter 2 discusses the shared workspace cooperation: the model, dynamic groups, sharing, floor passing, design aspects of tools for such cooperation, and desirable features. Requirements for such cooperation are then summarized. Chapter 3

presents an architecture supporting dynamic group formation, activities, and sharing of single-user tool for multi-user collaboration. A library call interface is specified. Examples are provided to show the use of such a mechanism. Possible extensions and issues are discussed.

In Chapter 4, two mechanisms are proposed: multi-user processes and shared capabilities-lists. A system call interface is specified. Examples are provided to show the use of such mechanisms. Design alternatives and issues are also discussed. Chapter 5 summarizes Graham and Denning's protection model, discusses jointly-owned objects and subjects, presents the extended protection model, and introduces the design of two instances of conditionally jointly-owned objects: quorum- and authority-based objects. Examples are also provided.

Chapter 6 describes the prototype indirect implementation for each mechanism proposed, sketches a direct implementation, and presents implementation issues and conclusions. Chapter 7 summarizes the research results and related work; future directions and conclusions are then presented. Appendix A describes the *Remote Shared Workspaces* application. Formation of a cluster and a session are described. Issues for single and replicated workspaces are also discussed. Appendix B gives a listing of a *C* program for the *Remote Shared Workspaces* prototype rebuilt using the proposed dynamic group library interface. It has been implemented and tested under *UNIX*.

CHAPTER 2

SHARED WORKSPACE COOPERATION

In everyday life people meet for certain purposes: to review documents, to solve problems, to develop programs, etc. For office work, the data on managerial communications indicate that top managers spend most of their time in meetings [Panko64, Mintzberg79]. It would be beneficial if computers could be used to support physical meetings. Through this support, the users could be more efficient in meeting activities, information exchange, and retrieval. In the following we give a model based on which meeting support applications can be built. We begin with some definitions.

2.1. Definitions

Object: a protected entity in a computer system. Examples are data arrays sitting in main memory and files containing texts, graphs, or images.

Subject: an identifiable active entity in a computer system to which authorizations are granted, and whose access to objects must be controlled. Examples are a process and a person with a computer account.

User: a person as a subject, who is usually bound to his login identifier.

Privilege: the set of rights owned by a subject.

Domain: the set of objects that currently may be accessed by a subject.

Owner: a subject who has full rights to an object. He is usually the one who creates the object.

Objects can be jointly owned. There are two possibilities: the ownership is shared equally among the users, or the ownership is delegated by others to a user.

Tool: an interactive program whose execution provides user interfaces (standard input/output/error channels) to the user(s) to manipulate objects.

Single-User Tool: a tool whose execution provides interfaces to one user.

Multi-User Tool: a tool whose execution provides interfaces to more than one user.

Cooperation (collaboration): an activity in which people work jointly to achieve a goal.

Session: a cooperation that has several users meeting together at the same time.

A user joining a cooperation is called a *participant* and the union of participants is called a *group*. A cooperation can be *real-time* (on-line) or *non-real-time* (off-line). In a real-time cooperation (also called a session), participants cooperate simultaneously. A session usually has a *chairman*. Participants may cooperate either face-to-face or remotely. A distinction should be made between *static* and *dynamic* groups. A static group usually corresponds to the structure of an organization and is formed for a long term, e.g. a project team. A dynamic group is usually formed for one session, where the members may come from different static groups.

The *mode* of a session is defined as some condition imposed on users' participation or behavior. The mode of a session can be *open*: a user not on the initial meeting list may join an ongoing session on approval of the chairman, or *closed*: the session is restricted only to participants on the initial list. It can be *public*: everyone can join, or *secret*: the system does not release any information about the session, and the flow of information is encrypted.

Two modes that influence different aspects (output/input) of group and subgroup activities are: *WYSIWIS* mode and *token* mode. In the *WYSIWIS* mode — *what you see is what I see*, some or all participants share the same *view* (the current screen or window output). In the *token* mode — some or all participants of a group work closely; only one participant who has the *token (floor* or *chalk)* may make his input at one time. The latter is useful when close coordination is required. Token mode usually implies WYSIWIS, but not vice versa. Users sharing a chalkboard view may be active simultaneously on different parts; no token is imposed.

A single-user tool converted to a multi-user tool will usually have the WYSIWIS and token modes throughout the session [Sarin84, Lantz86, Abdel-Wahab88]. A token is needed for a converted single-user tool because if more than one user simultaneously issues tool commands, the tool commands will be interleaved into the tool. If the commands cause conflict, problems may occur. These problems are serious if no roll-back or only single-step roll-back function is provided by the tool.

2.2. Shared Workspace Model

Workspace is an abstraction that denotes a collection of objects belonging to some cooperative work and the software tools needed to access these objects. For example, researchers writing a joint paper will have in their workspace objects such as sections,

figures and tables, and tools such as editors abd formatters. *Figure 1.1* shows a meeting: each participant has in front of him a workspace where he can operate with some tools on the same objects that other participants see. A *shared workspace* is a workspace shared by users for cooperation.

At first sight, this model looks simple. Several variations complicate the scene:

1. Users may vary: the joining participants may not be fixed, e.g. in a public session. Users may form subgroups sharing different workspace objects. Or an object owner who allows some users to share may change his mind later and allow more users to share or retract sharing. A departing user may want the remaining participants to share his objects.
2. Objects may vary: a user may not foresee all the objects to be shared within a session, and may bring an object into the workspace at any time during a session. It may not be possible to predict what will be contained in the workspace.
3. Tools may vary; a session may use several tools. A tool can be a general multi-user tool or a single-user tool converted for multi-user cooperation. *Figure 2.1* shows a multi-user tool with a centralized control scheme: the server mediates user agents' access to the shared workspace. The edge between a user agent and a workspace object or between the server and an object represents an access path. The server may allow each user agent to access an object after mediation or it may require that all user agents access an object through it. The edge between each user agent and the server represents a control path, used to send and receive requests and control messages. *Figure 2.2* shows a decentralized control scheme: user agents coordinate among themselves their access to the shared workspace. In *Fig. 2.3*, a centralized control scheme is used in converting a single-user tool to multi-user cooperation. A token is created and managed by the session server. The session server also mediates access to the shared workspace.
4. Access may vary: workspace objects can be shared for different levels of access: e.g. read-only, write-only, read/write, execute-only, etc. The user who brings an object into the workspace may change his mind during a session and allow other participants greater access rights to his object, or he may reduce or cancel their rights. Different objects accessed by a tool may lie in different user domains. Under what privilege domain should a tool be run? Sharing a single-user tool for real-time cooperation has its own problems (see Sec. 2.7.3).
5. Working mode may vary: different working modes may imply different security concerns. For example, an object initially not available for access may be made available for access under the owner's supervision, made possible by shared viewing (WYSIWIS).

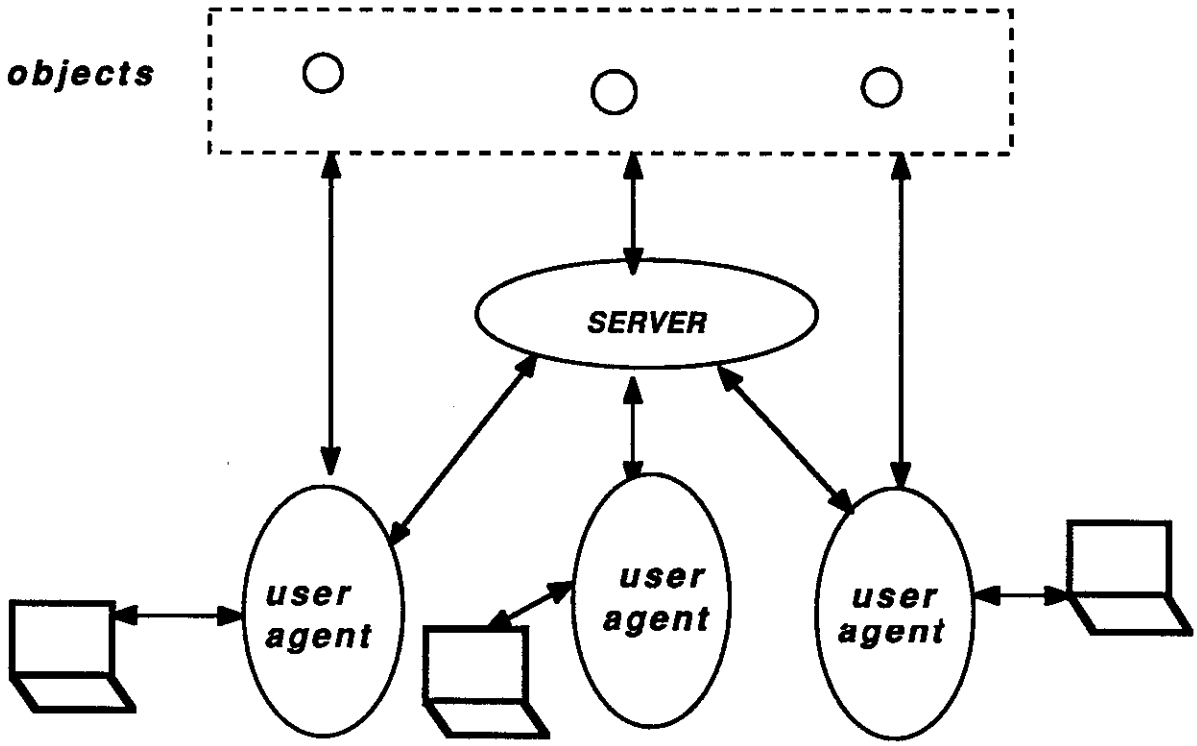


Fig. 2.1 Multi-User Tool (Centralized Control)

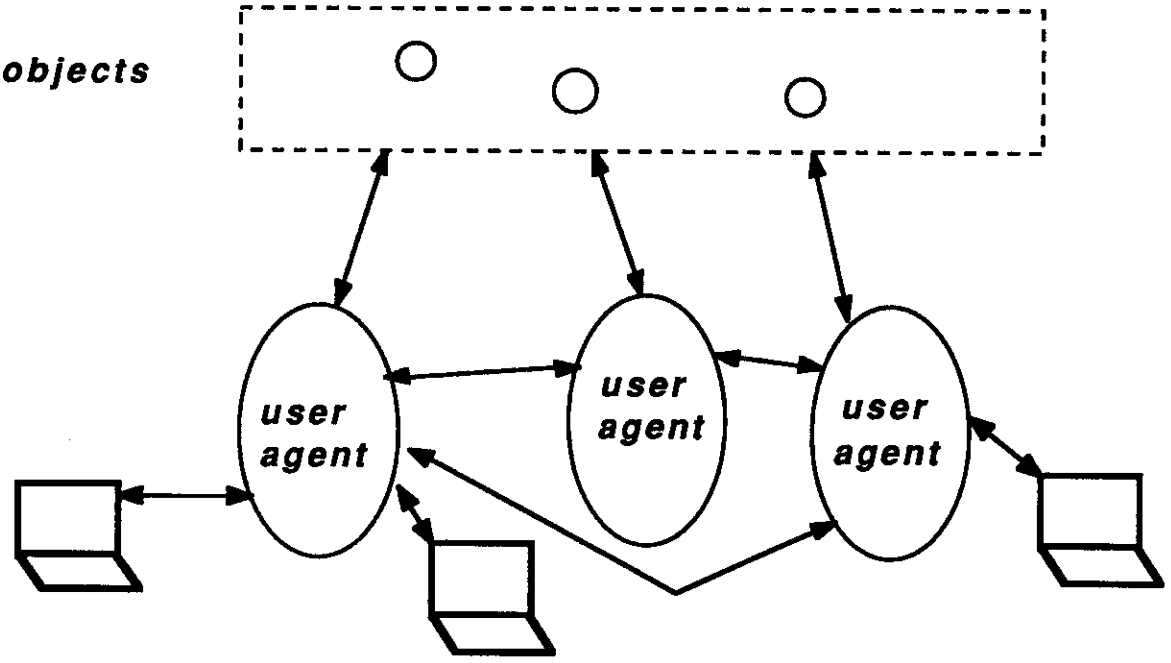


Fig. 2.2 Multi-User Tool (Decentralized Control)

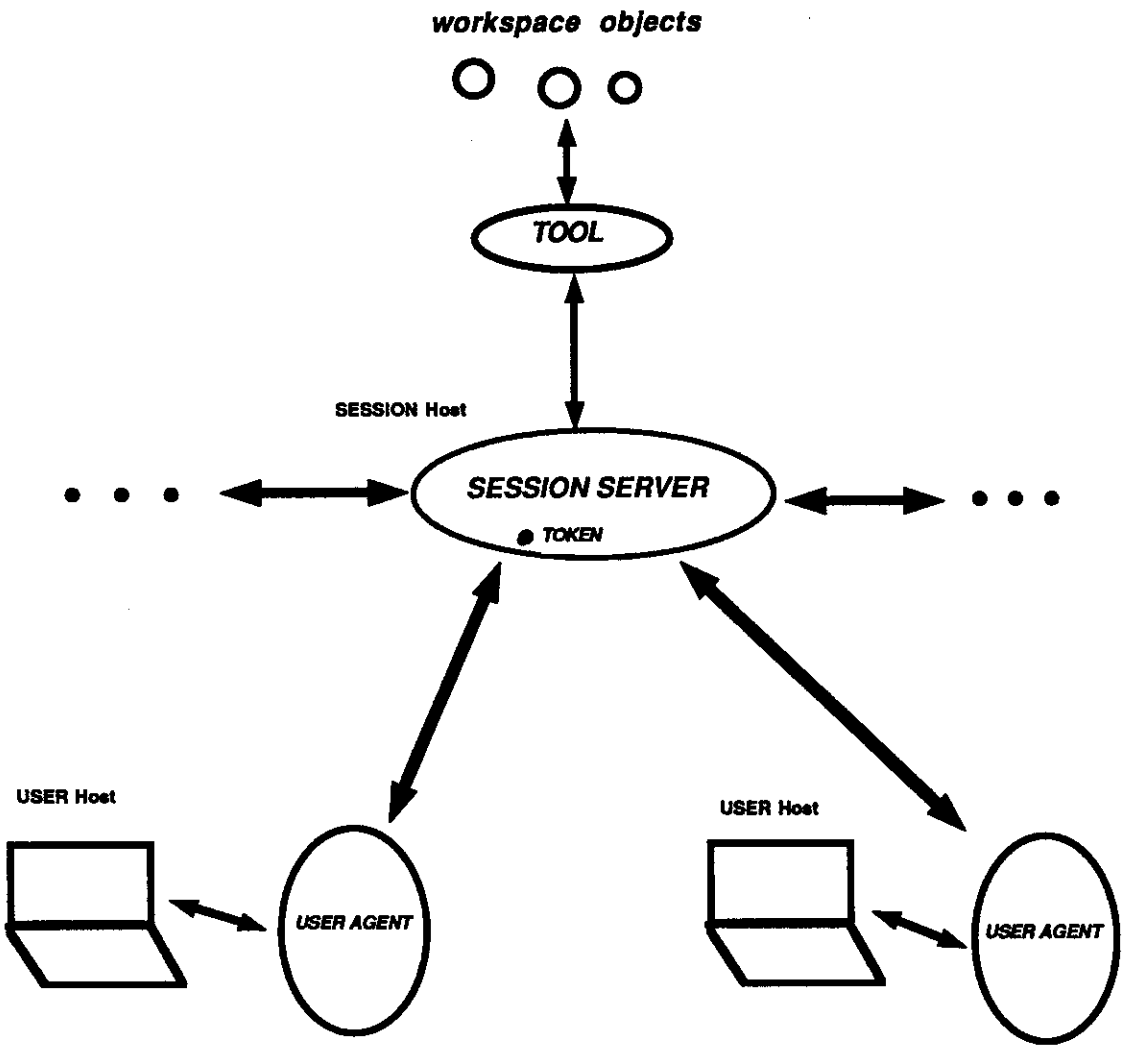


Fig. 2.3 Sharing a Single-User Tool in a Real-Time User Cooperation

In some cooperation, *roles* are assigned, with each role having different access rights to objects in the workspace [Fish88]. A user joining a cooperation is assigned one or more roles. Even with this scheme, the same requirements may still arise. The degree of sharing an object may depend not only on the roles, but also on the degree of trust among the participants. It is possible that a user shares some object with another user, but not with a third user in the same role as the second user. These variations are common to cooperation.

2.3. Dynamic Groups

Envision people having a meeting in a conference room, communicating with each other and occasionally forming discussion subgroups. People in the conference have freedom in exchanging messages and sharing their views. As people join or leave the conference, the union of participants forms a dynamic group.

To create a computer-supported dynamic group session, there are two possibilities:

1. The system assigns a name (usually a numeric identifier) when the chairman creates a session; he then communicates it to the participants through some real-time channel like telephones.
2. The chairman specifies a name in advance and communicates it to the participants; this can be done through some real-time or non-real-time channel like electronic mail.

Both approaches are used in the operating system world. For example, when a process is created, an identifier is assigned. Or when a file is created, the user gives it a name. For multi-user sessions, the first approach is not user-friendly and causes inconvenience, as real-time channels may not be available when a session is being created. It cannot be adopted when a real-time channel is not present. The system-assigned identifier approach usually has no problem for single-user applications because mechanisms are provided to establish real-time channels between cooperating processes under a single-user domain. For example, if the cooperating processes are parent and child, the parent can create a channel and pass to the child a capability to use the channel.

The second approach has no such inconvenience, as users furnish a name on which they agree beforehand. This approach allows the possibility of naming conflicts: two sessions may be created with the same name. This problem can be overcome with the partitioned name space approach as used in most recent file systems. A solution is presented in Ch. 3.

With the introduction of multi-programming computer systems, a user is able to run several programs simultaneously. This implies that a user is able to create or join

several computer sessions simultaneously. Here are the requirements of a dynamic group, analogous to the scenario of a session:

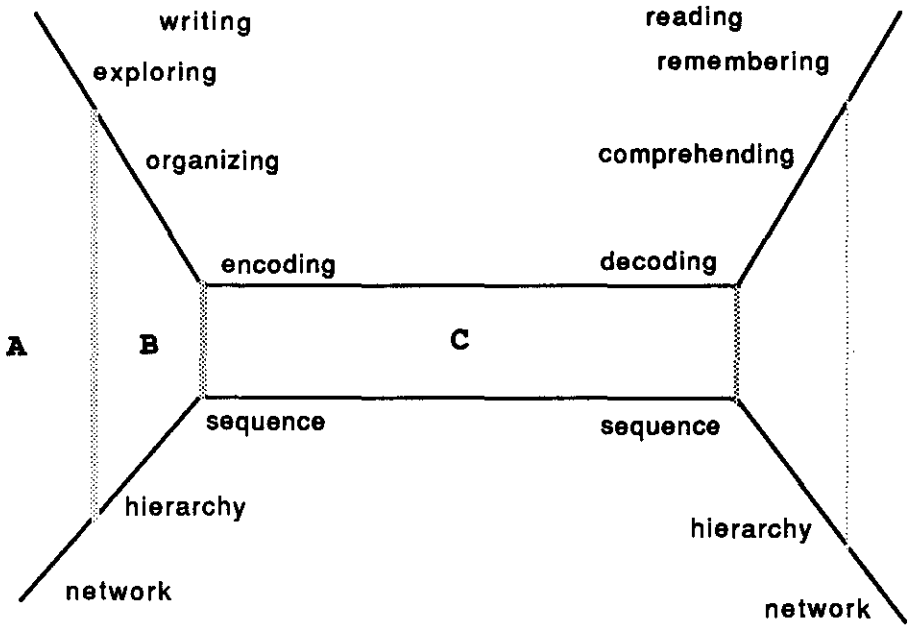
1. A user should be able to create one or more dynamic groups simultaneously.
2. A user should be able to join one or more dynamic groups simultaneously.
3. A user should be able to leave a dynamic group at any time he wishes. He should be able to re-join if the group still exists.
4. One-to-one or one-to-many communication should be provided to dynamic group users.
5. Different cooperation modes should be available for dynamic group users.

The requirements have not included whether the chairman or a participant of a session should be allowed to terminate the session, because this depends on the particular session itself. Sometimes a started session is deemed as owned by all the participants and should not be terminated without the agreement of all or a majority of the participants. Dynamic groups and the shared workspace model together form the basis of shared workspace cooperation.

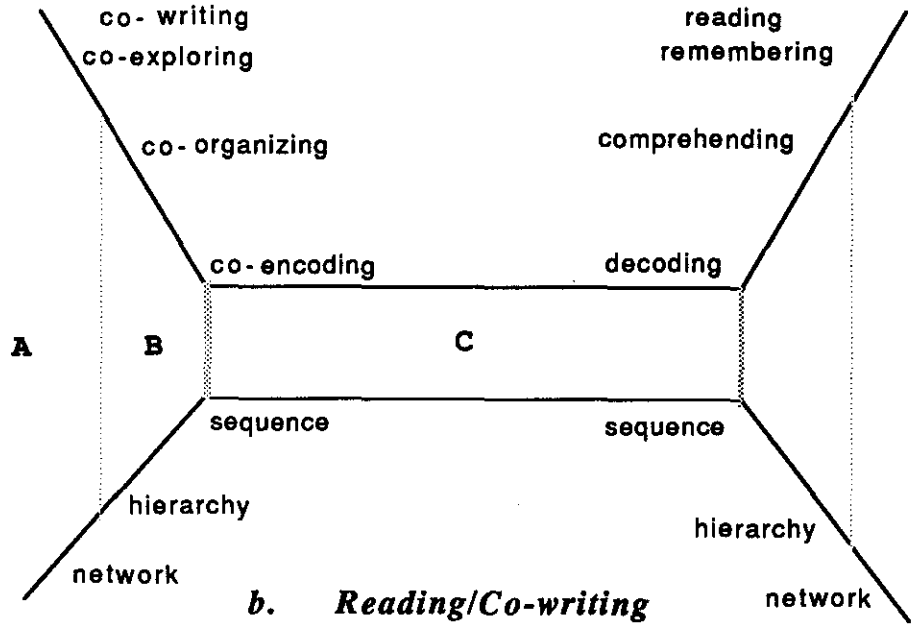
2.4. Sharing in Workspaces

Shared workspace cooperation is the core of most real-time cooperation. The following observation sustains this claim. *Cooperation is done through sharing.* In *Fig. 1.1*, it can be seen that the workspace objects and tools are shared. Further, *pointers (cursors), the token, views* and *user privileges* can be shared. Sharing of the token, pointers, or views is usually done within a real-time cooperation.

An object can be partially shared: there is one owner, who grants some rights to collaborators. An object can also be fully shared: there is more than one owner. Tools can be shared real-time or non-real-time. The owner of a tool grants his collaborators access rights to his tool. He may allow the tool to be used in an off-line cooperation under his domain: "setuid" process in *UNIX* [Ritchie78] is an example. A single-user tool can be converted for real-time cooperation by letting users share it. User privilege(s) can be partially shared: a user lets his collaborators work with him under his domain. User privilege(s) can be fully shared: a group cooperates real-time on an object has as its access privilege the union of the participants' privileges.



a. Reading/Writing (John B. Smith)



b. Reading/Co-writing

Fig. 2.4 Cognitive Models

The claim is also supported by the following observation of a co-writing activity from the cognition point of view. John Smith proposes a cognitive model for a reading/writing activity as shown in *Fig. 2.4a* [Smith87]. He explains reading as an activity of taking the linear stream of text, comprehending it by structuring the ideas hierarchically, and transforming it into long-term memory as a network. Writing is seen as the reverse activity.

Based on his model, I propose the following (*Fig. 2.4b*) for a reading/co-writing activity. Traditional support for cooperative writing, via electronic mail and commenting, occurs at stage C in an off-line fashion. A co-writer or commentator understands his colleague's work by following a reader's decoding sequence. With real-time cooperation tools like *Cognoter* [Foster86] or *WE* [Smith87], the cognitive activities of co-writers or commentators are facilitated by shared visual spaces through stages A, B, and C. Each worker knows his co-worker's activity by observing him in each stage, reducing the overhead of the decoding sequence. From this point of view, the shared workspace is a necessity for cooperation to achieve a shared structure of understanding. An efficient mechanism is needed to facilitate shared workspace cooperation in each cognition stage.

2.5. Token Management in a Session

When users want to have a close coordination or when a single-user tool is converted for multi-user cooperation, a token is needed. How can users share access to a token efficiently and fairly? For effective work, the token holder must be guaranteed an uninterrupted *quantum* of time, once he gets the token. For fairness, other participants must be able to request the token and obtain it within a certain known waiting time. When the token holder has to release the token, he is given a brief *grace period*, to complete his current task. When the grace period expires and if the token holder still has not released the token, it is then grabbed away. The values of quantum and grace period should be adjustable, and may depend on the tool in use and the number of users.

When several users request the token, they are placed in a queue according to some criterion (e.g. FIFO, priority first). The first user in the queue is the next to get the token. A user should be able to cancel his request, after which he is removed from the queue. *Figure 2.5* shows the detailed user token states, which are explained in Sec. 6.2.2.

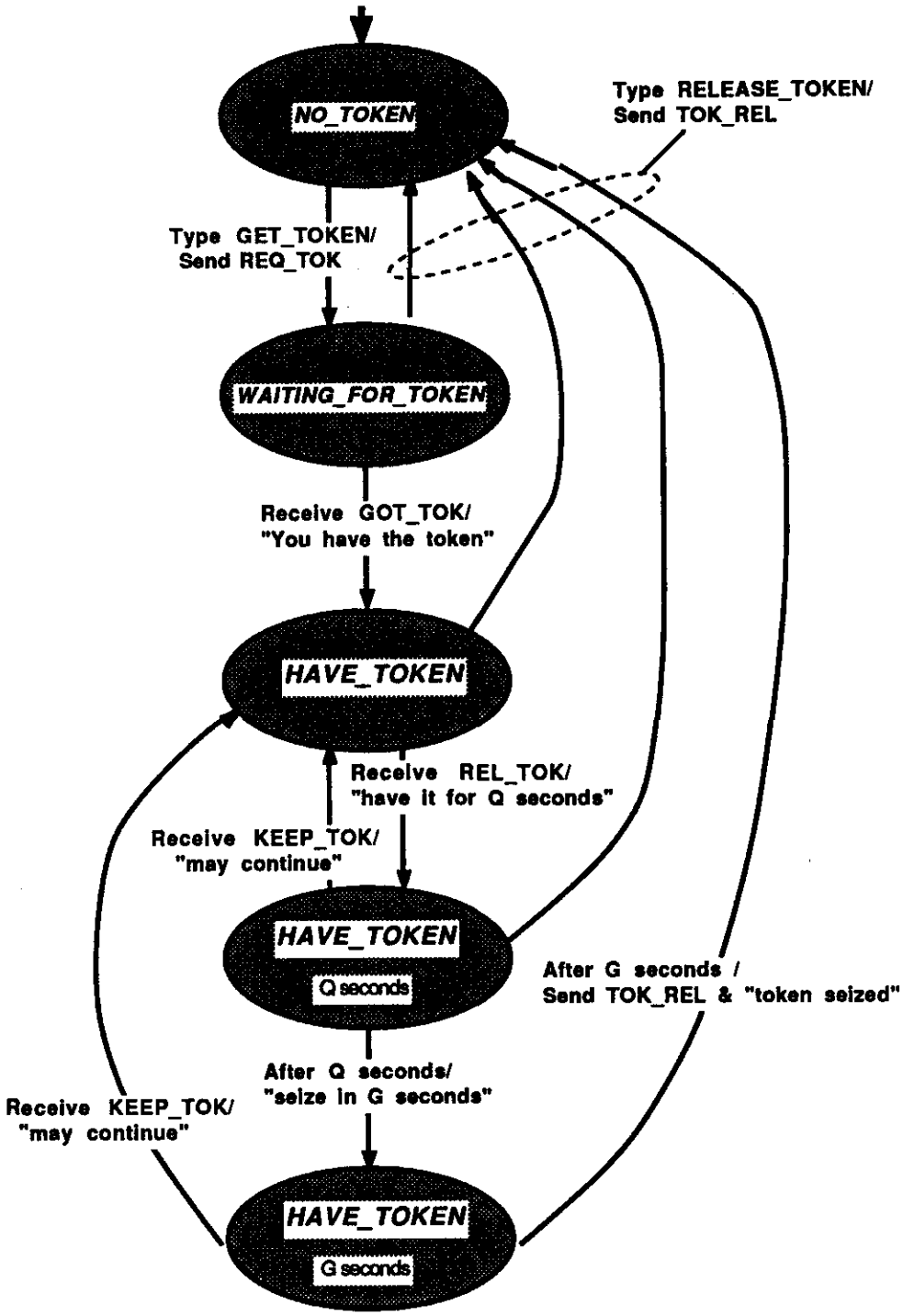


Fig. 2.5 Token Control States for User Process

2.6. Design Aspects of Shared Workspace Cooperation Tools

In this section, we summarize aspects of a tool for shared workspace cooperation.

Input Control

Input control concerns who has the floor to talk. Input control can be thought of as a special case of concurrency control: resolving the conflict of access to the token. Because input control is directly related the human interface, it deserves a separate discussion. For close coordination, the users are provided a token, and whoever has the token can make input. Interesting issues regarding how to share the token effectively and fairly have been discussed in the previous section. Alternatively, no token is imposed for maximum freedom, i.e. more pieces of chalk are provided and the users are free to make their input.

Replication

A session workspace can have a single copy of workspace objects and tools; this is useful when tools are not available on all participants' machines or when the objects are large. Alternatively, a session workspace can have replicated copies of objects and tools for all the participants; this is useful when performance is the issue. A session can also have replicated copies of objects and a single copy of tools [Suzuki86] or vice versa.

When replicated copies are used, issues to be resolved are: whether we let the users be aware of the replicated objects or not, whether the replicated objects are removed after the session or not, whether objects are updated instantly or occasionally, how to synchronize updates to all the replicated copies, how to insure that the replicated workspaces are identical.

Synchronization

A session can be centralized (*Fig. 2.1*) with a *server* synchronizing messages or resolving contentions to object access, or decentralized (*Fig. 2.2*) with all *user agents* coordinating and synchronizing distributedly. The former offers ease of design and synchronization. The latter is more reliable because there is no single point of failure: if one site goes down, the other sites can still continue with their copies. Centralized control can have hierarchies: if the cooperating users span a large area, one can link neighboring user machines into hierarchical groups to reduce message overhead, with a server for each group. A "root" server then coordinates and synchronizes all the group servers. The issue of replication is also closely related here: we can have a replicated copy for each group, with users in the same group sharing the same workspace copy.

I/O Transfer

User input and tool output sometimes need to be transferred during a session (*Fig. 2.3*). User input may be processed locally, with an encoding specified by a high-level protocol like *VGTP* [Lantz84] transferred to the *session server*. In a single copy workspace, the tool output can be similarly encoded and transferred to the local user agents. Alternatively, user input and tool output can be transferred in a low-level form (raw keystrokes or bitmaps). The architecture can be a mixture of these two, e.g. low-level input and high-level output encoding.

2.7. Desirable Features for Shared Workspace Cooperation

The basic entities of a shared workspace cooperation are users, tools, and objects. We elaborate on each entity and describe the features important to cooperative work.

2.7.1. Dynamic Groups Formation and Activities

A *protection group* [Saltzer75] mechanism exists in most operating systems, but it is usually oriented toward static groups: group structure cannot be changed easily. In a multi-user cooperation, participants may come from different groups. A dynamic group is formed for each session. The members of a dynamic group may not even be fixed, which is the case in a public session. A dynamic process group mechanism can be found in some operating systems, e.g. the *V* system [Cheriton84]. It is difficult to join a group in *V*: a group is created by asking the system for an identifier. For processes to join the group, this dynamically assigned identifier needs to be distributed first through some channel. Also *V* supports only the communication aspect of group activities. Other activities in a dynamic group, like view sharing or token passing, are not supported. Most effort of implementing multi-user tools is spent providing these functions (see Sec. 6.1).

A dynamic user group structure to model these cooperative sessions is needed. The structure should be easy to extend to a homogeneous distributed network, with computers running the same operating system. Based on this structure, functions can be provided to support dynamic group activities. Without such support, the formation and activities of dynamic groups need to be implemented from scratch for each application.

2.7.2. Development of Multi-User Tools

Section 2.6 discusses some design aspects of multi-user cooperation tools. Can mechanisms be provided to support the development of these tools? For example, what can be done by an operating system to support replicated workspaces? Should an operating system support replicated workspaces explicitly or (for performance only) implicitly? This interesting issue, although challenging, has not been dealt with in the dissertation research.

Let's take a closer look at *Figs. 2.3* and *2.1*. In *Fig. 2.3*: the users are sharing the tool process within a session. For multiple users to share this process, their standard input/output channels need to be redirected to it. This is because a traditional process (e.g. in *UNIX*) is attached with a single-user standard input/output channel. This effort of connection/redirection and communication overhead can be saved if the operating system allows a process to be attached with multiple users. Sharing is also easier with this mechanism. The same observation applies to *Fig. 2.1*.

2.7.3. Sharing a Single-User Tool in a Real-Time Cooperation

Many single-user tools can be converted for real-time, multi-user cooperation. Can general-purpose operating system mechanisms be developed to support this function? The following elaborates a problem that needs to be taken care of in the development of such mechanisms.

Suppose a group of remote users forms a shared workspace and uses a single-user tool for real-time collaboration (*Fig. 2.3*). Each member of the group should be able to access resources in the shared workspace. On the other hand, he should not be able to access resources outside the shared workspace, if he is not permitted. Now comes the problem: a group of users cooperates with a converted single-user tool on a machine where one group member grants them access. Since many tools have *escape* commands, the users may escape from the tool, and either access another object or execute another tool. Also many tools have built-in commands to access objects other than the workspace objects.

This kind of *tool escape* can be dealt with if the users share the same view all the time and the tool is run under an active participant's privilege domain. That participant acts as a host and is responsible for any problem caused by an escape. It must be noted that when that participant wants to leave then the tool needs either to be terminated, or to be run under another active participant's privilege domain.

2.7.4. Sharing of User Privileges Within a Session

First some definitions are given: an *access control list* is a list of subjects that are authorized to access some object. A *capability* is a token that allows the possessor to access an object. It is usually implemented as a data structure that contains a unique object identifier and access rights to the object.

For capability systems where a capability cannot be sent from a user to another user directly or access control list systems where the access control list cannot specify all possibilities or cannot be changed by a user freely, sharing privilege across domains is usually done in two approaches. It can be done by associating each domain with a process; the communication is done via interprocess communication. Or it can be done by associating multiple domains with a single process; the communication among protected subsystems is done via domain switching [Saltzer75].

Sharing objects via interprocess communication sometimes implies storage overhead: objects need to be replicated. Passing an object by reference is not feasible because the receiving process still does not have an access right to the object. If participants aim at working on the same objects, then the replicated objects need to be consistent. Sharing objects via interprocess communication also has the following problems. With centralized control (*Fig. 2.1*), the server is usually created by the chairman of the session. When he leaves the session, he may not want the server to continue running under his domain, and a new server needs to be started (*departing chairman problem*). Another problem (*departing workspace-object-owner problem*) related to both schemes (*Figs. 2.1, 2.2*) is that if an owner of workspace objects or tools leaves the session, he may still want the remaining participants to share his objects. There is no way to achieve this goal unless he leaves his agent process running under his domain, which may lead to an undesirable security break.

By thinking of the privilege each user owns as a domain, the second approach associates multiple domains with a process by domain switching. The essence of changing domains is, in access control list terms, to change principal identifiers; in capability terms it is to acquire the set of capabilities of the new domain. In both cases, domain switching is usually achieved through a *protected procedure call* [Saltzer75].

Sharing via the protected procedure call in a real-time cooperation has the following *shared workspace problem: 1.* a cooperating user may not foresee all the objects to be shared within a session. The pre-written protected procedure may not cover all the shared objects; *2.* an object owner may change his mind in a session to allow other participants more access rights to his object, or he may want to reduce or cancel their rights; *3.* for an open or public session where the participants vary, it is difficult to write the protected procedure in advance because an object owner may want to share his object with only a subset of the participants or he may want to grant different access

rights to different users; *4.* even when an object owner disallows write access to his object normally, he may relax this in a session and let a participant write over his object under his supervision. They share the same view so that the owner knows what the other is doing to his object; *5.* when simultaneous manipulation of objects across multiple user domains is needed, changing principal identifiers cannot achieve this requirement; *6.* even though distributed systems [Svobodova84, Tanenbaum85] exist, dynamic sharing in a distributed network is usually difficult to achieve.

Although it is possible for users to write real-time cooperation tools using the domain-switching mechanism to achieve privilege sharing, it is awkward and inflexible to deal with the shared-workspace problem. The programmers can be relieved of this burden if a simple mechanism can be provided by the operating system.

To summarize, traditional protection systems originated from a military environment that assumes an off-line cooperation environment where users are mutually suspicious. For real-time cooperation, the users are usually trustworthy. Through the interaction and the ability to supervise in a session, a user may allow others access to some of his objects that he would not allow otherwise. In this sense, permission of "who can do what to which object when" in a session is not necessarily identical to that of an off-line cooperation. The domain switching mechanism serves well for a mutually suspicious environment because the security concern is usually strict and static. While in a real-time session, however, the security concern can be flexible and dynamic. The same mechanism then seems not to serve well.

Note that we are not suggesting replacement of the usual protection mechanism. Instead, some mechanism that better supports real-time cooperative work needs to be provided, making up for the rigidity and inconveniences of the usual protection mechanism.

2.7.5. Provision of Jointly-Owned Objects

In real life, people own objects jointly: like the husband and wife sharing a joint account, partners jointly signing a contract, etc. In computer systems, an object is usually owned by a single user. Examples of jointly-owned objects can nevertheless be seen in computer systems: a virtual circuit [Tanenbaum88] that, once established, each party can read from, write into or disconnect; a link in a hypertext environment that spans across nodes in files of two users [Engelbart84a], each of whom jointly owns the link and should be able to tear it down whenever appropriate; a multi-threaded task [Accetta85, Rashid86] where each thread shares the same address space and capabilities, and can destroy the whole task.

With the falling cost of hardware and communication, today more and more people are experimenting with computer-supported cooperative work. We see people co-

writing a paper, or developing or debugging a program together. The object being worked on is usually jointly owned. Another requirement arising from such an environment is the following.

A multi-user cooperation tool when executed usually creates for each participant a user agent (*Figs. 2.1, 2.2*), running under that participant's domain. As multi-user tools are written, some of them will be written by system programmers, and some by the collaborators themselves. For the former case, tools installed will be trusted by the collaborators and used without any security concern, as users usually trust system utilities. For the latter case, if a participant joins a multi-user tool written by others, how can he know that the user agent executed in his name is not a Trojan horse [Saltzer75]? This doubt can be relieved when a collaborator knows about the multi-user tool he joins. But it cannot be totally removed unless he is sure that the multi-user tool has been installed and cannot be replaced unless he is notified. This requirement cannot be fulfilled with traditional single-owner objects, since one of the authors of the multi-user tool usually has full ownership of it and has rights to make such a change.

Under certain circumstances, an object may not be accessed until a sufficient number of users are together (referred to as *quorum-based object*). An example is a committee meeting that requires the presence of the majority of members. Another possible requirement is that an object may not be accessed without certain user's or users' presence (referred to as *authority-based object*). An example is a meeting that cannot be started without the chairman. What mechanism meets these requirements?

The problem of using a single-owner object mechanism to support jointly-owned objects is that when an object is created, one member of the group is overtrusted with the full ownership. Even if the group decides that the object is read-only, the assigned owner can still change its protection mode and write over it. Deletion of the single-owner from the group would mean a reassignment of all the jointly-owned objects to another user in the group.

Questions arise: how to maintain or verify access to a jointly-owned object? A difficult problem is the following: how to handle an access request to a quorum-based object? With computer access, the joint-owners need not even gather together to access a jointly-owned object. For some instance, say users with different interests collaborating in subgroups on different sections of an object, the system needs to know whether the requests issued from the joint-owners' processes are related. For example, if a jointly-owned object has four owners, and a quorum equal to two, suppose the owners form two groups. The read requests from these two groups of joint-owners should not be correlated by the system. The system needs some evidence to distinguish between requests from these two groups. The problem is the same with authority-based objects.

What is the protection model for jointly-owned objects? Although many protection models have been seen in the literature [Lampson71, Graham72, Harrison76, Linden76,

Landwehr81], none have dealt with this possibility.

2.8. Requirements for Shared Workspace Cooperation

The following criteria are suggested for an operating system that supports cooperative work:

1. It provides a dynamic group structure to model cooperative sessions.
2. It supports building of multi-user tools.
3. It supports conversion of single-user tools for real-time cooperation.
4. It facilitates sharing in a real-time collaborating environment. In addition to sharing of messages, we need sharing of views, the token, and capabilities.
5. It does not overlook the usefulness of jointly-owned objects.
6. It does not cause security breaches.
7. It supports remote cooperation in distributed networks.

Note that we do not claim the above is a complete list of requirements, e.g. concurrency control is not included. The list is compiled according to the desirable features identified. In the following chapters, solutions are presented to support these requirements.

CHAPTER 3

AN ARCHITECTURE SUPPORTING DYNAMIC GROUPS

The shared workspace model sheds light on how objects and tools are shared within a cooperation. To support real-time cooperation more effectively, we also need to support the users. This chapter presents a mechanism that supports dynamic groups and sharing single-user tools for cooperation, the requirements having been described in Secs. 2.7.1 and 2.7.3. A functional interface is proposed with a *UNIX*-style *C* [Kernighan78] library call format. Section 6.2 contains a complete list of the proposed dynamic group interface.

A *dynamic group* architecture is proposed to be placed at the system library/server level. It makes no assumption about the existence of a windowing system. Considering the diversity of group activities and the possibility of remote collaboration, support from this level is most appropriate (see Sec. 6.1 for more arguments supporting this).

Machines form a *cluster* (*Fig. 3.1a*) to facilitate users' cooperative work. Resident on each machine in the cluster is a *system-server* daemon, forming the backbone of dynamic groups (*Fig. 3.1b*). A global dynamic group pool is maintained by all system-servers in the cluster. Each dynamic group and its activities can then be maintained by a spawned *group agent*. Detailed implementation is discussed in Sec. 6.2. Note that the proposed architecture is only one possible solution. It is made as general as possible to provide a basis for user group formation and activities. More features can be added.

Several assumptions have been made here:

1. Machines within a cluster run the same operating system. Extensions to a heterogeneous environment need more investigation.
2. Each system-server has some way to authenticate the others, and has the ability to authenticate users on its machine.
3. Each user in a cluster is uniquely identifiable, and can be authenticated by at least one machine within the cluster.
4. Each process is invoked by a user and tagged with his user identifier. Each process has a set of standard input, output and error channel descriptors.

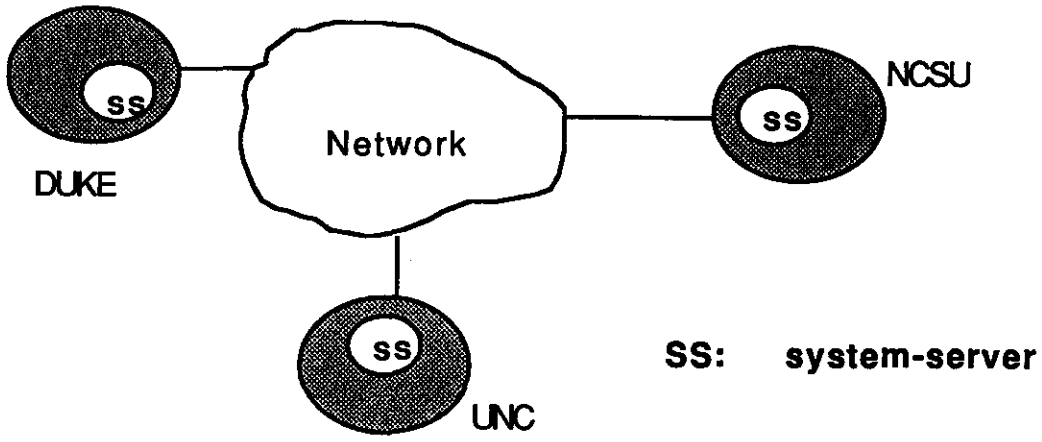


Fig. 3.1a Cluster

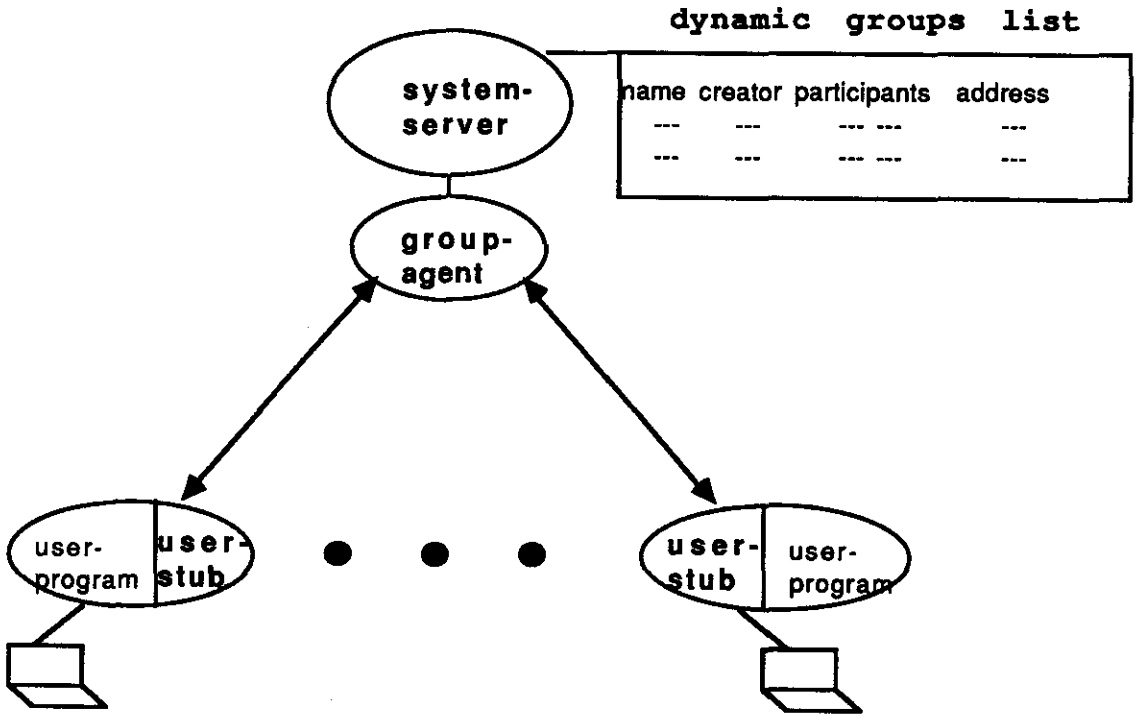


Fig 3.1b Implementation of Dynamic Groups

3.1. Design Concepts and Functional Interface

A dynamic group structure is provided to allow users to join or leave dynamically, to send or receive messages, and to share their views dynamically. Participants need not come from the same machine. As mentioned earlier, requiring the participants use a system-assigned identifier to join a meeting group is not user-friendly. Instead, we allow people the liberty to name their conference or meeting group. The system is not responsible for assigning a unique name for each group. Instead, the system keeps a record of all the groups created. Two groups may be named identically. To avoid such name conflicts, the record for each group should include the name of its creator.

To join a dynamic group, a user simply provides the group name and its creator's name. As users generally know who the meeting chairman is (usually the user who informs participants of such a meeting), this should cause no difficulty.

With this dynamic group structure, users in a group can send messages to each other or the whole group. For sharing views and the token, a **wysiwis** (what you see is what I see) mode is provided.

Any user in the cluster is free to create a non-existing dynamic group by specifying its name, *mode (public, closed, secret)*, and *participants-list:*

create_group *(group_name, mode, participants_list)*

```
char *group_name;       name of the group, '*' means a pointer
char mode;              mode can be 's' (secret), 'p' (public), 'c' (closed)
char *participants_list;
                        a NULL-terminated list of participants' names separated by spaces
                        a here names are cluster-unique user identifiers
```

The *participants_list* is specified without the creator's name on it. For a public group the *participants_list* is not specified (null), because anyone in the same machine or cluster can join. Secret or closed groups are defined as in Sec. 2.1. A user can create more than one group through his process(es), i.e. he can create several groups through one single process or through several processes — assuming he is able to run concurrent processes.

Similarly, a user can join more than one group simultaneously. A user joins a group by issuing **join_group** from his process. The call will return successfully if he is one among the *participants_list*. The user becomes an active participant in the dynamic group. An active user's process may leave its group and then rejoin if the group still exists. When a user process exits, it leaves all its groups.

join_group *(group_name, ifnotexist, time_out)*

char *group_name;
int ifnotexist; *whether to block if the specified group does not exist*
int time_out; *the call is blocked until the group is created or time-out expires*

leave_group *(group_name)*

char *group_name;

Flag *ifnotexist* can be *BLOCKED* (the call will be blocked if the group does not exist), *NON_BLOCKED* (the call will return error if the group does not exist) or *TIMED_BLOCK* (if the group does not exist, the call will be blocked until the group is created or the specified *time_out* expires). Note that *time_out* need not be specified (null) if the flag *ifnotexist* is not *TIMED_BLOCK*.

To avoid naming contentions, a joining participant is required to specify the *group_name* in the format of *group_name:creator_name*. The combination of *group_name* and *creator_name* spans a global name space to the extent of a machine boundary or a distributed cluster of machines, if a cluster has been formed. Joining a dynamic group should not require knowledge of its location, which makes session migration possible. More implementation details are discussed in Sec. 6.2.3.

The names of a group and its creator can be known beforehand or ascertained by the following **list_groupname** primitive.

char * **list_groupname** *( )*

This call returns a NULL-terminated list of all existing groups in the cluster, in the format {group_name:creator_name {participant_name}* }*, where "*" means repeating zero or more times. The list will include those secret groups only when the issuing user is one of the participants in those secret groups or when the issuing user is a superuser. A user can get further details of a non-secret group through the following:

GROUP_LIST * **list_group** *(group_name)*

char *group_name;

A list of the following information is returned:

typedef struct group_info {
  char mode;
  char *p_list; *a NULL-terminated list of participants' names separated by spaces*
  char *ap_list;
    a NULL-terminated list of active participants' names separated by spaces

} GROUP_LIST;

Participants are users on the *participants_list* specified in **create_group**. Active participants are participants who have already joined. A user on a secret group list can get information from that group also through this primitive.

close_group *(group_name)*

char *group_name;

Close_group allows a dynamic group entry to be marked as closed: the dynamic group will no longer accept new or late-joining participants, yet the session continues until all current participants leave. It is analogous to closing the conference door and is useful when a session has started and does not want to be disturbed. Only the group creator is able to do this.

In the following it is shown how some features can be supported through this dynamic group structure. Based on different design considerations or tradeoffs, this support can be optional. Several calls are provided for active users of a group to share views or messages. For a system that has full-fledged message communication support, a designer may choose not to provide the following **send** or **receive** primitives.

send *(user_name, group_name, message, length)*

char *user_name, *group_name, *message;
int length;

MESSAGE ***receive** *(user_name, group_name, buffer, length, blockornot, time_out)*

char *user_name, *group_name, *buffer;
int length, blockornot, time_out;

A message can be sent to an active participant within the same group or broadcast to all active participants in the group (excluding the sender). We leave it to the implementors to decide whether to buffer a message when some users have not joined yet. For sending a message to the whole group, the field *user_name* is left null. For receiving a message from anyone in the group, the field *user_name* is also left null. If a user has two representatives (i.e. processes) in the same group, then the message will be sent to each one. A user provides a *buffer* and the maximum *length* of data he wants to receive. The *blockornot* flag specifies whether the call is *BLOCKED*, *NON_BLOCKED* or *TIMED_BLOCK* if no message is available. In **receive**, the caller may specify from whom the message is to be received. *NULL* is returned if there is no message; otherwise the message is placed in *buffer* and the following information is returned:

```
typedef struct ms {
  char *from_user;      from which user this message comes
  int length;           length of the received message
  } MESSAGE;
```

The output, or view, of a process can be shared through a **wysiwis** mode in the same group. A participant allows the *standard output* of his process to be shared by issuing **wysiwis** (*Fig. 3.2*). View sharing is enabled until the mode is ended. Each user in a group is eligible to create a **wysiwis** mode. A token is created, a token queue is associated with it. Other participants in the same group can get information about **wysiwis** modes by **list_wysiwis** and enter one by **enter_wysiwis**. Assuming a cooperative environment, any user process in the same dynamic group as a **wysiwis**-creator may enter the mode. A group thus defines a secure and cooperating environment like a conference: outsiders are not able to enter a **wysiwis** in the group; insiders are free to enter any **wysiwis** in the group. When **enter_wysiwis** is issued, the entering process is suspended for a reason to be explained shortly. Thereafter the participant shares the same view as the user who created the mode.

To make his input, a **wysiwis** participant first needs to get the token. Each **wysiwis** participant is eligible to request the token. Users requesting the token are placed in the token queue according to a policy chosen by the implementors. When released, the token is given to the first user in the queue. Input from a user holding the token is fed into the **wysiwis**-creator's standard input channel until the token is released. Users not holding the token are disabled for input. Standard output of the **wysiwis**-creator's process is replicated to each **wysiwis** participant. From this description, we see it is best to suspend a **wysiwis** participant's process after issuing the **enter_wysiwis** call because he is no longer in control of his original process. Note here that a process that issues **wysiwis** is not suspended.

A user process can enter only one **wysiwis** mode. If it were allowed to enter more than one **wysiwis** mode, there would be conflict in redirecting its standard input channel.

```
int wysiwis ( group_name, quit_signal, get_signal, release_signal, quantum,
              grace_period )
char *group_name;
char quit_signal, get_signal, release_signal;
int quantum, grace_period;
```

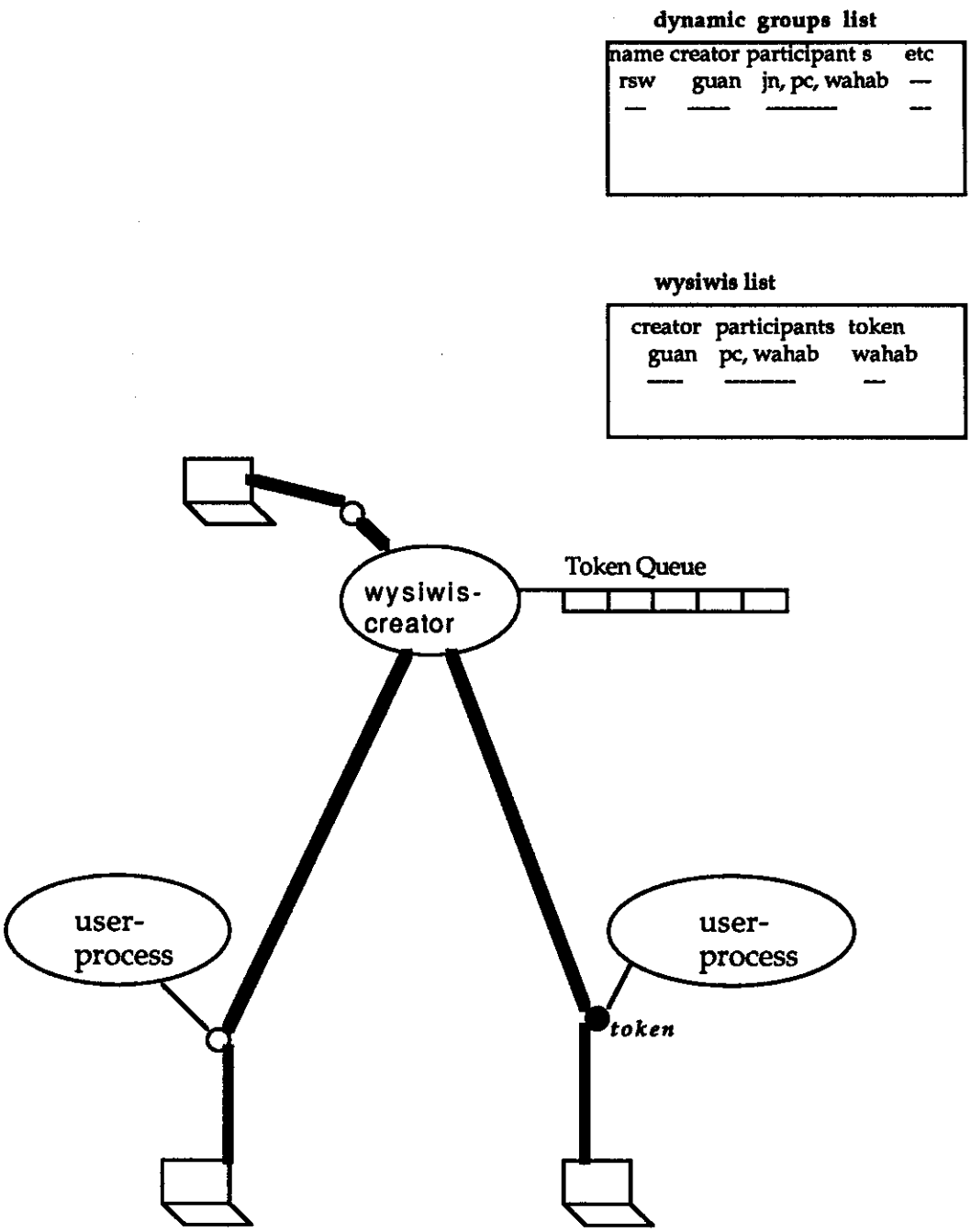


Fig. 3.2 WYSIWIS

The *quantum* and *grace_period* are as described in Sec. 2.5. Values for the *quantum* and *grace_period* are specified in seconds by the **wysiwis**-creator. *Get_signal, release_signal,* and *quit_signal* are special control signals (e.g. ^C, ^G) that will be interpreted even if the issuing user does not have the token. *Get_signal* is used to get the token; *release_signal* is used to release the token or cancel a token request. When a user in the **wysiwis** mode issues the *quit_signal,* he leaves the **wysiwis** mode, and his process is resumed. His standard channels are also restored. If the **wysiwis**-creator issues the *quit_signal,* the **wysiwis** mode is ended. The other participants' processes are resumed, with their standard channels restored.

An identifier is returned for each **wysiwis,** unique within each group. This identifier is required because a user may have several representatives joining the same group. Hence, "group_name:user_name" may not be sufficient for identification when a user tries to **enter_wysiwis.** This **wysiwis** identifier can be chosen the same as the process identifier of the process that issues the **wysiwis** call.

WYSIWIS_LIST * **list_wysiwis** *(group_name)*
char *group_name;

This call returns a linked list of the following information:

```
typedef struct w_list {
  char *creator_name;       name of the user who creates a wysiwis mode
  int id;                   identifier of a wysiwis mode
  char *name;       a NULL-terminated list of names of users who are in the wysiwis mode
  char *token_queue_status;
  char quit_signal, get_signal, release_signal;    signals used in this wysiwis mode
  int quantum, grace_period;
  struct w_list *next;      next entry in the list
  } WYSIWIS_LIST;
```

The variable *token_queue_status* is a pointer to a NULL-terminated list of names of users. The first user name on the list stands for the user who is holding the token; the remaining names represents users waiting in the token queue.

enter_wysiwis *(group_name, identifier)*

char *group_name;
int identifier;

The *identifier* can be made known by **list_wysiwis** or learned from the **wysiwis**-creator. When a participant process enters a **wysiwis**, his process is suspended until either he leaves the **wysiwis** mode or the **wysiwis** mode is ended by its creator.

Several **wysiwis** modes may be going on simultaneously within the same group; the group is partitioned into several disjoint **wysiwis** modes. This is analogous to a conference with several ongoing panel discussions. A **wysiwis** mode is ended by **leave_wysiwis** in the creator's process or by the *quit_signal* issued by the creator. This kind of control is similar to the use of exit call or a control signal to terminate a process execution.

leave_wysiwis *()*

Since a process can be in only one **wysiwis** mode, no identifier needs to be specified. As a **wysiwis** is ended, the participants' processes are resumed, with their standard channels restored. When a process quits execution, it ends the **wysiwis** mode it has created.

A shared workspace can be achieved by several participants accessing objects in the **wysiwis** mode. The participants can also achieve real-time sharing of a tool by executing a program in the **wysiwis** mode. A shared workspace consists of all objects and tools accessed under such mode. When in the **wysiwis** mode, the **wysiwis**-creator may execute a single-user tool, and the other **wysiwis**-participants will be able to share the same tool using the token. *Example 3.2* shows how **wysiwis** is used to convert a single-user tool into a multi-user tool.

The following primitive has been added in the implementation phase to assist in system maintenance:

destroy_group *(group_name)*

char *group_name;

Destroy_group allows a dynamic group to be removed, i.e. the corresponding dynamic group no longer exists and ongoing **wysiwis**'s are ended. As this call has a destructive effect, it can be made only by a privileged user, e.g. superuser. It is useful when an error occurs or a shutdown is imminent.

Wysiwis provides functions similar to terminal linking [Engelbart68, Engelbart75]. The **wysiwis** primitives provide the possibilities of shared-viewing in different subgroups and in different windows during a group session. Users can form disjoint

subgroups by being active in different **wysiwis**. Each user can simultaneously join more than one **wysiwis** and still have his own private workspaces through the use of multiple windows. (This requires the user to invoke multiple processes, one per **wysiwis**.) Further, **wysiwis** need not be enforced throughout the whole session as is the case with terminal linking. The values of *quantum* and *grace_period* can also be changed according to user needs and the application itself. In these aspects, the proposed mechanism offers more flexibility than the terminal linking mechanism.

This computer-supported dynamic group model is actually more powerful than the real-world dynamic group model: a user may create or join several group sessions simultaneously; he may even join a group session with two representatives.

To achieve remote cooperation, each machine that wishes to cooperate will join a cluster. A global dynamic group pool is maintained for this distributed cluster. A user within a cluster may join any group if eligible. He does not have to log in or have an account on the same machine as that of the group creator. Interesting implementation issues arise in maintaining this distributed dynamic group, for example: how to achieve a unique user naming scheme, how to achieve consistency of dynamic group information across the cluster, how to achieve session migration when the performance of a machine becomes intolerable, and how to perform error recovery, etc. These are investigated in Sec. 6.2.3.

3.2. Examples

Two examples are presented. It is assumed that the following system calls exist: *execlp* (execute), *getlogin* (get user login name), *exit*.

Example 3.1 — Multi-User Session Tool

The following *C* program represents a multi-user session tool built with the proposed dynamic group primitives. It is similar to an N-party talk utility [Hughes88] with additional session information provided. It can be easily expanded into a multi-N-party talk utility, with a user participating simultaneously in several group talks.

```
main(argc, argv)
int   argc;
char  *argv[];
{
char chair_name[30], *group_name, tmp_group_name[30];
char command[80], message[80], *message_ptr, *uname, whom[30];
GROUP_LIST *group_l;
```

```
MESSAGE *ml;

printf ( "Group Name:");
     /* printf: a C-library function to write to standard output */
scanf ( "%s", tmp_group_name );
     /* scanf: a C-library function to read from standard input */

if( strcmp(argv[1], "-c") ) {          /* the chairman */
        /* strcmp: a C-library function to compare two character strings */
  create_group ( tmp_group_name, "c", "wahab pc" );  /* create a closed group */
  strcat( tmp_group_name, ":" );
     /* strcat: a C-library function to concatenate two character strings */
  group_name = strcat( tmp_group_name, getlogin() );
                /* in the following, group_name is specified with the creator's name */
  }
else {                 /* a participant */
  printf ( "who is the chairman ?" );
  scanf ( "%s", chair_name );
  strcat( tmp_group_name, ":" );
  group_name = strcat( tmp_group_name, chair_name );
  join_group ( group_name, BLOCKED, "" );
     /* join the specified group, block if not exist, time_out need not be specified */
  uname = getlogin ( );          /* get user login name */
  message_ptr = strcat(uname, " joining the session");
  send ( "", group_name, message_ptr, strlen(message_ptr) );
        /* inform the whole group "I am joining" */
     /* strlen: a C-library function to return the length of a string */
  }

while (TRUE) {          /* loop forever until exit */
  if( (ml = receive ( "", group_name, message, 80, NON_BLOCKED, "" )) != NULL)
     /* receive from any member, time_out is not specified */
    printf ( "Messages from %s : %s", ml->from_user, message );
  scanf ( "%s", command );

  if (strcmp (command, "list") == 0) {   /* display session information */
    group_l = list_group ( group_name );
    printf("mode:%c users:%s active users:%s", group_l->mode, group_l->p_list,
    group_l->ap_list);
    } else
```

```
  if (strcmp (command, "talk") == 0) {          /* address to the whole group */
    printf ( "message:" );
    scanf ( "%s", message );
    send ( "", group_name, message, strlen(message) );
    } else

  if (strcmp (command, "chat") == 0) {          /* chat with someone */
    printf ( "with whom? " );
    scanf ( "%s", whom );
    printf ( "message:" );
    scanf ( "%s", message );
    send ( whom, group_name, message, strlen(message) );
    } else

  if (strcmp (command, "exit") == 0) {
    leave_group ( group_name );
    exit();
    }
}               /* end while loop */
}         /* end main */
```

Example 3.2 — Conversion of a Single-User Tool into a Multi-User Tool

The following is a multi-user tool built by the conversion of a single-user editor. The first *main* is the chairman's program. The second *main* is the user agent that a joining user executes. Unlike the previous example, the chairman agent program and user agent program are separated because of their significant difference.

```
main()      /* chairman's agent */
{
char *group_name, tmp_group_name[30], tool[30], object[30];
char *enter_message = "enter WYSIWIS mode, ^C to quit, ^G to get the token, ^R to release";

printf ( "Group Name:");
scanf ( "%s", tmp_group_name );
create_group ( tmp_group_name, "p", "" );      /* create a public group */
strcat( tmp_group_name, ":" );
group_name = strcat( tmp_group_name, getlogin() );
```

```
/* enter wysiwis and run a single-user tool, which is converted for multi-user cooperation */
printf( "Tool Name" );
scanf( "%s", tool );
printf( "Object Name" );
scanf( "%s", object );

wysiwis ( group_name, "^C", "^G", "^R", 20, 10 );
  /* "^C" to leave WYSIWIS, "^G" to request the token, "^R" to release the token */
       /* Quantum : 20 seconds; Grace_period : 10 seconds */
send ( "", group_name, enter_message, strlen(enter_message) );
  /* inform other participants that I am in wysiwis */
execlp ( tool, tool, object, NULL );          /* execute the tool */
exit();
}          /* end main */


main()          /* user agent */
{
WYSIWIS_LIST *w_info;
char chair_name[30], command[80], *group_name, tmp_group_name[30];
MESSAGE *ml;

printf ( "Group Name:");
scanf ( "%s", tmp_group_name );
strcat( tmp_group_name, ":" );
printf ( "who is the chairman ?" );
scanf ( "%s", chair_name );
group_name = strcat( tmp_group_name, chair_name );

join_group ( group_name, BLOCKED, "" );

while(TRUE){
  if( (ml = receive ( "", group_name, message, 80, NON_BLOCKED, "" )) != NULL)
     /* receive from any member */
    printf ( "Messages from %s : %s", ml->from_user, message );

  printf("Command:" );
  scanf ( "%s", command );

  if ( strcmp (command, "list_wysiwis") == 0 ) {   /* display wysiwis information */
    w_info = list_wysiwis ( group_name );
```

```
    printf("%s in WYSIWIS, %c to quit, %c to get token, %c to release", w_info->creator_name,
          w_info->quit_signal, w_info->get_signal, w_info->release_signal);
    } else
  if ( strcmp (command, "enter_wysiwis") == 0 ) {
    enter_wysiwis( group_name, w_info->id );
          /* Assume only one wysiwis, now users cooperate through sharing a tool */
    } else
  if ( strcmp (command, "exit") == 0 ) {
    leave_group( group_name );
    exit();
    }
  }          /* end while */
}            /* end main */
```

3.3. Possible Extensions and Discussion

Heterogeneous Distributed Cluster

How do we extend the proposed dynamic group to a cluster of machines running different operating systems? The following are issues that need to be dealt with. What communication protocol is to be used? A possible candidate is the Internet protocol [Postel81, Cerf83]. What session protocol is to be used? The protocol should cover most session routines. What presentation protocol is to be used? It should define the shared textual/graphic workspaces details (e.g. virtual terminal, height and width of a window, data formats of process input/output, fonts, and bitmap resolution). What naming scheme is to be used? Names may have to be translated across machines.

Cluster Creation and Maintenance

As mentioned earlier, a cluster needs to be formed for the dynamic group mechanism to be used within a distributed system. Creation and maintenance of a cluster can be done by the system staff without system support. Alternatively, a set of privileged function calls can be provided to them, e.g. *create_cluster, delete_cluster, join_cluster*, and *leave_cluster*. Interesting issues that need to be resolved are: what cluster naming scheme should be used? How should a cluster respond to errors or network problems? How should the cluster information be maintained? Chapter 6 discusses some of the implementation issues.

Dynamic Group Users

It may be useful to allow a new user to be added to the *participants_list* after a group is created or to allow a user to be deleted from the *participants_list*. This makes a group more dynamic. It may also be useful to allow an active user to be removed from a group, when the performance of that user's machine (or the user!) becomes harmful to a session. The group can either ask him to withdraw or dismiss him from the session. For the latter, a primitive needs to be provided.

Dynamic Group Modes

Three group modes are provided: public, closed, and secret. Public mode allows users within the same machine or cluster to join. It may be useful to consider other modes, such as *universe* (everyone from the network can join). Also a useful primitive may be to allow a group to change mode, e.g. from closed to public.

Permanent Dynamic Groups

Permanent dynamic groups may be allowed to exist. This is useful because it can subsume the functions of a bulletin board or a newsgroup, with additional real-time conference functions. A permanent group is no longer destroyed when the last participant leaves.

Token Control Signals

Get_signal is used to get the token; *release_signal* is used to release the token or cancel a request. A case when no token signals need to be specified is to get/release the token through mouse movement. A user gets the token whenever he is the first to move his mouse into the **wysiwis** window focus. His token is released when he moves his mouse out of the **wysiwis** window. If an implicit token scheme is adopted, then users do not have to specify the foregoing signals. An implicit token scheme can be done through human coordination, e.g. through handshaking over the phones.

To simplify our design, there are only two token signals, but others may be considered: queue_signal (who is in the queue), grab_signal (grab the token), etc. Instead of letting a user specify these signals in the **wysiwis** system call, the terminal I/O control system routines (e.g. in *UNIX*: stty) may be modified to allow users to define these signals if the mechanism is to be supported from the operating-system level. The conflict of token control signals and tool commands can be alleviated by having a separate window for each. The window for the token control signals can be smaller and used also for displaying the token status messages. To enter **wysiwis**, users are required to have the identical window size. The system is responsible for adjusting their window sizes to that of the creator's and for restoring them when they leave this mode.

Token Status

The system may need to provide a status line to show any message that informs users of the token status. If there is a separate system status window (e.g. console display window), it can be used for displaying the token status. Otherwise, the cursor shape can be changed when the token is received. Another alternative when using a terminal with meager screen space is to have a token status message displayed temporarily for a short period. The terminal bell can also be used to reflect status change of the token.

Wysiwis can be relaxed so that no token is imposed: users are free to make their input at any time. This can be useful in some applications, e.g. a brainstorming tool.

Terminal Characteristics

Another problem has to do with achieving **wysiwis** when users are collaborating with different kinds of terminals. As different terminals have different interpretations for the escape sequences, the replicated output from the **wysiwis**-creator needs translations. To overcome this, a *virtual terminal* [Tanenbaum88] is introduced. A virtual terminal is an abstract representation of a real terminal, with various abstract operations. The operations may include writing text on the virtual screen, reading text from the keyboard, etc. In the **wysiwis** mode, the tool output will be produced according to a virtual terminal protocol. Each user's window, created the same size, will be associated with the same virtual terminal. Examples of virtual terminals are: *xterm* in the MIT *X* window package for the bit-mapped terminals [Scheifler86], Stanford's *VGTP* virtual graphics terminals [Lantz84] or a network virtual terminal [Tanenbaum88].

Secret Sessions

Most operating systems will release the information of *who* is currently logged on, and what processes are running on the system. By some reasoning and guesses, a user will be able to know who is involved in a secret session. The objects or tools they are currently using may also be inferred from system status commands. To achieve secret dynamic groups, such information should not be disclosed. Authentication of a joining user is required, to prevent a malicious user from joining with a false identifier. Authentication of messages received is also required, to prevent a malicious user from jamming messages into a session. Information flow among the session components needs to be encrypted. Connections among the session components need to be robust, e.g. it should be secure from jamming or tampering.

Closing or Destroying a Dynamic Group

Closing a group is like closing the conference door. Should every joining user be granted this capability? Or only the creator? Destroying a group is useful when a shutdown is imminent, or an error occurs and the superuser wants to delete a dangling group (explained in Sec. 6.2.3), or a permanent group is to be removed.

CHAPTER 4

MULTI-USER PROCESSES AND SHARED CAPABILITIES-LISTS

With the investigation of Sec. 2.7, the following two chapters present operating-system level solutions for real-time shared workspace cooperation. In this chapter, two mechanisms are proposed: *multi-user processes* and *shared capabilities-lists*.

4.1. Multi-User Processes

The multi-user process mechanism is a solution to multi-user tool development and sharing of user privileges in real-time cooperation, the requirements having been described in Secs. 2.7.2 and 2.7.4. General concepts and the system call interface are first described. An example is provided to demonstrate the design of a joint-browsing tool using the multi-user process. Design issues and alternatives are then discussed. Section 6.3 discusses implementation details and contains a complete list of the proposed multi-user process interface.

4.1.1. Design Concepts and Functional Interface

Traditionally, a process is associated with a single user. For real-time cooperation, *multi-user process* is proposed for *access control list systems* and *systems with mixed strategy* [Saltzer75]. A system with mixed strategy is a system where an access control list is used for the secondary storage or file system, while a capability scheme is used for the rest; the capabilities cannot be copied into the file system [Saltzer75].

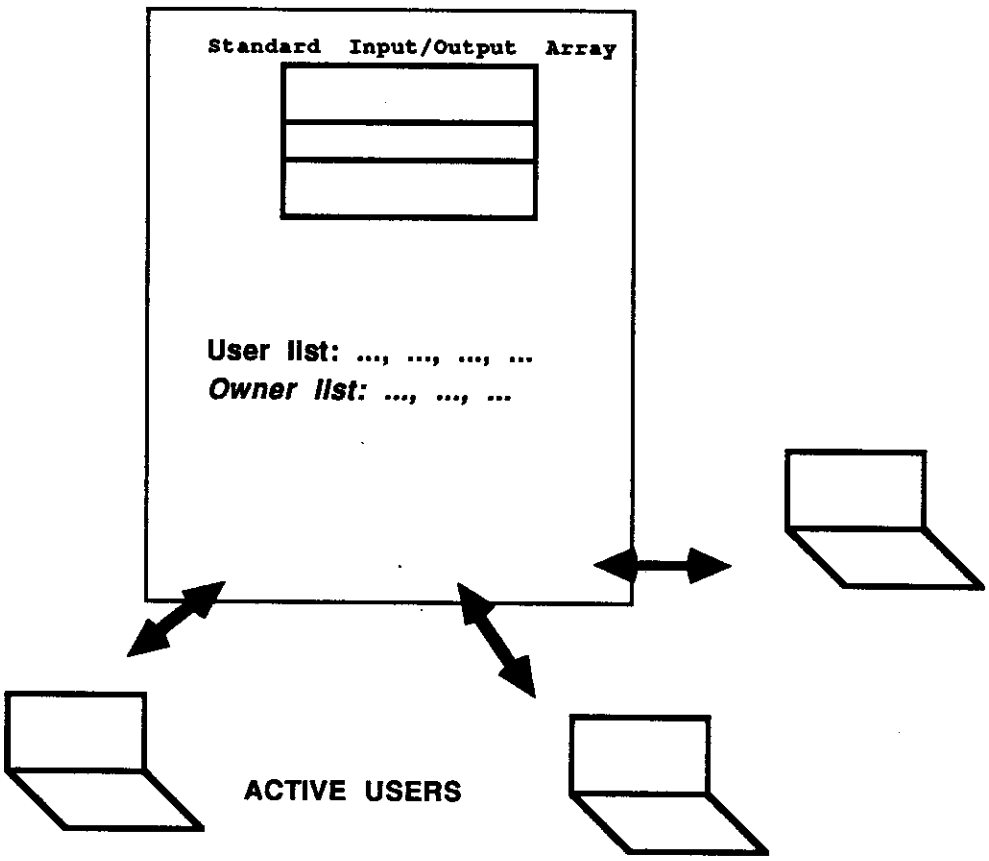


Fig. 4.1 Multi-User Process

A process runs initially with its creator as the *owner* (against whom the protection check is made). The creator makes the process multi-user by issuing **allow_join**: a nickname and a list of users who may join are specified (*Fig. 4.1*). The creator then issues **wait_join** when ready to accept participants to join. When a process issues **join_proc**, if the issuing user is one on the list, the process is suspended. A standard input/output/error channel is created in the multi-user process for this user, whose terminal becomes connected to the channels. The user becomes *active* in the multi-user process. The multi-user process can read the input of the joining user by reading from his standard input channel and can write output into his standard output channel. An active user leaves the multi-user process by issuing an exit control signal from his terminal or when the process executes *exit*. The process can be killed by an active user with some special control signal.

allow_join *(nickname, users_list)*

```
char *nickname;
char *users_list;
```

If a multi-user process with the same name and created by the same creator already exists, then it returns ERROR. The creator is an assumed participant whose name need not be specified in the *users_list*. If *users_list* is not specified, then any eligible user on this machine can join.

wait_join *(flag, time_out, joint_user)*

```
int flag, time_out;
JOIN_INFO *joint_user;

typedef struct j_user {
  char *username;
  int in, out, err;
  } JOIN_INFO;
```

Wait_join waits for one user (process) at a time to join. A program can be coded with a simple loop so that **wait_join** is executed several times until all the users on the *users_list* join. *Flag* can be *BLOCKED* (the system call will be blocked if no user issued **join_proc**; it will return when the process of a user on the specified *users_list* issues **join_proc**), *NON_BLOCKED* (the system call will return if no user issued **join_proc**) or *TIMED_BLOCK* (the system call will be blocked if no user issued **join_proc**; it will wait until either the specified *time_out* expires or the process of a user on the specified *users_list* issues **join_proc**). With this call, the multi-user process is ready to accept participants. The returned information *joint_user* includes the name of

the joining user and the created standard input/output/error descriptors for his terminal.

join_proc *(nickname, creator_name, ifnotexist, time_out)*

char *nickname, *creator_name;
int ifnotexist, time_out;

The naming requirement of a multi-user process is the same as that of dynamic groups. To avoid naming conflicts, a joining participant is required to specify the name of the multi-user process creator in addition to the *nickname*. The flag *ifnotexist* can be *BLOCKED* (the system call will be blocked if the multi-user process does not exist or is not ready to accept participants; it will return when the multi-user process issues **wait_join**), *NON_BLOCKED* (the system call will return if the multi-user process does not exist or is not ready to accept participants) or *TIMED_BLOCK* (the system call will be blocked if the multi-user process does not exist or is not ready to accept participants; it waits until either the specified *time_out* expires or the multi-user process issues **wait_join**).

A multi-user process is jointly owned by its active users, i.e. the participants. The process has its active users as joint-owners, against whom the protection check is made. If an active user leaves by issuing an exit control signal, then that user loses his ownership to the multi-user process and his original process is resumed. An object or a process created during the execution of the multi-user process will generally be owned by the joint-owners (see Ch. 5).

A shared workspace is achieved this way: a joining user's resource can be shared whenever the process opens it or acquires a capability for it. Simultaneous manipulation of objects across multiple user domains (e.g. simultaneous opening of objects under different users' domains by a process) is possible because the process runs under the union of multiple user domains.

The *departing workspace-object-owner problem* (Sec. 2.7.4) is solved by having a departing user leave behind capabilities of his objects so that others can continue working on his objects. There is no *departing chairman problem* (Sec. 2.7.4) because when the creator of a multi-user process leaves, the process continues with the remaining users.

Note that having multiple standard output channels in a multi-user process does not imply WYSIWIS. Shared viewing is achieved by writing simultaneously to the standard output channels of participants in the program. The decision whether or not a token is imposed to control users' input is also left to the participants. If they judge that close coordination is required then the program is implemented with a token control scheme.

4.1.2. An Example

Example 4.1 — Joint-Browsing Tool

The following is a multi-user process formed to access files across two users' domains simultaneously. It shows how a multi-user tool can be written with the multi-user process system calls and how users can share their objects dynamically. It is a two-party joint-browsing tool that opens a file under one user's domain, and presents the file contents buffer by buffer simultaneously to two participants. After viewing over a buffer, each participant acknowledges by striking a key when he is ready. The buffer is then written to a file under another user's domain, and the next buffer is presented. The program can be easily generalized to the N-party case.

It is assumed that the following system calls exist: *read, write, open,* and *close*; the protection checking of these calls has been also changed to incorporate the multi-user process mechanism. *FD_ZERO, FD_SET, FD_ISSET* and *select* are *4.3BSD UNIX* interprocess communication primitives [Leffler86].

```
#define N 2     /* 2-party */

/* global data */
JOIN_INFO joint_user;
int in[N], out[N], err[N];          /* standard input/output/error channel descriptor array */
int f1, f2, i, rc, nb, ack[N];
char buf[80], ackbuf[10];
char multi_upname[30];              /* multi-user process nickname array */
char whom[30];
char file_name1[50];          /* name of the file to be read */
char file_name2[50];          /* name of the file to be written */
fd_set read_template;               /* bit array used for 'selecting' user input */
struct timeval  wait;         /* time-out variable for 'select' */

main()        /* multi-user process creator's agent */
{
in[0] = 0;
out[0] = 1;
err[0] = 2;

printf("\n Name of the multi-user process?");
        /* printf is addressed to the standard output descriptor 1 */
scanf("%s", multi_upname);
        /* scanf is addressed to the standard input descriptor 0 */
```

```
printf("\n Who is joining?");
scanf("%s", whom);
allow_join ( multi_upname, whom );

wait_user();                /* subroutine */
request_filenames();                /* subroutine */
f1 = open ( file_name1 , O_RDONLY );
f2 = open ( file_name2 , O_WRONLY );

while( (rc = read( f1,  buf, sizeof(buf) )) > 0 ) {   /* read while not EOF */
     FD_ZERO ( &read_template );
                /* a macro call that clears a bit-array */
     for (i = 0; i < N; i++) {
       if( in[i]  >=  0)              /* if this channel is open */
         ack[i] = FALSE;
       else ack[i] = TRUE;
     }
     wait_acks();       /* subroutine: wait for users' acknowledgements  */
        /* now all participants have acknowledged */
     for (i = 0;i < N; i++) {       /* display another buffer of output */
     if(out[i] > 0){            /* if this channel is open */
        if(write ( out[i], buf, rc ) <=0) {  /* user leaves */
          out[i] = -1;        /* reset his standard channel array */
          in[i] = -1;
        }
     }
     write ( f2, buf, rc );          /* write this buffer into another file */
     }      /* end for */
}           /* end while */
cleanup();
}                   /* end main */


wait_user()                 /* subroutine: wait for user(s) to join */
{
char *join_message = " joining the process";

for (i = 1;i < N; i++) {            /* waiting for each participant to join */
     wait_join ( BLOCKED, "", &joint_user );            /* time-out not specified */
     in[i] = joint_user.in;                  /* joining user's stdin, stdout, stderr descriptors */
     out[i] = joint_user.out;
```

```
    err[i] = joint_user.err;
    }
for (i = 0;i < N; i++) {   /* inform participants that a new user is joining */
    write ( out[i], joint_user.username, strlen(joint_user.username) );
    write ( out[i], join_message, strlen(join_message) );
    }
}                   /* end wait_user */


request_filenames()
        /* subroutine: request file names from the creator, let the other(s) know too */
{
char *view_message = "\n Name of the file to be viewed?";
char *write_message = "\n Name of the file to be written?";

for (i = 0;i < N; i++) {
    write ( out[i], view_message, strlen(view_message) );
    }           /* end for */
    rc = 80;      /* assume 80 is the maximum name length */
    nb = read (in[0], file_name1, rc);
  /* Assume the creator furnishes the name of a file of his own, say: "/unc/guan/file1" */
    file_name1[nb] = "\0";
for (i = 1;i < N; i++) {
    write ( out[i], file_name1, strlen(file_name1) );
    }           /* end for */

for (i = 0;i < N; i++) {
    write ( out[i], write_message, strlen(write_message) );
    }           /* end for */
    rc = 80;      /* assume 80 is the maximum name length */
    nb = read (in[0], file_name2, rc);
    /* Assume the creator furnishes the name of a participant's file, say: "/unc/chen/file2" */
    file_name2[nb] = "\0";
for (i = 1;i < N; i++) {
    write ( out[i], file_name2, strlen(file_name2) );
    }                           /* end for */
}                   /* end request_filenames */


wait_acks()             /* subroutine: wait until all users acknowledge */
{
```

```
wait.tv_sec = 5;
wait.tv_usec = 0;
do {
      for (i = 0; i < N; i++) {
        if( in[i]  >= 0 )
          FD_SET ( in[i] , &read_template );
              /* FD_SET: a macro call that sets a bit in a bit-array */
        }    /* end for */

      nb = select ( FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait );
 /* select: examines the I/O descriptor set to see whether some of them are ready to be read */
      for ( i = 0; i < N; i++ ) {
        if (FD_ISSET ( in[i], &read_template )) {
                    /* a macro call that tests whether a bit is set in a bit-array */
          if (read ( in[i], ackbuf, sizeof(ackbuf) ) <=0) {  /* the channel is closed */
             in[i] = -1;
             }
          ack[i] = TRUE;
          }
        }          /* end for */
      } while (!(ack [0] && ack [1])) ;              /* end do */
}                  /* end wait_acks */


cleanup()                  /* subroutine: clean up */
{
for (i = 0; i < N; i++) {
      close ( in[i] );
      close ( out[i] );
      close ( err[i] );
      }
close( f1 );
close( f2 );
}                  /* end cleanup */



/* The following code is for a joining participant. */
main()
{
int rc;
```

```
char multi_upname[30], whom[30];
char *leave_message = "Leaving the multi-user process ... \n";

printf( "\n Name of the multi-user process?");
scanf( "%s", multi_upname);
printf( "\nJoin whom?");
scanf( "%s", whom);
join_proc ( multi_upname, whom, BLOCKED );
write ( 1, leave_message, strlen(leave_message) );
}                    /* end main */
```

4.1.3. Design Alternatives and Discussion

Multi-User-Threaded Task

Instead of letting multiple users share a single process, a *multi-user-threaded task* could have been provided so that each user would have his own thread. Here a *thread* can be thought of as a light-weight process. The multi-user-threaded task (*Fig. 4.2*) is similar to the *multi-threaded task* [Accetta85, Rashid86] except that each thread is associated with a different user. The thread runs with that user's privilege. All user threads within the same multi-user-threaded task share the same virtual address space, including capabilities-lists. A shared workspace is achieved by envisioning the multi-user-threaded task as providing an environment, resources inside which are available to the participants. A participant can bring to the environment any object he wants to share. This is done by sharing his capability to the object. He may also disable sharing by withdrawing the capability any time he wishes.

The *departing workspace-object-owner problem* (Sce. 2.7.4) is solved when a departing user leaves behind capabilities for his objects so that the others can continue working on them. This multi-user-threaded task provides sharing of different user domain objects under the same computation, useful in real-time cooperation. Any conflict of using the resource is coordinated within the environment, e.g. by locking or implementing a monitor [Hoare74] within the multi-user-threaded task. Simultaneous manipulation of objects across multiple user domains is possible because each thread shares resources in the same task. An initial design has been laid out in [Guan88]; further effort is needed to study this mechanism.

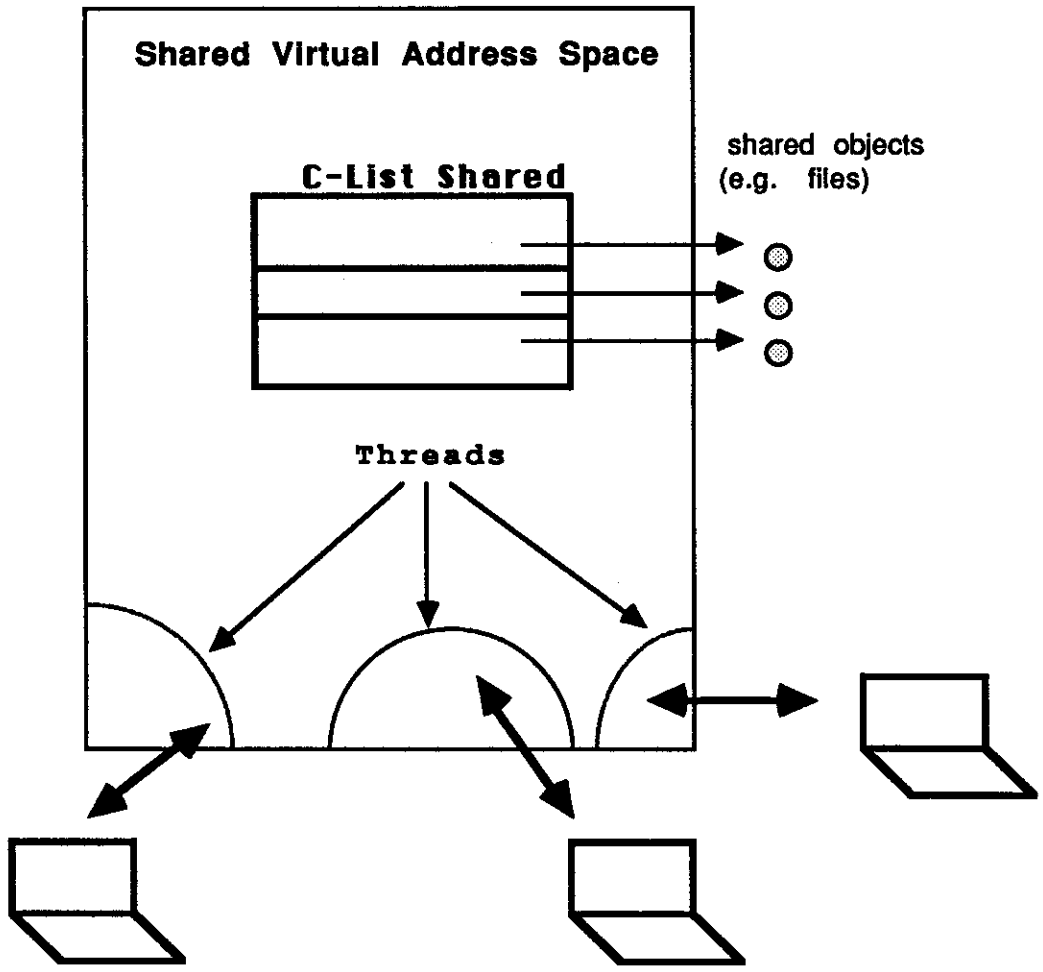


Fig. 4.2 Multi-User-Threaded Task

Group_Owned Process

Another design choice is the following. The multi-user process is run with the union of the active participants' privileges. Under some circumstances, this may be undesirable. One example is a group of slightly-untrusted cooperating users who cooperate through a tool process (*Fig. 2.3*), but none would like the tool process run under his domain. This may happen either because the accounting will be made singly to the user who runs the tool process when actually this is a group job or because the user who runs the tool process would not like the other participants to abuse his privilege. What mechanism supports such a requirement?

Most existing operating systems do not provide this capability. For example in *UNIX*, a process can be run with a certain GID (group identifier), but the UID (user identifier) comes into effect first. So either accounting or protection will be made against the user who runs the process for the group. A mechanism is needed to run a process with group privilege. Further effort is required to study this mechanism.

Naming and Creation of Multi-User Processes

Similar multi-user process naming conventions have been adopted as for the dynamic groups (see Sec. 3.1). Alternatively, the system can assign an identifier when a multi-user process is created. Then the creator communicates it to other participants through some real-time or pre-established channels.

The proposed system call **allow_join** is one way of creating a multi-user process: by granting ownership of a process to other users. Another way of creating a multi-user process is to allow a process to be jointly created. When a multi-user process spawns or creates another process, the newly created process is jointly owned by owners of the original multi-user process.

Joining a Multi-User Process

Wait_join and **join_proc** use a synchronous rendezvous protocol. It is possible to use an asynchronous rendezvous protocol, with which a user joins a multi-user process without the process waiting for him (i.e. **wait_join**). Similarly in the real world, a user may join a conference through some receptionist, or he may join without any reception. With the latter approach, some mechanism needs to be developed for a multi-user process to know its current participants and their standard channel descriptors.

Wait_join allows waiting for one user at a time. An alternative is to wait for several users at a time, i.e. the system call returns until the number of **join_proc** reaches the number the creator specified. The returned information should include all attaching users' information. Each corresponding **join_proc** will return only when **wait_join** returns.

Suspending a Multi-User Process

If the participants of a multi-user process cannot finish their work within a certain length of time, they may want to suspend (stop) the multi-user process temporarily. This can be done by one participant issuing some control signal from his terminal. Later when the participants decide to resume their work, each participant may do so by informing each other and issuing a resume command with the corresponding multi-user process identifier (or *nickname* + *creator_name*). If any participant decides not to resume, then the resuming procedure will fail (after a time_out period expires).

4.2. Shared Capabilities-Lists

This section presents the functional design of shared capabilities-lists (C-lists). They provide a solution to sharing user privileges in real-time cooperation, the requirement having been described in Sec. 2.7.4. General concepts and the system call interface are first described. An example is provided to demonstrate the use of a shared C-list to achieve a shared workspace in a session. Design issues and extensions are then discussed. Section 6.4 discusses the implementation details and contains a complete list of the proposed shared-clist interface.

4.2.1. Design Concepts and Functional Interface

Achieving a fully shared workspace through a multi-user process can be useful for most applications, but if users want finer control of sharing as stated in the *shared workspace problem* in Sec. 2.7.4, then the following solution is proposed. It is intended for capability systems and systems with mixed access strategy, with capabilities kept in the kernel space. Descriptors or indexes are used to reference the capabilities. Each process runs under its own address spaces and has its own capabilities-list (C-list). Every C-list is assumed to have the same number of entries, including empty ones. A process can create a *shared C-list*, returning a *key* for further access (*Fig. 4.3*). To share the shared C-list, the process provides this key to other processes. Keys should be sparse and difficult to guess. The key is verified for each access to see if it points to a valid shared C-list.

double **create_clist** *(count)*

int count;

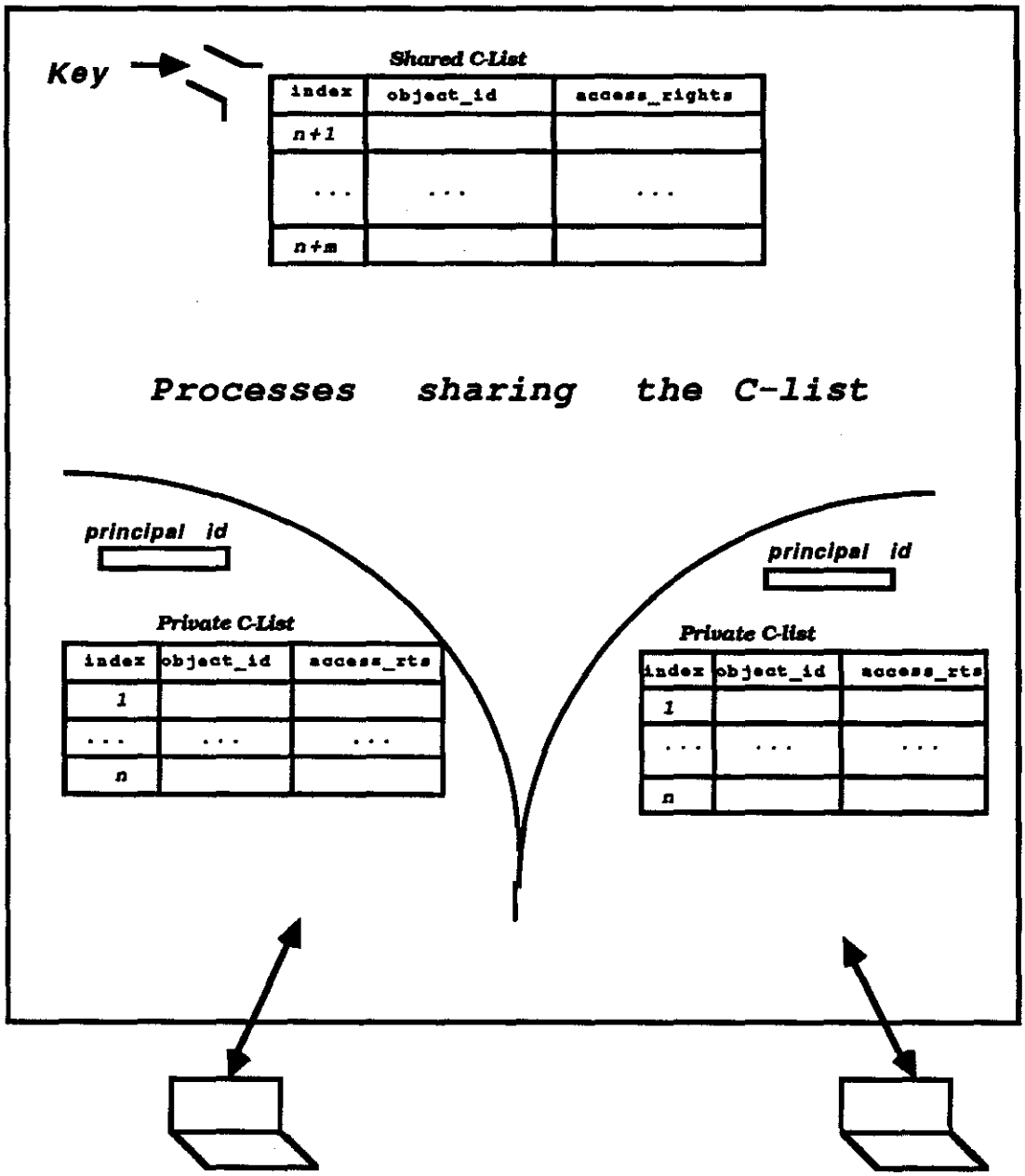


Fig. 4.3 Shared C-List

Count specifies the number of entries to be created for the shared C-list. The range of descriptors for each shared C-list is disjoint from the set of private capability descriptors held by a process (*Fig. 4.3*). This allows the system to distinguish easily a private capability descriptor from a shared one.

To use a shared C-list, a process presents the key to the system. A process makes public a capability of its own by issuing **put_public** or **dup_public**. The former moves the corresponding capability entry into the shared C-list; the latter copies the corresponding capability entry into the shared C-list. **Dup_public** is useful when the contributing process still needs access to the object in its own way. After a capability is placed in the shared C-list, a *shared capability descriptor* is returned for further access, and can be made known to other processes sharing this C-list. A shared C-list is removed whenever anyone who has the key issues **delete_clist**.

int **put_public** *(key, pd)*

double key;
int pd;

The *key* specifies which shared C-list is to be used. The capability indexed by the private descriptor *pd* is moved into the shared C-list. A shared capability descriptor is returned.

int **dup_public** *(key, pd)*

double key;
int pd;

The capability indexed by the private descriptor *pd* is replicated into the shared C-list. A shared capability descriptor is returned.

delete_clist *(key)*

double key;

After the *key* is validated, the shared C-list is deleted. Note here that users sharing a C-list are responsible for deleting it whenever it is not needed.

To reference an object through a shared capability descriptor, the associated key is also presented. The system calls dealing with capability descriptors (e.g. read, write, close) need to be extended to handle the shared capability descriptors:

_read *(fd, buffer, length [, key])*

int fd, length; char *buffer; double key;

_write *(fd, buffer, length [, key])*
int fd, length; char *buffer; double key;

_close *(fd [, key])*
int fd; double key;

These extended system calls check the specified capability descriptor *fd*. If it is a shared one, a key is required. If it is a private one, a key is not needed and the processing proceeds as before.

Shared C-lists can be used with the dynamic group mechanism or other interprocess communication facility (e.g. BSD 4.3 IPC) so that a process after creating a shared C-list sends the key to participants that it wants to have share its objects. A user has freedom in choosing partners in a session. By creating different shared C-lists, a process may share with different processes different shared C-lists. For a capability made available in a shared C-list, a user may let other participants know by sending the returned shared capability descriptor to their processes through some channels. A concurrency control scheme like locking or a monitor is needed to avoid conflicts using these shared objects.

Unlike passing a capability directly, the proposed shared C-list does not have the difficulty revoking capabilities, i.e. canceling granted capabilities, because the capabilities are kept by the operating system. A shared capability, once granted into the shared C-list, can be removed by closing the shared capability descriptor. Allowing greater access rights (e.g. from read-only to read/write) to a shared workspace object can be done by closing the shared capability descriptor first, then reopening and posting it with new access right. Reducing an access right (e.g. from read-write to read-only) to a shared workspace object can be done in a similar manner. Using shared C-lists allows user processes to share capabilities in a dynamic way.

With shared C-lists, it is possible for users to achieve shared memory or data structure by sharing a capability to a data segment. A shared workspace is achieved in a flexible manner. The *departing workspace-object-owner problem* (Sec. 2.7.4) is solved, because a departing user can leave behind capabilities to his objects so that the others may continue working on his objects. Simultaneous manipulation of objects across multiple user domains is possible because a shared C-list can have capabilities to objects across multiple user domains. The protection domain of each participating process sharing a C-list is the union of domains referenced through its private capabilities and the capabilities in its shared C-lists.

4.2.2. An Example

Example 4.2 — Session Tool Using a Shared C-list

This example shows how the shared C-list system calls are used together with the dynamic group mechanism to achieve a shared workspace. The following two agent programs, used by the chairman and another user, cooperate by forming a 2-party session and sharing a C-list. Each user opens a file, deposits the capability into the shared C-list, and reads the other file.

Assume system calls open and exit exist. Note here *ntohs* and *htons* belong to the *4.3BSD UNIX IPC* mechanism [Leffler86].

```
char messag[30] = "this IS a test\n";

main()        /* chairman's agent */
{
char response[80], message[80], whom[50], buf[80], gr_nm[80], *group_name;
char *g_name = "rsw";
char key_str[20], key_string[10];
char fd_str[20], fd_string[10];
MESSAGE *ml;
int agenda, pub_agenda;
int priv_fd, rc, key;

/* The session chairman opens the 'agenda' file for read access, creates a shared C-list, posts
the agenda descriptor to the shared C-list, passes the key and shared agenda descriptor to the
other participant. */

agenda = open ( "agenda", O_RDONLY );
key = create_clist( 5 );
pub_agenda = put_public ( key, agenda );

printf("\nSession with whom?");/* a 2-party session */
scanf("%s", whom);                    /* name of the other participant */
strcat(whom, getlogin());              /* form the participant list */
                                  /* getlogin: get my login name */

create_group ( g_name, "c", whom );   /* create a closed group */
group_name = strcat ( g_name, ":" );
                  /* concatenate the group_name with the chairman's name  */
```

```
strcpy ( gr_nm, group_name );
group_name = strcat ( gr_nm, getlogin() );

if( (ml = receive ( whom, group_name, message, 80, BLOCKED,"" )) != NULL)
{
  printf("%s\n", message);
  sprintf( key_string, "%d", htons(key) );
        /* htons: convert host byte order to network byte order */
  /* sprintf translates "htons(key)" according to the format "%d" and places the output,
     followed by the null character (\0), in consecutive bytes starting at "key_string" */

  send ( whom, group_name, key_string, strlen(key_string) );
     /* share the key with the group */
  sprintf( fd_string, "%d", htons(pub_agenda) );
  send ( whom, group_name, fd_string, strlen(fd_string) );
     /* send the agenda shared file descriptor to the group */

  if( (ml = receive ( whom, group_name, message, 80, BLOCKED,"") ) != NULL)
  {
/* waiting for his peer to join, receive a shared file descriptor to his peer's private file. */
     sscanf( message, "%s", &fd_str );
      /* sscanf: read from the character string "message", interpret it according to
                 format "%s", and store the results in fd_str */
     priv_fd = ntohs( atoi(fd_str) );
        /* ntohs: convert network byte order to host byte order */
           /* atoi: ASCII to integer conversion */
  }

  while( (rc = _read( priv_fd, buf, sizeof(buf), key )) > 0 ) {
        /* _read: extended 'read' system call */
    _write( 1, buf, rc );
        /* _write: extended 'write' system call */
    _write( priv_fd, messag, strlen(messag), key );
    }

  printf("Type any character to quit:");
  scanf ( "%s", &response );
  _close( pub_agenda, key );
        /* _close: extended 'close' system call */
  delete_clist ( key );
  leave_group ( group_name );
```

```
  exit();
  }
}
```

The following code is for a participant who shares his private file with the chairman for READ/WRITE access.

```
main()      /* user agent */
{
char chair_name[30], response[80], *group_name, message[80], buf[80];
char *g_name = "rsw:";
char *join_message = "Joining the session.\n";
char priv_string[10], key_str[20], agen_fd_str[20];
MESSAGE *ml;
int  private;              /* file descriptor for "private" file */
int pub_private;      /* shared file descriptor for "private" file */
int key, agenda_fd, rc;

private  = open ( "private", O_RDWR );
printf ( "\nShare with whom?" );
scanf ( "%s", &chair_name );
group_name = strcat ( g_name, chair_name );

if( join_group ( group_name, BLOCKED, "" )>=0) {
  send ( chair_name, group_name, join_message, strlen(join_message) );
  if( (ml = receive ( chair_name, group_name, message, 80, BLOCKED,"" )) != NULL)
  {
     sscanf( message, "%s", &key_str );
     key = ntohs( atoi(key_str) );
  }
  pub_private = put_public ( key, private );
  sprintf( priv_string,  "%d", htons(pub_private) );
  send ( chair_name, group_name, priv_string, strlen(priv_string) );
  if( (ml = receive ( chair_name, group_name, message, 80, BLOCKED,"") ) != NULL)
  {
     sscanf( message, "%s", &agen_fd_str );
     agenda_fd = ntohs( atoi(agen_fd_str) );
  }
  while( (rc = _read( agenda_fd, buf, sizeof(buf), key )) > 0 ) {
```

```
    _write( 1, buf, rc );
    }

  printf( "Type any character to quit:" );
  scanf ( "%s", &response );
  _close( pub_private, key );
  leave_group ( group_name );
  exit();
  }              /* end if */
}
```

4.2.3. Possible Extensions and Discussion

The same shared C-list mechanism applies to a homogeneous distributed environment, i.e. a shared C-list can be shared across a distributed system where each machine runs the same operating system. Distributed shared C-lists are accessible within the cluster of trusted systems, e.g. a cluster of machines on a campus. Maintaining distributed shared C-lists can be done by system-servers. Each system-server maintains an identical table of shared C-lists, with address information for each shared C-list (see Sec. 6.4 for more implementation details). User processes sharing a C-list can reside on different machines. The capability descriptors specified in the extended **_read**, **_write** or **_close** calls can reference a remote object. Remote access to a shared capability will be forwarded to and processed by the system-server where the shared capability resides. An implementation of distributed shared C-lists is sketched in Sec. 6.4.1. Implementation issues are discussed in Sec. 6.4.2.

The key should be generated by the system and that the key space should be sparse enough so that a key cannot be forged without substantial effort. As the *Data Encryption Standard* (DES, [NBS77]) uses 56-bit keys, we suggest here that a key uses at least as many bits (the double data type has 8 bytes).

It is also possible to let a user specify the key when creating a shared C-list. The case in which two identical keys are presented when two processes are creating different shared C-lists is difficult to deal with, because if the second request is rejected then the system reveals that there is a shared C-list using the same key.

As it is useful to replicate capabilities (e.g. through **dup_public**), it can also be useful to provide a mechanism to replicate a shared C-list. The replicated shared C-list is associated with a different key and has its capabilities replicated from the original shared C-list.

As capability systems can be expensive [Cohen75, Levy84], shared C-lists can be restricted to the file system only, as has been shown in the prototype implementation. Systems with mixed strategy [Saltzer75] like *UNIX* can be easily adapted with shared C-lists restricted to the file system only.

CHAPTER 5

PROTECTION MODEL FOR CONDITIONALLY JOINTLY-OWNED OBJECTS

In this chapter, Graham and Denning's protection model [Graham72] is summarized first. Jointly-owned objects are generalized to conditionally jointly-owned objects to help resolve conflicts among joint-owners. A mechanism realizing conditionally jointly-owned objects is presented, the requirements having been described in Sec. 2.7.5. Graham and Denning's protection model is extended to provide a protection basis for conditionally jointly-owned objects and subjects. A design of conditionally jointly-owned objects is specified at the system-call level in Sec. 5.6. Examples are provided in Sec. 5.7. Implementation details are discussed in Sec. 6.5.

5.1. Graham and Denning's Protection Model

Graham and Denning [Graham72] proposed a protection model based on Lampson's work [Lampson71] to permit the cooperation of mutually suspicious subsystems. Their model is summarized below. Readers are encouraged to read their original paper.

There are three components in their model: *objects, subjects* and *rules*. An *object* is an entity to which access must be controlled. A unique identifier is assigned to each object. A *subject* is an active entity whose access to objects must be controlled. A subject may create an object, and becomes the *owner* of the object. The owner right allows him to grant himself any access to his object. When a subject is being created, a *control* right is granted to him by his creator. This right allows him to read or delete rights from his protection state. Subjects are also objects, as they must be protected.

Rules control the accessing of objects by subjects. The information specifying the types of access subjects have to objects constitutes a *protection state* of the system. The protection state can be represented conceptually as an *access matrix* A, with subjects identifying the rows and objects the columns. The entry A[S, X] specifies the access rights held by subject S to object X. A *copy flag* can be associated with an

access right. If the flag is on, it permits a subject to grant to any other subject any access right he holds for an object. If the flag is off, it prevents a subject from giving away access to the object.

A *monitor* exists for each type of object; it validates all accesses to objects of that type. An access proceeds as follows:

1. S initiates access to X in manner *a*, e.g. read, write, etc.
2. The computer system supplies the triple (S, *a*, X) to the monitor of X.
3. The monitor of X interrogates the access matrix to determine whether *a* is in A[S, X]. If so, access is permitted; otherwise, it is denied.

There is an *access matrix monitor* that enforces several rules ([Graham72], Table I). For example, when a subject has owner right to an object (or subject), he may change or read the protection state of his object (or subject) in the matrix.

Graham and Denning make a restriction that each subject is owned or controlled by at most one other subject. By enforcing this, a tree hierarchy of relation "subject" is maintained. It is still possible in their model for the owner attribute of a nonsubject object to be granted, but they argued that either multiple ownership should not be provided or coordination among the joint-owners themselves needs to be done to avoid contradictory actions, e.g. one joint-owner grants access the others do not want granted.

5.2. Conditionally Jointly-Owned Objects

If multiple ownership is allowed and each owner has full right to the object jointly owned, the joint-owners need to coordinate among themselves to resolve conflicts. There is no way to prevent one owner from accessing the object abusively unless the system has some knowledge about the owners' coordination and enforces it.

In this section, a design is presented to allow multiple owners to specify some *condition* to the system. A *condition* defines one or more subsets of the set of users who have the rights to an object. It can be a quorum (e.g. at least two joint-owners must be present), an authority-list (e.g. joint-owners A and B must be present), or a feature say, "more than 60% of the number of joint-owners must be present". These objects are called *conditionally jointly-owned* (CJO) objects. The system ensures that the condition is met before the object can be accessed or its protection state can be changed.

An object's condition has two distinct parts. An *access-condition* (AC), if placed on an object, needs to be met before the owners or authorized users can access the object. A *control-condition* (CC), if placed on an object, needs to be met before the owners or authorized users can change the protection state of the object (e.g. grant an access right to another user, destroy the object). Each user (process) when making an

access or changing the protection state of an object needs to inform the system if a joint action is intended (see next section "How to validate access to a CJO object" for explanations). If so, the system will wait until the required number of participants join and then verify that the condition is met. Using the access- or control-condition, the joint-owners' conflicts are resolved with their joint presence.

An access condition is useful if the joint-owners of an object want more awareness of each other's actions on the object. For example, when two users jointly open a bank safe, they are aware of each other's actions on the safe in addition to the knowledge of joint presence. An access condition can include presence constraints for read, write or execute access that require all or a majority of the joint-owners' presence.

Note that not all CJO objects have achievable conditions, i.e. conditions that can be met by qualified users. For example, if a quorum greater than the number of eligible users to an object is specified, the condition is not achievable.

A jointly-owned (JO) object is a special case of CJO objects with null control-condition. Each owner of a JO object has full ownership to the object. The access-condition of a JO object may be non-null. Obviously, a singly-owned object is a special case of a jointly-owned object.

5.3. Creation and Maintenance of Conditionally Jointly-Owned Objects

In the following, issues for creating a CJO object, committing and withdrawing joint-ownership, verifying joint access, performing joint operation, changing conditions, and deleting a CJO object are elaborated.

How is a CJO object created?

A CJO object may be created by several users jointly, e.g. through a multi-user process. These users become joint-owners of the CJO object. During the creation of the CJO object, the users specify the access- and control-condition jointly.

Alternatively, the owner of an object may grant ownership to another user, if that user accepts. In most protection systems, granting an access right needs no agreement of the grantee [Graham72]. For granting ownership in this model, it is required that the grantee agree. This is so because ownership frequently implies obligation. Sometimes a user does not want such a granted ownership; he may even be charged for disk space quota if he jointly owns an object. An object can be a contract: granting ownership is like offering a contract; accepting ownership is like signing the contract. The original owner of an object makes it jointly owned by specifying the joint-owners. The granting of ownership to a user is completed when that user accepts it, thereby becoming

committed. A uncommitted user has no owner right to the object.

Just as an access-condition may specify different presence constraints for each kind of access (e.g. read, write, or execute), a control-condition may also include different presence constraints for changing different parts of the protection state of a CJO object. For example, two co-authors collaborating on a paper may agree that each author may grant the read right to other users at will, but both authors need to agree on adding a third co-author.

There is some difficulty in requiring a user to commit before ownership is granted. The protection state of a CJO object may not even be changed until all the joint-owners commit because the control-condition is not met until then. To solve this difficulty, the *effective control-condition* (ECC) is defined as the control-condition being evaluated without considering uncommitted joint-owners. The effective control-condition is used in place of the real control-condition. Similarly, the *effective access-condition* (EAC) is defined as the access-condition evaluated without considering uncommitted joint-owners.

Can a joint-owner withdraw his ownership?

When the effective control-condition is null, a joint-owner may withdraw his ownership at will. Otherwise, the effective control-condition must be met because for some CJO objects, e.g. a contract, an owner should not withdraw at will. A withdrawing user is removed from the committed users list.

How to validate access to a CJO object?

The difficulty of validating access to a CJO object like an authority- or quorum-based object can be seen here. With computer access, users need not even gather together physically to access an object jointly. With single-user processes, it is difficult for users to provide evidence to the system that they are "together" to open the object. If each user issues *open* in his process, the requests received by the operating system are still serialized, and the system has no way to verify that users are together. The system cannot simply wait until all users have issued their requests.

Assume users with different interests collaborating in subgroups on different sections of an object. The system needs to know whether the requests issued from the users' processes are related. For example, assume a quorum-based object has four users who have read access rights, and a read-quorum equal to two. Suppose the users form two groups. The read requests from these two groups of users should not be correlated by the system because they may work on different parts of a document.

The difficulty can be solved with a multi-user process as described in Sec. 4.1. The multi-user process can be programmed to ask agreement from its participants and do the joint action for the users. The multi-user process notifies each participant of the

result of the joint action by replicating it to each participant's standard output channel. Alternatively, it can be solved in the following way: before accessing a CJO object jointly, one user process provides some information (e.g. *time_out* or the number of users to be together) to the system and asks the system to provide it a unforgeable token. It then distributes the token to its cooperating user processes that want to access this object jointly. These user processes may notify the attaching users, seek their agreement, and present the token when making their requests so that the system knows that they are together to make the access. The system waits until all expected participants make the access request (or the specified *time_out* expires). It then checks whether the effective access- or control-condition is met, e.g. if the number of users at least equals to the effective access- or control-quorum.

How is a joint-operation performed?

With a multi-user process, a joint-operation is performed in a straightforward manner. A *read* or *write* action is performed once; the result is returned to the multi-user process itself. With several processes issuing a joint-operation through a token, we have "write once, read many" operation. A write operation, whether into a file, channel, data structure, or memory, is performed only once. Thus for a joint-write operation, only one process, preferably the one which asks the system to assign a token for the joint-operation, needs to tell the system all the information needed for the write operation. For a read operation, whether from a file, channel, data structure, or memory, the result is replicated to all the participating processes. Thus all the participating processes need to provide the system consistent information regarding how the read operation is to be done (e.g. how many bytes to be read, where to store the result).

How are conditions of a CJO object changed?

The access- or control-condition of a CJO object can be changed if the issuing user(s) meet the effective control-condition.

When is a CJO object removed?

This can be done only by the joint-owners issuing a command *destroy*. The effective control-condition needs to be met. When the condition is null, this can be done by any joint-owner. Alternatively, it is removed when the last joint-owner withdraws.

5.4. Jointly-Owned Subjects

As subjects are also considered to be objects, it is natural to expect that the concept of "jointly-owned" can be applied to subjects. Does there exist a subject jointly owned in computer systems? The *multi-threaded task* [Accetta85] is an example: all

threads within a multi-threaded task execute in parallel and share the same address space and capabilities. A thread may destroy the whole task. The task can be suspended or resumed as a whole by any thread within the task. Thus the task is jointly owned by its threads.

A jointly-owned (JO) subject is defined as a subject that has several owners, each of whom has full ownership. A subject can be jointly created and owned by several owners; alternatively, an owner can grant ownership to another subject, who becomes a joint-owner if he agrees. With this extension, ownership can be granted, and the relation "owner" no longer defines a tree hierarchy. A joint-owner cannot invalidate the ownership of another joint-owner. Thus an ownership, once granted, cannot be taken back. A joint-owner (subject) may grant some of his rights to a JO subject; the conferred rights or the subject itself may be removed if another joint-owner removes it. An object that is created by a JO subject is a JO object. The notion of "conditionally jointly-owned" can be similarly applied to subjects. The results of the next section thus apply to CJO subjects also.

The multi-user process is another JO subject, where the process is jointly owned by all attaching participants. The creator of a process makes it "multi-user" by giving a list of users who may join. When a participant joins the process, he becomes a joint-owner of the process. An object or a process created during the execution of a multi-user process will generally be owned by the joint-owners. Because the multi-user process runs under the union of multiple user domains, simultaneous manipulation of objects across multiple user domains within a single process is possible.

How is it possible that a multi-user process runs with multiple user privileges? It is assumed that participants in a multi-user process will share with each other the access rights needed for object access when they **join_proc**. So, when a multi-user process accesses an object, the user(s) who have the access right grant it to the others so that they can make joint access. The grantor(s) need to have the copy flag set with their rights (i.e. they are permitted to replicate their rights) [Graham72], and the effective control-condition needs to be met. The right granted will be used only for the life of the multi-user process. We see an analogy in real-time collaboration, where we allow a participant to share access to an object. After the collaboration, the participant may no longer access the object.

In the next section, Graham and Denning's model is extended to model these jointly-owned subjects.

5.5. The Extended Model

5.5.1. Access

Associated with each CJO object in the access matrix will be two fields: access-condition (AC), and control-condition (CC). The effective access-condition (EAC) is AC evaluated without uncommitted joint-owners. The following notations are adopted in the access matrix:

owner& *; uncommitted joint-owner*
owner *; owner or committed joint-owner*

Generally, an access proceeds as follows:

1. SV initiates access to an object X in manner *a*. SV can be one or several subjects. When SV stands for several subjects, it is denoted by a vector of these subjects.
2. The system supplies the tuple (SV, *a*) to the monitor of X.
3. The monitor of X interrogates the access matrix to determine whether *a* is in A[SV, X] and the effective access-condition is satisfied. If so, access is permitted; otherwise, it is denied.

For authority-based objects, the last rule says that the effective access-authority, i.e. the access-authority members who have committed, must be present to make access. For quorum-based objects, |SV| (dimension of SV) may not be less than the effective access-quorum for users to make access.

An example: shown in the access matrix of *Fig. 1* is a CJO object X with three joint-owners B, C, D and one user E. It is an authority- and quorum-based object with an access-condition: (read-quorum = 0, write-quorum = 2, execute-quorum = 0), and a control-condition: (control-authority = B, control-quorum = 2). Since all joint-owners are committed, the effective access/control-condition is the real access/control-condition. Because read- or execute-quorum is zero, user B, C, D or E can read or execute the object individually. Any two users together are allowed to write the object. Users C and D together, although they are joint-owners and meet the control-quorum, may not change the protection state of X because the control-authority (owner B) is not present. Users B and C, or B and D together are allowed to change its protection state because the control-authority is present and the control-quorum is met.

Note here that for multiple owners J, K of an object X, their access matrix entries A[J, X] and A[K, X] may not always be identical, because an owner can delete rights (see next section) from its own entry. In *Fig. 5.1*, the access- and control-condition are stored with an object. An ideal place is to store them with the access control list of the object if there is one; otherwise they may be stored as part of the features of the object.

OBJECTS

SUBJECTS	X AC: read-quorum = 0 write-quorum = 2 execute-quorum = 0 CC: control-authority = B control-quorum = 2
B	owner read write execute
C	owner read write execute
D	owner read write execute
E	read write execute

Fig. 5.1 Extended Access Matrix