Coordination Infrastructure in Collaborative Systems

A Dissertation
Presented to
The Academic Faculty

by

W. Keith Edwards

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Computer Science

Georgia Institute of Technology
November 22, 1995

Coordination Infrastructure in Collaborative Systems

Approved:

____
Dr. John T. Stasko, Chairman

____
Dr. Prasun Dewan

____
Dr. James D. Foley

____
Dr. Scott E. Hudson

____
Dr. Colin Potts

Date Approved ______________________

Dedication

Despite claims to the contrary, a doctoral dissertation is never a solitary work. A great number of people have influenced this work, offering ideas, encouragement, and sustainment.

I owe many thanks to my Ph.D. committee for their support and suggestions, as well as for tolerating scheduling problems and short deadlines. Drs. John Stasko, Prasun Dewan, Jim Foley, Scott Hudson, and Colin Potts have all offered outstanding reviewing and advice on this work. In fact, Jim Foley originally suggested the topic of this dissertation to me as a qualifying examination question.

I've been fortunate in that I've been able to work on my research almost exclusively during graduate school. This is largely due to the generous support that Sun Microsystems Inc., has given me during my tenure here at Georgia Tech. The folks at Sun have not only offered financial support, but have also made excellent technical suggestions and offered much-needed moral support. Thanks to Rick Levenson, David Gedye, Tom Jacobs, Greg McLaughlin, and the rest of the people in the COCO and AMP groups at Sun.

Thanks also are due to my office mates who have tolerated me through the time I've been here (especially during the last few hectic weeks). The Multimedia Lab is a wonderful place to be a graduate student because of the great support and exchange of ideas that take place here. This is one of the most outstanding groups of people I've had the honor and pleasure of working with.

One Georgia Tech alum deserves special thanks: Henry Strickland was almost a mentor to me during my undergraduate stay here, and more than anyone else is probably responsible for my interest in computer science, and my decision to go to graduate school in the first place.

A very special thanks to Beth Mynatt, who has been my coworker, confidant, and friend during my graduate school years. Beth offered innumerable suggestions and worthwhile advice during the course of this research. Perhaps more importantly, she has been a constant and close friend through the years.

And finally, this work must be dedicated to my family, without whom none of this would have been possible. They have constantly supported me throughout my eleven-plus year stay at Georgia Tech, and indeed throughout my life. I am indebted to them all for their kindness and support.

Table of Contents

List of Tables

List of Illustrations

Summary

Research into collaborative software systems has recognized the importance of coordination in promoting effective interactions among collaborators. Coordination is the process of bringing groups of users together into a common action. More specifically, in the computer-supported cooperative work realm, coordination refers to the “real world” issues of collaborator location, awareness, rendezvous, and policy that must be addressed if collaborative software is to approach the fluidity of physical human interactions.

My research has focused on the development of software infrastructure to support the coordination needs of collaborative applications. In essence, my research examines how to bring contextual information about users and their activities into the mediated computer world, and the implications of having such information available to applications and other users.

Three specific areas of concern are user awareness, session management, and policy control. These areas are interrelated and share a commonality of implementation. This dissertation presents a taxonomy of information sharing in collaborative applications, and examines the three specific topics in detail. I also present an underlying model that can be used to implement the coordination goals provided by my infrastructure.

Chapter I

Introduction

What is CSCW?

Computer-supported cooperative work, or CSCW, is an emerging area of both research and commercial interest that lies at the intersection of several fields, including human-computer interaction, sociology, organizational psychology, and distributed systems theory to name a few. Broadly defined, CSCW is the body of theory and practice that is concerned with the use of computers in an enabling fashion to support and enhance the work activities of groups. Grudin [47] defines CSCW as the “field that examines how it all fits together: organizations, groups, and individual computer users; applications designed for individual users as they are used in group and organizational contexts; groupware designed to support multiple users while interacting with each person individually and adjusting to organizational contexts; and systems developed to support organizational goals that inevitably act through individuals and groups.”

We define a “collaborative application” as a system that supports the interaction among a number of people to achieve a single goal or set of goals. Collaborative applications may be synchronous (real-time interactive) or asynchronous (off-line). They may be designed for cooperative or competitive groups of users; they may be designed around replicated or distributed architectures. The central defining feature of these applications is that they are multi-user and support the process of collaboration among their participants. According to Grief [45], such applications are “software designed to take group work into account in an integral way.”

Collaborative Infrastructure

Collaborative applications are notoriously hard to build. As Jonathan Grudin says, “The design process fails because [developer’s] intuitions are poor for multi-user applications” [48]. While Grudin’s comments were directed at interface designers for collaborative systems, they are just as applicable for those building collaborative applications.

A number of features of collaborative applications account for this reputation. Many have been dealt with specifically in the literature. Some of these include:

- Multiple input streams are present. Applications must be prepared to deal with multiple event sources representing multiple users [77].
- Serialization and synchronization issues that result from multi-user input must be addressed [83].
- Many collaborative applications are distributed; that is, they are made up of cooperating processes running on different machines on a network. This model of programming is alien to most developers and introduces new concerns (such as latency and reliability) that are trivial in the non-distributed case [43][79].
- The views of the users must be consistent to a greater or lesser degree, depending on application requirements [59][105].

The category of problems represented above is architectural, since it stems from artifacts of the implementations commonly used to build collaborative systems.

In addition to the architectural issues that result from the multi-user focus of collaborative applications, there are a host of social and cultural issues which also make effective collaborative systems hard to build. In fact, based on the research literature, overcoming the social barriers to acceptance of collaborative systems may be the harder of the two problems [48]. Social issues in collaboration that are typically outside the experience and expertise of traditional developers include (but are not limited to):

- Applications must address the need for privacy. Users will be reluctant to take advantage of collaborative systems if they cannot trust that their privacy and security are not being compromised [56].
- There are benefits that can be derived from peripheral awareness of others [87].
- The problem of rendezvous: how do collaborators initially come together [77].
- Applications must deal with issues related to control over the environment. How do users specify their desires about how the system should behave in relation to other users who are potential collaborators [67]?

Many of the issues above are at odds with one another. For example the need for privacy must be weighed against the potential benefits of awareness. Problems such as these are largely the domain of sociologists, psychologists, and interface designers. The builders of computer software typically have little experience in these issues and are often left to guess about the best design choices in these areas; the result is often collaborative software which is overly-restrictive, inflexible, and does not support the fluid interactivity seen in human-to-human, non-computer mediated collaboration.

Thus, as Wexelblat [115] and others have reported, the very nature of collaborative applications makes them extraordinarily hard to construct. Collaborative systems have all of the design problems that go along with single-user, stand-alone applications (code structuring, interface design, bugs), but add a host of others. Because of the distributed and multi-user nature of these applications, the programming models familiar to most programmers are not adequate for building collaborative applications. Because of the high degree of interactivity present in most of these applications, collaborative systems typically place high demands on the runtime environment in which they are executing. Further, there is little support for addressing the social and cultural issues related to collaboration. Problems of rendezvous, privacy, awareness, and control must be addressed on a per-application basis, each time a new system is built. There is no infrastructure to leverage off of to help build these socially-based facilities.

Early in the development of a new class of software, implementation mechanisms tend to be *application-centered*. That is, solutions are implemented directly in the applications themselves. As the problem field matures, and developers acquire more experience about the needs of applications, solutions tend to become more *environment-centered*. Common application programming interfaces (APIs) and runtime facilities are developed which are shared by all applications. We have seen the trend toward more environment-centered approaches recently in the literature of software support for collaboration (see, for example, [12][19][29][39][77][89]).

Without common facilities to build atop, developers may find it difficult to address the problems inherent in constructing software in a given domain. Common, high-level APIs (whether for collaborative systems, graphical user interfaces, systems programming, or some other domain) enable applications built on those APIs to be developed more quickly, and with greater standardization and integration than would be possible otherwise. Further, APIs which "talk" to a shared runtime facility (such as a window server or an OS kernel) allow coordinated control over the applications built on those APIs: policy can be implemented at the shared entity to affect all of the clients of that entity.

In the domain of collaborative software, such facilities are called a *collaboration support environment* [18]. A collaboration support environment provides new programming paradigms and runtime support mechanisms designed to help developers who are creating collaborative applications.

Models of Information Sharing

The process of collaboration fundamentally revolves around the sharing of information: information about users, artifacts, current goals. Collaboration support environments are designed to support and enhance the ability of applications to share information easily and robustly.

Most collaboration support environments to date have focused on what might be termed the "classical" forms of information sharing that is typified by research in distributed systems. This class of sharing involves the application-internal data structures that must be shared, viewed, and updated among multiple users. The text that is shared in a multiple-user editor is an example of this type of information.

I term this form of sharing *application information sharing*, because it governs the use of application-internal data in a collaborative application. Application information sharing includes all data sharing that is essential for the application-specific aspects of the collaboration; the data being shared is the actual data of discourse (the artifacts) of the collaborative endeavor. The defining characteristic of application information sharing is that the shared information is dependent on the domain of the applications themselves. Each application may require a different specific type of shared information that is meaningful only to it.

The problems that must be addressed to implement application information sharing are essentially those enumerated above as architectural issues: distribution, replication, locking, synchronization, and view consistency, among others. A number of collaboration support environments have addressed these architectural issues and provided support for the application information sharing needs of applications. Examples of these systems include Arjuna [99][100], a system for supporting the low-level distribution needs of applications; Rendezvous [77], which provides support for sharing of views and data structures; DistEdit [57], a domain-specific tool for constructing shared editors based on the ISIS [4] framework; and SUITE [18][20], a system that provides support for several dimensions of "coupling" between replicated data structures and views, based on the SUITE user interface management system [21][22].

Very few collaboration support environments, however, have addressed the information sharing problems that must be overcome to satisfy the coordination requirements of collaborative applications, which often derive from social issues. Recall that coordination requirements in collaboration include rendezvous, privacy, policy, and awareness. While some social issues manifest themselves directly in applications (and affect, for example, the user interfaces of these applications), a large number of these social issues cut across application boundaries. The need for privacy of user information, for example, spans all of the collaborative applications in use and, more importantly for this work, the solution to problems of privacy can be addressed in a domain-independent fashion. Similarly, problems of rendezvous are not particular to any one application.

While these issues may again manifest themselves in applications (via application controls for rendezvous, for example), they have the defining characteristic that the information that is shared is not dependent on any particular application domain. Whereas the text buffers in a shared editor application will only "make sense" to that application, the information required for rendezvous, awareness, and privacy is largely independent of any particular application.

I term this form of information sharing *coordination information sharing*. Coordination information is used to facilitate the process and mechanics of collaboration itself; it is information about the collaboration, rather than information used by any one application in that collaboration. Coordination information is used to knit multiple applications into a unified (*coordinated*) fabric of collaboration, and to make collaboration applications more "situation aware" by bringing real world knowledge into the environment. It is used to organize entry into a collaborative setting, retain user state, and control cross-application policy. Coordination information includes information about users, information about conferences or sessions, and even the sharing of application-specific data as it is used by the collaboration environment (rather than the application) to facilitate collaboration.

Focus of Research

The focus of my research is the development of a collaboration support environment to support the coordination information sharing needs of collaborative applications. There are two primary motivations for this focus:

- Infrastructure for the sharing of coordination information is essential.
- The sharing of coordination information may require a *different* infrastructure than that required by application information sharing.

The first motivation results from my belief that coordination has not been sufficiently addressed by current collaboration support environments. Most systems to date have focused primarily on the application sharing needs of collaborative systems; support for coordination in these systems is typically limited to rendezvous and session management. While a great deal of research has indicated that the issues comprising coordination are essential for effective collaboration [27], the amount of systems-oriented research in the area of coordination is limited.

The second motivation results from the fact that no one has addressed support environments for coordination entirely in their own right. The information used for coordination is fundamentally different than application-internal information: it is cross-application, domain-independent, and situationally-based in the real world environments of the users. As we shall see, applications have different needs for sharing of coordination information than they do for the sharing of application information. These different requirements lead to different infrastructure solutions.

This research has resulted in the development of a collaboration support environment that addresses the research goals of this work; this support environment is called *Intermezzo*.

Partitioning the Problem Space

There are a large number of research problems related to coordination. In this research, I am specifically interested in examining a subset of these problems. My interests can best be described as an examination of ways to bring situational information into the collaborative environment, and the issues that must be dealt with when this information is readily available: what are the implications of having this information present, and how can we leverage it to support more powerful coordination among users.

The central thesis of this work is that it is possible to bring a wealth of situational knowledge into a collaborative environment by modeling the activities of users. Further, once this information is present, it can enhance the quality and richness of collaboration.

My research is centered around three aspects of coordination:

- Awareness
- Session Management
- Policy

Awareness is the notion of knowledge about the activities, locations, and situations of other users. Session management dictates how the collaborative process is started. And policy governs how users express their desires about how the system should react to a changing environment.

While these three aspects of coordination may seem disparate, they are actually closely related. They build upon one another, share a commonality of implementation, and interact extensively. A collaboration support environment that provides all three can achieve a synergy that goes beyond what these components can achieve in isolation.

Some previous research has addressed certain aspects of coordination in isolation. My work is the first to address not only infrastructure for coordination as a whole, but is also the first to detail the implications of coordination that are revealed when situational information is present in the environment.

A Development Model

My work is not concerned solely with a theoretical partition of the space of coordination. I am also interested in devising a set of concrete mechanisms that can used by developers to satisfy the coordination needs of the applications they develop. To this end, I am not only investigating *what* coordination features are useful to collaborative applications, but also:

- How these features can be constructed economically and efficiently.
- How these features can be effectively used by developers.

This research will show that all three of the aspects of coordination that I am studying share a certain commonality of implementation: it is possible to devise a single software substrate on top of which all three can be constructed.

I am also investigating development paradigms that can be used to make the construction of robust applications with sophisticated coordination needs easy. Current research into collaborative “toolkits” has largely focused only on the application sharing needs of collaborative systems (locking, view consistency, and so forth). Programming models for coordination have not been investigated.

A Methodology

The selection of features present in any software infrastructure (whether for collaboration, graphics, or any other problem domain) must be based on a requirements analysis of the applications that will use the infrastructure. Typically the design of the infrastructure is driven by features which are present and common in the target applications. For example, in the domain of graphical user interfaces, creating a dialog box is a common task which can be profitably ensconced in a software infrastructure that makes the task trivial to accomplish.

I have taken a top-down approach in this research. Starting from application requirements in the area of coordination, I have devised a set of requirements for a collaboration support environment intended to support coordination. The features of the infrastructure that are visible to application writers are called the *coordination features* of the infrastructure; these are features that support the three aspects of coordination discussed in the last section (awareness, session management, policy).

As stated, simply defining a set of coordination features is not sufficient. I also intend to show that these features can be implemented, that they share a common implementation, and that this implementation is both economical and efficient. The internal characteristics of the infrastructure on top of which the coordination features are constructed are called *foundation features*. Foundation features do not embody collaborative functionality; instead they are artifacts of the implementation of the system use to create such functionality.

The layering in the system is shown in Figure 1-1, “Infrastructure Layers.” My approach to presenting this material is to begin with a set of application requirements, derive a set of coordination features that can support those application requirements, and then derive a set of foundation features that can be used to implement the developer-visible coordination feature set.

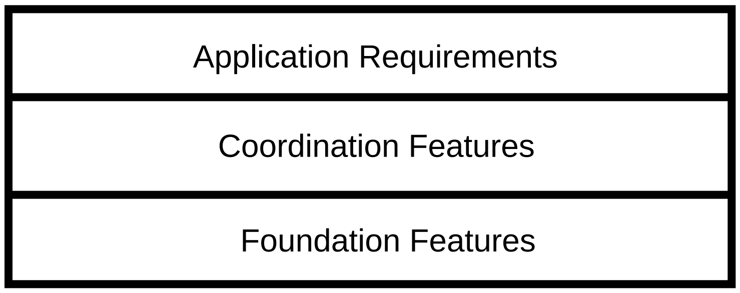


FIGURE 1-1 Infrastructure Layers

Once the requirements have been established, I will begin at the “bottom” by discussing the foundation features first; it is necessary to understand the foundation layer in order to understand how the foundation features are composited together to produce the coordination features. After the foundation layer has been introduced, I will address each of the coordination features in turn.

Dissertation Roadmap

The remainder of this dissertation is organized as follows.

Chapter II discusses the Intermezzo foundation layer in detail. This chapter describes the common features with which all of the higher-level coordination features are constructed.

Chapter III presents an overview of user awareness for coordination. This chapter presents a model for activity-based awareness and an implementation of that model based on the foundation feature set.

Chapter IV provides a taxonomy of session management as it exists in collaborative systems today. It also introduces several new forms of session management, implications for policy and access control, and an implementation of session management on top of the foundation layer.

Chapter V details the policy controls that are necessary to allow users to express their desires with regard to the collaborative environment. This chapter presents an extremely flexible system for policy control based on information provided by the awareness features.

Chapter VI examines programming models for creating applications that take advantage of the coordination features provided by Intermezzo. The programming interfaces provided by the system allow developers to easily create applications that can cooperate with the Intermezzo runtime service, and access coordination information from other applications.

Chapter VII summarizes the research of this dissertation. I present a set of conclusions, and detail some of the weaknesses of the approaches I have used.

Finally, Chapter VIII outlines some potentially fruitful areas of research that logically follow from the work investigated here.

Projected Contributions

This research contributes to the field in a number of ways. The first contribution is a taxonomy of the information sharing needs in collaborative applications. By separating sharing of coordination information from sharing of application information we can gain a clearer insight into the issues surrounding these orthogonal types of sharing, and examine the infrastructure issues related to both.

The second broad contribution, and indeed the main contribution of this work, is an in-depth investigation of the uses and implications of bringing various forms of coordination information into the environment. My research focuses on three forms of coordination support: awareness, session management, and policy. These forms are complimentary and interdependent; the presence of one form practically requires the presence of the others for maximal benefit and application support. Additionally, support for these forms of coordination carries with it a host of implications and potential problems that must be addressed for the support environment to be useful to applications: issues related to privacy, flexibility, and usability of information must all be addressed; this research focuses not only on the forms of coordination information sharing, but also on the implications of these forms of sharing.

The third and final general contribution is an investigation of the foundations of an infrastructure for coordination information sharing. A study of coordination features in a vacuum is of limited utility; therefore this work explores the requirements for the construction and use of coordination features by applications. My research investigates how an infrastructure for coordination can be built, and how it will appear to developers through its programming interfaces.

Chapter II

Foundations

This chapter provides an overview of the fundamental constructs in Intermezzo on top of which the higher-level coordination features are constructed. We begin by discussing the concepts in the foundation object model. This discussion details the runtime system used by Intermezzo, how the system performs authentication and naming, access control, information sharing and consistency, and persistence of data. This chapter also presents an in-depth rationale for the features that are present in Intermezzo, and compares the choices of foundation constructs with those present in other systems (such as object databases and toolkits for distributed systems programming).

Introduction

While the primary goal of this research is to identify and investigate a set of features for coordination that can be embodied in a collaboration support environment, a secondary goal is to identify how such a set of features could be implemented robustly and economically, in terms of volume of code and feature sharing.

In this research, I break the discussion of features of the support environment into two layers: coordination features and foundation features. Coordination features are mechanisms in the support environment that are visible to the developers of collaborative applications. They are features which embody some common task in developing collaborative software, and are specifically intended to support collaboration.

Foundation features, on the other hand, do not necessarily support collaboration directly. They are the “invisible” (to the developer) features of the support environment on top of which the coordination features are constructed. They do not themselves necessarily embody any special knowledge about collaboration or how groups of users interact. Yet they represent the commonality of implementation of the coordination features, just as the coordination features represent the commonality of application requirements in collaborative systems.

This chapter discusses the foundation feature set on top of which the higher-level developer-visible coordination features are built. My contributions in this area are not so much in the production of new technologies (new algorithms for distributed consistency or locking), but rather in the identification of a minimal set of features which can be used to support the higher-level coordination goals of the support environment. Many of the coordination features, while they may seem disparate, actually share some common implementation traits—in fact, it is possible to define a core foundation layer which can support all of the coordination features covered in this work, and more.

As we shall see, many of the foundation features used by Intermezzo are similar to those found in, say, object databases and distributed systems toolkits. The field of distributed systems theory, in particular, has produced a number of algorithms that, on the surface at least, appear to be amenable to applications in computer-supported cooperative work. We must take care, however, when applying “traditional” distributed systems approaches to collaboration. As Saul Greenberg, *et al*, have noted [44]:

> *Groupware is a fundamentally different application domain from traditional distributed systems, because the transaction process includes people as well as computers. Different concurrency control methods...have quite different impacts on the interface and how transactions are shown to and perceived by group members.*

The same can be said for design principles used in object database systems.

I believe that by designing the foundation of the support environment “from the ground up,” rather than basing it on some existing infrastructure for distribution and sharing, I can justify why *specific* features are required, or not required, for particular aspects of collaboration. This approach also frees us from the constraints of a world view that might be imposed from beginning with an existing foundation layer (such as an object database) and then building up from that, rather than basing the foundations strictly on the requirements of the higher-level features.

This chapter outlines the specific components of the foundation model for the Intermezzo collaboration support environment, describing those features in detail. After this presentation I will examine how the model used by Intermezzo differs from that of other collaboration support environments and more traditional systems, such as object-oriented databases. Previous research in these systems is also considered.

Note that this chapter describes the features of the foundation layer only, not the particular APIs or programming models used to access these features. Chapter VI describes how developers interact with the foundation features as well as the higher-level coordination features.

Object Models for Building Collaborative Toolkits

The Intermezzo foundation layer is based on an object-oriented paradigm. There are three primary concepts used in the foundation, representing data storage, computation, and communication. These abstract concepts are implemented as a collection of class hierarchies which developers can use to construct applications. Together these class hierarchies provide facilities to support a distributed data store, inter-thread, inter-process, and inter-machine communication, and secure computation services.

The three primary concepts in the foundation layer are Resources, Threads, and Ports.

Resources are the fundamental representation for data in Intermezzo. Resources are distributable, sharable, persistent, and access-controlled. Resources have a type associated with them, and contain a collection of key-value paired data. Keys are represented as strings, and the values associated with keys can contain arbitrary data (including even collections of other resources or data). Resources maintain the notion of ownership (the identity of the user who created the resource), and support access control lists for the resource as a whole as well as the individual attributes of the resource. Together, the collection of resources in the environment constitutes a distributed data store which is shared among collaborative applications.

Threads provide an abstraction for computation. Threads can be instantiated as needed whereupon they will perform some function and then exit gracefully. Like resources, threads maintain the notion of ownership.

Ports are an abstraction for communication. A port connects two threads, which may exist within the same address space, different address spaces on the same machine, or even different machines. The port class provides mechanisms for data interchange independent of any particular transport; port subclasses implement the port semantics on top of particular transport mechanisms.

Runtime Overview

Resources, threads, and ports form the building blocks of the Intermezzo foundation infrastructure. In the runtime environment, these constructs are combined into useful entities that can support higher-level functions.

The most important entity in the runtime environment is the Intermezzo server. The server is a process that has the responsibility of maintaining a sharable store of resource data, called a *context*. Each server maintains its own database of the resources known to it. In general, there will only be one server for any extended group of collaborators or potential collaborators.

The server ensures consistency among the resource data it maintains and cached copies of that data that may be located in clients. It also provides naming services (the ability to retrieve any object by its name or handle), and search facilities for locating objects in the database.

The server also provides the ability to store and retrieve data objects to and from long-term storage. Persistence is used for checkpointing of crucial data, and for storage of data that, by its nature, should be long-lived.

Clients have the ability to request that the server notify them when certain “interesting” events occur. The server provides an array of notification mechanisms which can be used by clients, including events and triggers. The server also supports “embedded” computation at the request of clients. Thus, clients can download code into the server to be run on their behalf.

To summarize, the main functions of the Intermezzo server include:

- Storage and coordination of resources among clients.
- Long-term persistence of resource data.
- Name service and rendezvous among clients.
- Remote code execution facilities.
- Remote notification facilities.

The server process consists of a number of threads, interconnected by ports. The server process instantiates a special “listener” port that accepts incoming connections from clients. When a client connects, a new port is created to be used for communication between the server and the client, and a new thread is instantiated that will service requests made by that client.

A well-defined protocol that enumerates the possible service requests and responses is passed over the client-server connection (see Appendix C for a description of the protocol). It is this protocol that defines the operations allowable on the shared data store implemented by the server.

An Intermezzo "client" is simply an application that communicates with the Intermezzo server process. A developmental toolkit provides communication and other services that make it easy for applications to engage the server. The client-side toolkit is itself built using the "building block" classes. Resources are used within the toolkit to store data that is potentially distributable. The toolkit instantiates a number of threads to deal with communication, consistency update messages, and event processing. And these threads are connected to each other and to the server via port instances.

The Intermezzo "universe" may be viewed as a set of cooperating threads joined by ports. This web is distributed across machines and processes, with some threads executing in the server and other threads executing within client applications running on hosts on the network. The server process is a central coordinating authority among clients. The server connects a set of collaborative applications, and by extension their users, together in a fabric of data sharing.

Authentication and Naming

Authentication

All of the constructs which are visible in the runtime system (threads, ports, resources) maintain the concept of *ownership*. Objects are considered to be owned by the user who created them (rather than, say, the application or machine they were created from); ownership is established at object creation time and is immutable.

In any system with authentication, users must be represented by some machine-internal form. In the UNIX operating system, the ultimate representation of a user is the user ID number. In Intermezzo, users are represented by an identifying string called a *certificate*. Certificates typically contain a name and electronic mail address and are considered to represent the ultimate identify of a given user. An example certificate is:

```
W. Keith Edwards <keith.edwards@cc.gatech.edu>
```

Intermezzo certificates are essentially the same as those used by some electronic mail privacy systems, notably PGP [116]. The certificate forms a unique and easily human-readable token for a given user. Note that the name is not required for uniqueness, but is included in the interest of readability.

A simple but reliable authentication procedure is performed whenever a new client connects to the Intermezzo runtime service. RSA (Rivest, Shamir, Adleman) public-key encryption [85][86] is used to implement a digital signature system that supports secure authentication of client applications.

Upon connection, the server transmits a random string to the client. The client takes an MD5 hash [84] of the string, encrypts the hashed version of the string with its private key, and transmits the cyphertext version along with a plaintext copy of its certificate to the server. The MD5 hashing function is used for performance, since encryption using RSA private keys is an expensive operation. MD5 produces a shorter plaintext, which can be encrypted more quickly. This technique is commonly used in public key cryptosystems, see for example Davies and Price [16].

Upon receipt of this message, the server retrieves the public key corresponding to the claimed certificate from a database of public keys. The cyphertext string is decrypted using the public key and compared to the original string. If the two match, then the identity of the client is authenticated, since only the user represented by the certificate could have known the private key for that certificate.

This procedure is not perfect, since it exposes both plaintext and cyphertext versions of the string to "snooping" attacks. But the size of the keys used limits practical attacks. If desired, stronger authentication procedures are available [96].

Once authenticated, any sharable objects (threads, ports, resources) created by the client will be marked as being owned by the user running that client. The result of this authentication scheme is that the collaboration support environment can provide secure checks of identity, and trusted knowledge about the "true" ownership of any object in the system.

The public and private keys are created via a key generation utility, and are not typically available in "human-readable" form. Instead, the private key (which must be kept secret), is stored in the user's home directory, and is itself encrypted via the Data Encryption Standard (DES) algorithm [70]. DES uses a human-readable string (a password) as a key to decrypt the larger, more secure, RSA private key.

The initial assignment of users to certificates, and the registration of public keys is performed by an administrator; a protected directory contains the mappings between certificate names and public keys that is used by the system at runtime to authenticate users. Once the initial mapping has been put in place by an administrator (that is, once the user has been "introduced" to the system and security established), users are free to change their keys at will. The system can authenticate users who request that their keys be changed.

Naming

All sharable objects have a unique ID associated with them. The ID is itself an object which is represented by a tuple:

$$ID = \{Certificate, Process, Identifier\} \qquad \text{(EQ 2-1)}$$

The *certificate* identifies the actual human user who created the object. *Process* represents the particular client which created the object (a single user may be running multiple clients at the same time). *Identifier* provides a unique token for each object within a particular client, and is implemented as a per-client counter. IDs are guaranteed to be unique, network-wide.

Objects created by clients do not necessarily become shared with other clients: they are only shared if the client chooses to expose them to the server process which coordinates distributed access to objects. For this reason, resource names are actually generated by the clients themselves, not given to them by the server. Clients must be able to generate "local-only" resources easily, without server overhead. This ability is especially important for supporting asynchronous and autonomous collaboration. The tuple naming scheme supports this easily.

When objects become shared, the server validates resource names to ensure security, but still accepts any name generated by clients as long as the certificate matches the authenticated *certificate* for the client, and the *process* token is unique for the client. Note that the client-generated object name scheme is similar to that used by the X Window System [95], in which the server communicates a per-client token, and clients build their own global resource identifiers which consist of the per-client token and a unique within-client value.

Authorization

Resources are protected from unauthorized access by a set of access control lists, or ACLs. ACLs specify the rights allowed to users. In essence, an ACL is a map which, given a certificate, produces an access right token which corresponds to the rights allowed to the user represented by that certificate.

The system maintain ACLs for the resource object as a whole, and for individual attributes on a given resource. Resource and attribute ACLs control the ability to read and write member data of the resource, append new attributes, check for existence of the resource or an attribute on it, or delete resources or attributes. Access rights are strictly ordered, so bestowing a particular right to a given user bestows all rights of lesser power as well.

The access rights supported by attributes of resources are:

- None
- Exist
- Read
- Write
- Del

None indicates that the associated user has no access rights whatsoever to a particular attribute: the user may not read or write the attribute, may not remove it, and may not even determine whether the attribute exists in the resource.

Exist allows the indicated user to test for the existence of the attribute; however, the user has no permissions to read the actual value of the attribute.

Read allows the user to examine the contents of the attribute.

Write allows the user to modify, append, or overwrite the contents of the attribute. The user may remove all information associated with an attribute (essentially writing a “blank”), but may not remove the attribute itself (its key).

Del gives the user the permission to remove the attribute from the resource entirely. Since rights are ordered, the user also has permissions to read or write the attribute.

The access rights supported by resources as a whole are:

- None
- Exist
- Del

None indicates that the specified user has no access rights to the resource. The user may not examine any attributes of the resource, and may not even determine its existence.

Exist grants the user the right to determine whether the resource exists. Once the user can determine existence of the resource, he or she may attempt to access individual attributes of the resource, constrained by any access rights on those attributes (see below for a discussion of interactions between resource rights and attribute rights).

Del grants the user permission to remove the resource entirely. Of course, a user with *Del* rights can also examine resource contents or determine the existence of a resource.

Resource rights interact with attribute rights. A resource with *None* rights set on it will not allow any access to its member attributes, even if those attributes are flagged with permissions which would otherwise allow access. In order to access any attributes, a handle to the resource containing that attribute is required. Thus, *Exist* rights are required for any potential access to attributes. Once *Exist* rights to a resource have been granted, individual attribute rights come in to play and decide the access to data members of the resource.

Table 2-1, "Interaction of Resource and Attribute Rights," shows the how the two types of rights interact to yield new "effective" attribute rights. The horizontal axis of the table shows the set of access rights that a resource may take; the vertical axis shows the set of possible attribute rights. The intersection of the original right on that attribute with the right of the resource that contains it shows the "effective" right on any given attribute.

TABLE 2-1 Interaction of Resource and Attribute Rights

		Resource Rights		
		NONE	**EXIST**	**DEL**
Attribute Rights	**NONE**	NONE	NONE	NONE
	EXIST	NONE	EXIST	EXIST
	READ	NONE	READ	READ
	WRITE	NONE	WRITE	WRITE
	DEL	NONE	DEL	DEL

The use of both resource and attribute rights allow applications to "open up" certain data to allow other users or applications to access it. For example, an application may create a resource with *Exist* rights on it, generally allowing others to test for the existence of the resource, and perhaps read, update, or even remove attributes of the resource, depending on the rights of the individual attributes. Others would not be able to delete the resource entirely, however. Some data stored in attributes may be considered sensitive, and would be granted the *None* right. In general, most attributes would have *Exist* or *Read* permissions associated with them. Occasionally, some attributes may be world-writable, in which case they would have the *Write* permission on them.

Ports maintain an ACL which designates which users are allowed to write data to the port.

Threads do not maintain ACLs. Operations on threads from "external" sources (such as starting, stopping, or killing a thread) are only allowed to the owner of the thread. Other than this, the code executing in a thread is free to implement its own access control policies. For example, a thread may create a port for communication with other threads, and allow any thread to write data to this port. But the code that reads data off the port can decide how to respond to the messages it receives based on the originator of the message.

Note that it is important that ownership of objects is established in terms of actual human users, rather than more abstract "virtual" concepts like processes or clients. Defining ownership in terms of certificates allows access control lists to grant access to specific people regardless of the machine they may be on or the applications they may be running. Users can grant access directly to friends and coworkers without having to go through intermediate concepts like hostnames.

The access control list scheme used here supports the principles that Saltzer [92] (as reported in Dewan [23]) has identified as important:

- *Permission rather than exclusion.* By default, users have no rights to objects they do not own (that is, *None* is the default permission); rights are specifically granted to allow access.
- *Total mediation.* All accesses to objects are checked for correct authorization.
- *No secret design.* The ACL-based security mechanism does not depend on obfuscation or secrecy of the access control scheme. Instead, it relies on cryptographically-secure authentication of identity based on the secrecy of keys (not algorithms).
- *Least privilege.* Users should possess the least strong right required to do their work. The multiple rights supported by the ACL scheme allow selective granting of rights.
- *Ease of use.* The Intermezzo policy system provides a veneer on top of low-level access control lists that allows users and application writers to describe access control to the system. This feature is covered in detail in Chapter V later in this work.

The foundation layer provides mechanisms for restricting access to resources; these mechanisms are combined and extended by the higher-level coordination features to implement constructs such as policy and roles (see later chapters for more information).

Sharing

The Intermezzo foundation layer supports both shared and local-only objects. An object that is shared can be accessed by multiple clients simultaneously, given the proper access rights. Furthermore, the “view” of this shared object from all of the clients can be consistent (meaning that they will all see the same version of the data in the object).

Whenever a client is accessing a shared resource, it maintains a *replica* of that resource. A replica is a locally-cached copy of the global data maintained by the server. Each replica can have its own parameters that determine how closely it will mirror the “master” copy of the data on the server.

A collection of resources is called a *context*. Contexts may be local (meaning that they only contain resources visible to the client that created the context), or they may be shared (meaning that they contain data potentially visible to many clients). Clients may create one or more local contexts, which are essentially “scratch spaces” for their own data storage needs. The server implements a single shared context that clients may connect to. Any resources located in the shared context are potentially visible to any clients connecting to the server (only *potentially* visible because authorization constraints may prevent given clients from seeing particular resources).

Resources from a local context may be moved into the shared context. The process of copying a resource between contexts is called *cloning*. When a clone or replica of a resource is made from a local context to a shared context, the original copy remains in the local context. Once shared, other clients may clone the resource from the shared context into their own local contexts.

The foundation layer facilitates keeping local replicas consistent with the shared copy via a variety of *consistency policies*. See the section "Consistency," below for more details.

Consistency

Consistency is the degree to which a local replica of a resource is kept synchronized with a replica present in a shared context.

Each replica in a local context has a consistency policy associated with it that determines how closely the replica will follow changes to the shared copy, which is considered to be the master. If there are many local replicas across many machines, each may have its own consistency policy.

The server and clients cooperate using a simple read lock/write lock protocol to ensure consistency. Whenever a client wishes to read an attribute, it issues a read lock request message to the server. The server will grant the request for a read lock if there are zero or more read locks currently granted, and zero write locks currently granted. The server locks the entire resource containing the attribute, and returns the most current global snapshot of the resource in question when the request can be granted. Clients block until the request is granted.

When the client has finished its read operation, it relinquishes its read lock by issuing a read unlock request. The server removes the lock from the resource.

If a client wishes to update an attribute on a particular resource, it issues a write lock request message to the server. The server keeps a queue of pending write locks, and grants the request whenever no other locks are held by any clients. The server responds with a message granting the lock and returning the most current snapshot of the resource being updated. The client blocks until its request for a write lock has been granted by the server.

When the client has finished its write operation, it releases the write lock by issuing a write unlock request to the server. This message also passes the newly-updated version of the resource back to the server, which becomes the new “current” version. At this point the server removes the write lock and takes the next requested lock in order.

Locks are granted on entire resources, not attributes of those resources (thus, hierarchical locking is not used). The current client-side library will request and release locks each time an attribute is accessed. The server performs access control checking at the time locks are requested, to ensure that clients cannot read or write data that they are not allowed access to.

Consistency policies are ordered from greater to lesser degrees of consistency. Intermezzo assures that client replicas will receive consistency at least as strong as their specified policy. For performance reasons, the runtime system will sometimes update local replicas even when their consistency policies would not have indicated that an update should be performed. [1]

The consistency policy to be used for a given replica is specified whenever the replica is initially cloned.

1. The “at least as strong” consistency guarantee allows greater performance on the client side. The toolkit supports multiple references (or *handles*) to a given resource within a single address space. These handles are essentially implemented as reference-counted pointers to the actual resource data object. This property allows us to achieve consistency updating within a single address space “for free” (the one data object is updated and all handles “see” the change at once). Supporting different consistency policies for each handle or replica would require multiple copies of resource data within the client, and hence would incur a cost of performing updates to keep a client’s own copies of a resource in sync. To avoid this cost, the single copy of the resource data is updated with the strongest consistency policy required by any handle to it within the client address space.

The consistency policies supported by Intermezzo are:

- Atomic Synchronous
- Lazy Synchronous
- None

Atomic synchronous consistency is the most strict form of consistency supported by Intermezzo. Client replicas with atomic synchronous consistency are updated whenever changes are made to the server copy of the data. When the server updates a resource value, it flushes updates to all clients with atomic synchronous replicas. Atomic synchronous consistency ensures that clients always have a fully consistent copy of replicated data.

Suppose clients *A* and *B* each have replicas of some resource *R*, and that *B*'s replica was created with atomic synchronous consistency. *A* updates an attribute on *R*, causing a write lock to be issued from the server. Whenever *A* completes the update, it releases the write lock, returning the changes made to the replica to the server. Whenever the server receives the message that *A* has finished updating the resource, it sends a consistency update message to *B* which causes the local replica of *R* to be updated to the new state. An example is shown in Figure 2-1, “Atomic Synchronous Consistency.” As Client *A* updates resource *R*, the updates are immediately propagated to Client *B* (which is assumed to have a replica of resource *R*).

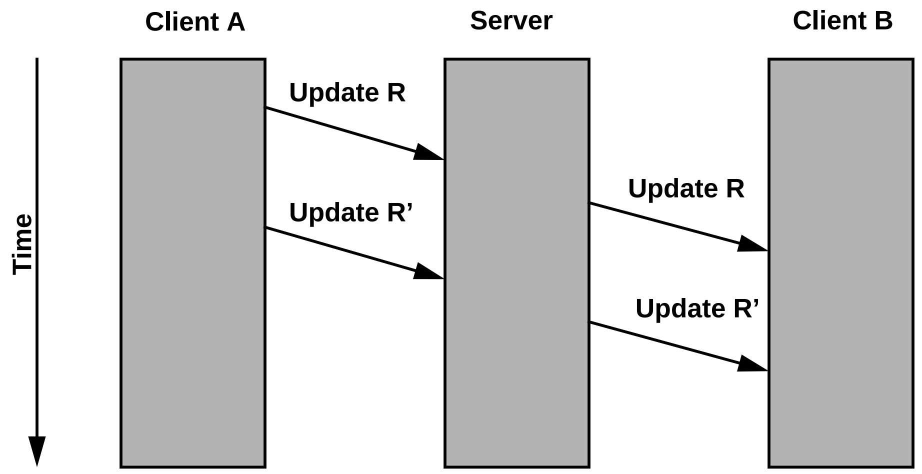


FIGURE 2-1 Atomic Synchronous Consistency

Atomic synchronous consistency is expensive both in terms of network overhead and computational workload. In general, lazy synchronous consistency (see below) is used by clients that have a need for synchronous data. Atomic synchronous consistency is primarily useful only for clients which bind trigger functions to local resource replicas (rather than downloading them into the server) and require that the triggers be run in a timely fashion after the resource has been updated. Trigger functions are explained in more detail in Chapter VI later.

Lazy synchronous consistency, like atomic synchronous consistency, provides a fully consistent view of the data, but with less network and computational overhead. In lazy synchronous consistency, updates of local replicas are performed only when the client owning the replica “looks” at it. Replicas that are unexamined may be several generations out-of-date with respect to the server copy. This is unlike atomic synchronous consistency, where all replicas are updated at once.

(Note that my use of the term “lazy consistency” differs from that used by Narayanaswamy and Goldman [69], where “lazy consistency” is used to indicate a policy of broadcasting proposed changes, and then formulating consistency requirements as the state of proposed changes evolves.)

Suppose clients *A* and *B* each have replicas of a resource *R*, and that *B*’s replica was created with lazy synchronous consistency. *A* may acquire and release multiple write locks on *R* over a period of time; during this time the server generates no consistency updates to *B* if *B* never attempts to read or write *R*. Instead, at some later point in time when *B* reads an attribute of *R*, it acquires a read lock from the server. When the server grants the read lock, it returns the *current* state of the resource to *B*. This new snapshot may contain the results of any number of changes to the resource since *B* received its last update. An example is shown in Figure 2-2, “Lazy Synchronous Consistency.” Client *A* updates resource *R* repeatedly. Only when Client *B* attempts to read the resource’s state is the most recent consistent version sent to it, as a reply to its request to read resource *R*.

Lazy synchronous consistency allows clients to use resource data safely (that is, they can be assured that the data is consistent and up-to-date), but without the overhead of continual updates. The client’s view of the resource is synchronized with the server’s only when the client accesses the data. Lazy synchronous consistency is the most common consistency policy, and may be used safely anytime the client is not using trigger functions that may need to be executed immediately when the server’s (that is, the “current” or “correct”) copy of the resource is updated.

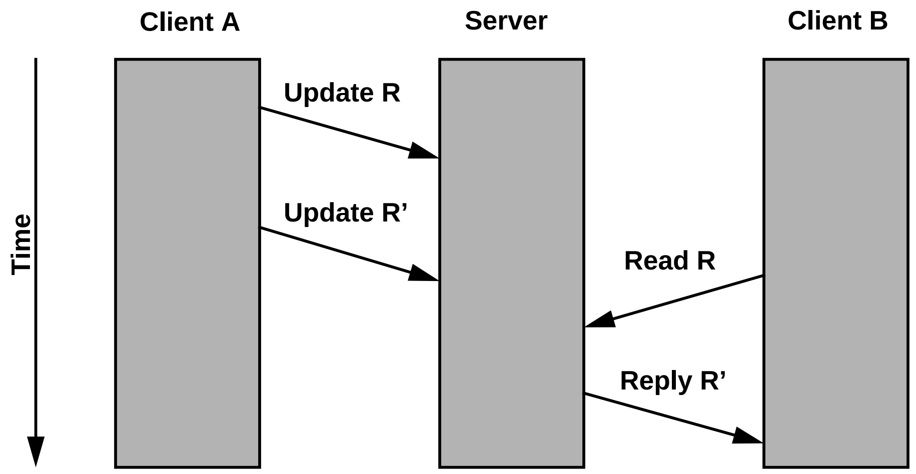


FIGURE 2-2 Lazy Synchronous Consistency

The *none* consistency policy indicates that the replica is a one-time-only snapshot of the state of a resource at a single moment in time. The server generates no consistency updates to a resource replica marked with this policy, and clients do not issue read or write lock requests. The default consistency policy is *none*.

Note that Intermezzo is not a true distributed system: there is only one unique server process. Further, the resources present in the shared context implemented by the server are considered “authoritative” for purposes of consistency. Local replicas only mirror (to a degree determined by the level of consistency specified) the copy in the shared context.

See the chapter “Future Directions” for an indication how of the non-distributed nature of the system affects consistency policies, and a discussion of some possible future consistency policies.

Persistence

Resources support the notion of *persistence*, which is an indication of how long-lived the resource will be. Unlike consistency, which is a per-replica characteristic of resources, persistence is a trait of a resource as a whole (no matter how many replicas exist of it).

Like consistency, however, a number of policies are supported for persistence. The persistence setting controls when a resource will be destroyed. In all cases, resources may be destroyed by an explicit request to free them. There are other conditions under which a resource may be freed, however, and these conditions are controlled by the persistence policy in effect.

The policies are:

- Transient
- Client
- Server
- Permanent

Transient persistence indicates that the resource should automatically be freed (destroyed) when there are no active “handles” or references to it. Thus, if a resource is used locally as “scratch” storage it will be freed when it is no longer in use. Resources that are shared but marked as transient will be freed when no context contains a replica of the resource.

Transient persistence is the default mode for persistence, since it is the cheapest to implement and yet ensures that data will remain intact while it is in use.

Client persistence indicates that the resource is now allowed to outlive the client which created it. Thus, if a client creates a number of resources of client persistence and then exits, all of those resources (meaning all of the replicas of those resources anywhere they may exist) will be freed, even if the client has not explicitly deleted them. Client persistence is useful for ensuring that the global data space is not polluted by “dead” resources left in place by inactive clients.

Server persistence indicates that the resource is not allowed to outlive the server maintaining the shared context it exists in. Thus, even if the client which created the resource terminates or disconnects, the resource persists in the shared context and any local context which may have a replica of it. If the server terminates (either expectedly or unexpectedly), all replicas of the resource will be freed. Server persistence is useful for data which is significant beyond the scope or lifetime of a single client, but loses its value at the end of a session (as denoted by a server exit).

Note that server persistence assumes that the resource exists (or will exist) in a shared context, since the notion of server persistence is only meaningful if a replica of the resource is visible to the server. If a resource is created with server persistence but never shared, it will be freed when the client that created it terminates.

Permanent persistence indicates that the resource should be written to long-term storage and will outlive even server invocations. Permanent persistence provides a mechanism for checkpointing resource data and recovering it after server restarts. Like server persistence, permanent persistence only makes sense for resources which exist in a shared context; only the server has the facilities to save resources to permanent storage and recover them when it restarts.

Note that with any of these policies, an explicit request by an authorized client to free a resource will cause the resource (and all of its replicas) to be freed. These policies only govern when resource data will be freed if no explicit request is made.

Comparison with Other Object Models

The facilities provided by the Intermezzo foundation layer are in many ways similar to those investigated by previous research in distributed programming environments, object databases, and collaboration support environments. The set of features included in the foundation layer was determined by examining the higher-level coordination features, and then basing the foundation feature set on the requirements of the coordination features.

When this research began, I first started to look at creating my own foundation layer because I felt that many of the features of, for example, full-blown object databases or distributed programming environments, were unnecessary for supporting coordination. As the research progressed, it became clear to me that some “classical” features of such systems are in fact needed, along with some other fairly novel features which proved to be useful. Many of the features of these environments are not needed, however.

In this section I will examine a collection of particular features common to object databases and distributed programming environments. I will present my rationale for why some features were included in the Intermezzo foundation and why others were deemed not necessary.

There are actually two distinct object models present in the Intermezzo foundation. One is the set of constructs manipulated by the programmer: concrete classes representing resources, threads, and ports. The second is the space of data represented by the resources present in the system at runtime. The first is the development-time object model; the second is the run-time object model.

As an example of the distinction between these two, consider a programmer who is manipulating an object (perhaps implemented in C++) that represents a resource in his or her program. This object has methods on it for retrieving the attributes of the resource, updating attributes, and so on. But this class is a *proxy* for what it represents: the “real” resource, which is represented as a collection of data in the server, and possibly in other local contexts across the network. Interaction with the actual data represented by the resource is accomplished via the server protocol (even though this protocol is largely hidden from the developer).

Thus we can speak of the programming model provided by the particular class library used by the developer (the collection of classes and the methods on those classes), and we can speak independently of the facilities available for accessing and altering the shared space of resource data via the server protocol. The facilities available via the server protocol in fact define the “true” object model, which is only manifested in the particular collection of classes and methods in whatever language the developer happens to be using. The server protocol defines an abstract object model which is realized and used via a particular API. It is the server protocol, not the methods that allow access to it, that define the operations that are possible on the shared data space.

Chapter VI discusses the actual programmers' view of using Intermezzo. The comparison in this section is based on the abstract object model which is accessed via the server protocol.

Table 2-2, "Summary of Foundation Feature Inclusion," summaries the features which are present in the foundation data model. Understanding the rationale for some of these features depends on understanding the higher-level coordination features that motivated them. For these features, only a brief justification is given here. We shall refer back to the foundation layer when we discuss the particular coordination features present in the support environment.

TABLE 2-2 Summary of Foundation Feature Inclusion

Feature	Used?	Rationale
Objects	◐	Used for encapsulation and typing; some other features not used.
Inheritance	○	Object types are fixed by convention; no clear inheritance hierarchy.
Methods	○	Objects used for storage only; utility of methods not clear.
Replication	●	Propagation of awareness requires sharing.
Persistence	●	Persistent subjects allows us to refer to inactive users.
Consistency	●	Supports safe use of data.
Ownership	●	Authentication and ownership required for access control.
Access Control	●	Supports privacy and policy controls.
Computation	●	Efficiency of database access; flexible notification.
Events	●	Can support most application notification requirements.
Triggers	●	Our object model supports triggers as a "catch-all" notification scheme.
Activation	○	No methods on object, so no need for activation/passivation.
Linking	●	Activity records must refer to other resources.
Distribution	○	Useful for scalability, but not required.

● = Feature Included ○ = Feature Not Included ◐ = Feature Partially Included

Objects, Inheritance, and Methods

The notion of an object has a fairly rigorous definition in the research literature: an object is an entity that supports (1) data hiding and encapsulation, (2) reuse through subclassing, and (3) extensibility through polymorphism [64]. The popular literature has a somewhat less rigorous definition of object. Typically in this definition an object represents a “chunk” of data which is typed and can be manipulated as a unit.

Intermezzo provides a shared data storage facility. The data items populating this data store have some features in common with popular definitions of objects, but are not objects in the restricted sense of the word.

Intermezzo data items (resources) are discrete entities which have a type associated with them. Resources can be subclassed indirectly by aggregating the attributes on the parent classes into a new resource, but there is no direct support in the server protocol for inheritance.

Further, and perhaps most importantly, the shared data items do not support methods (associated functions with special access to data within the object). Without methods, there can be no provision for data hiding in resources.

In a sense polymorphism can be supported by resources, since a client can determine if a particular attribute exists by querying its name. The indirection of attribute lookup by name is similar to late binding of function names to functions in method dispatch, and does provide “polymorphic access to data” (which is not, however, polymorphism in the traditional sense).

A significant amount of prior research has explored data models for collaborative systems; most of these have used a “classical” object model, with many of the extensions for multi-user access discussed here (replication, persistence, access control, and so on). Grief and Sarin [46] establish the scope of data management requirements of collaborative systems, and introduce a set of features important to collaborative applications. Many of these features, including triggers and linking, are supported by the Intermezzo foundation layer. Others, such as inheritance, are not present. Suite [18] uses a data model based on *active values* that are shared and “coupled” with each other, and with interface components in the users’ environments. Other systems, such as ConversationBuilder [55], take a more structured approach to infrastructure: a set of “building blocks” to construct domain-specific collaboration tools, rather than the free-form object model used by Intermezzo.

There are, however, a number of reasons that a full object model is not necessary in the shared data store:

- Resources are passive entities. The are used only for storage of information. Thus, the utility of providing methods on these objects (other than the “built-in” ability to get and set attributes, and change resource traits such as consistency, persistence, and access control) was not clear.
- Without the use of methods, data hiding (separating data items from the methods used to retrieve them) is a *non sequitur.*
- Subclassing is almost never used in the coordination feature set, so I felt that special facilities for subclassing were not needed and would increase the complexity and size of the foundation layer.

Of course, while a full object model in the shared data store is not necessary, it would almost certainly not detract from the power of the system. In fact, it may be helpful for the sole reason that it corresponds to programmer expectations about how “object-oriented” systems should behave.

Note that while the shared data store in Intermezzo is not based on a full object-oriented paradigm, the actual APIs and programming models used by developers are.

Throughout much of this writing I will use the term “object” to mean a resource in the shared data store. Primarily this is for lack of a non-overloaded word in common usage that expresses the concept of a self-contained data entity.

Replication

The Intermezzo foundation layer supports replication (essentially consistent local caching) of data objects. The server process maintains an “authoritative” copy of the data, which local replicas mirror, to some greater or lesser degree. Data sharing requires replication of data structures for efficiency, especially when the number of clients is large. Replication provides a degree of concurrency in the system, and moves the data closer to the computation that operates on it. The alternative to replication is to keep no cached data cached on clients; every access to a data item must proceed through the server. There are other characteristics of a shared data service which must be considered when replication is brought into use however: consistency and distribution are perhaps the most important of these and are discussed below.

Note that since the server, as a centralized entity, maintains an authoritative copy of the data, Intermezzo is not a true distributed system. Replicas are only used as cached copies of the “real” data in the server. Through the use of various consistency policies (see below), performance can be increased over a purely-centralized, non-replicated arrangement.

Persistence

Persistence has been identified as a useful feature for a number of collaborative applications in the literature. For example, RTCAL [46] uses persistence to support versioning of data; Dewan’s object model for conventional operating systems [24] uses persistence as a safe-guard against machine or software crashes; Arjuna uses persistence as a general application-building tool for distributed systems [25][26]. Persistence is present in the Intermezzo data model because it adds value to the object model; specifically it supports a number of useful constructs in the higher-level collaboration feature set. In particular, persistence is required for storing information about users who may or may not be logged on to the system; without persistence, all information is transitory based on the lifetime of the client application which created it. Non-persistent user data makes some types of awareness applications more difficult to realize.

(There are several other reasons persistence was added to the data model. These will be discussed in Chapter III in more detail.)

Consistency

In many situations where data is replicated, consistency is a desirable characteristic of the data model. Consistency ensures that all applications have a coherent and meaningful view of the data space. Without hard consistency guarantees, data in replicated objects is worthless for some applications since its correctness cannot be assumed. Thus, nearly every system designed to support collaborative applications has provided some form of consistency control. The PREP collaborative writing environment [73], the Duplex editing system [75], and the GroupKit toolkit [44][88] (among many others) have all provided some form of consistency control over shared data.

The Intermezzo model provides a spectrum of consistency settings so that application writers can trade off performance against data integrity. Most applications will always want a fully consistent view of the data store, but some applications may only need to poll or sample the data space. For these applications, incurring the overhead of a distributed consistency mechanism is not needed. As noted in [44], the domain of collaborative systems is different than that of traditional distributed systems. The wide range of needs of collaborative applications indicates that a range of consistency levels is useful.

Ownership

Ownership is the notion of associating a user or process with the data objects it creates (or “owns”). Ownership is supported in Intermezzo for two reasons: first, ownership is necessary within the server for tracking resource usage among clients. When a client terminates, the server may, modulo persistence constraints, free resources owned by a particular client or user.

Second, ownership is required for supporting access control on data objects. The access control subsystem must grant access to *something*. In the case of Intermezzo, that something is the human user represented by a particular certificate produced by the authentication/ownership subsystem.

Access Control

While access control to objects is not necessary for a collaborative system (and is, in fact, absent from many collaborative systems), it is often highly desirable to ensure privacy of user information and attributes [33]. Access control is especially useful when the data being shared represents potentially embarrassing information (or at least information not commonly available to strangers), or when social protocols are insufficient to prevent unwanted intrusion. Examples include the locations of users and their current tasks, all very useful to a system for supporting coordination, as we shall see later.

Collaborative systems often require the “buy in” of a critical mass of users before they can realize their potential benefit. If concerns about privacy of data are an impediment to user acceptance, then collaborative systems are less likely to be used. Concerns about trust, privacy, and access control must be addressed by any system designed to support coordination among users if it is to be effective.

The Intermezzo foundation layer provides strong guarantees about access control. Digital signature authentication guarantees trusted knowledge about who is requesting access to a particular data item. Users can restrict access to private information by creating access control lists for those data. Intermezzo access control lists provide a range of rights that may be used to govern access to particular objects by particular subjects. The set of rights is fairly coarse-grained (the ability to read an object, write an object, delete an object), but the data objects those rights are associated with are fine-grained. Access control lists may be bound not only to resources as a whole, but also to component attributes of those resources.

While the access control facilities provided by Intermezzo are not as flexible or general as those presented by, for example, Shen and Dewan [98], they still provide a substrate on top of which an extremely flexible and general policy subsystem can be constructed. An implementation of the policy systems described in Chapter V with a more robust access control foundation such as Shen’s would likely provide an even stronger set of mechanisms for authorization.

Computation

The Intermezzo server supports computation on behalf of clients. Clients can download code to be run in the server’s address space; this code runs with the ownership and permissions of the user who ran the client that downloaded the code. Several other systems have supported downloadable code, notably the NeWS window system (nee SunDew, [41]) and the Java language [42], but to my knowledge no collaborative environments have made use of embedded computation.

Computation in this form is not typically present in object database systems and most distributed programming environments. Instead, these systems associate computation with the data objects themselves and the computation is used to update the data object on which it is being run. That is, methods may be invoked on an object which resides someplace on the network; the computation of that method may happen on a different machine, but it is associated with the object itself and is not "global."

Computation in Intermezzo is not associated with any particular resource. Instead, computation is indicated by the creation of a thread in the Intermezzo server; this computation can take place even if no resources have been created by a particular client.

Server-embedded computation was added to the foundation layer for two main reasons:

- *Performance*. Computations which access or search a large number of resources may run more efficiently in the server where they have ready access to the data store, than in a client separated from the data store by a network connection.
- *Autonomy.* Users can download code to act on their behalf, even if they are not running any client applications. For example, a user may go on vacation for a week and download a piece of code into the server to collect "interesting" information for presentation when he or she returns. Without server-embedded computation, a client application would have to run continuously to sample the environment.

The ability to download and execute code on the behalf of clients is an important feature which has a number of uses and implications in Intermezzo. Chapter VI discusses embedded code in more detail.

Events and Triggers

Events and triggers are both mechanisms for notifying clients of changes in the "world state" represented by the global data store maintained by the server. Events are a common form of notification familiar to most developers (see Dewan [17] for example, among many others). Clients solicit certain types of events from the server that they consider to be interesting; the server then generates messages to the clients whenever an occurrence of the specified type takes place. Events provide a general and easy-to-use system for notification.

Triggers are a notion from access-oriented programming [104] in which the act of accessing (reading or writing) a data item causes code to be run. Intermezzo supports triggers on resource attributes and on resources as a whole (so that it is possible to specify code which will be executed whenever any attribute on a resource is accessed).

Triggers were added to the foundation layer as a useful mechanism for notification of changes in the data store; they provide a general-purpose tool for activating client code upon state changes.

Chapter VI contains more information on triggers, events, and other notification mechanisms.

Activation/Passivation

Activation and passivation are the notion of moving objects to and from long-term storage. Typically activation and passivation are present in object systems in which objects are active entities (that is, they can perform some computation in their own thread of control) [24]. Passivation allows objects that may not be needed in the near future to be "pickled" and placed in a cheaper long-term store. Activation is the retrieval of a passive object so that it may begin computation.

Intermezzo does not support these concepts of activation and passivation since the data items in the shared store do not support computation internal to the data items themselves. However, Intermezzo does support the storage and retrieval of "objects" from a persistent store, even though it does not "activate" these objects once they are retrieved.

Linking

Linking is the ability to create associations between two data items in the data space; linking has been used by a number of collaborative systems to provide associativity between data objects [9][10][46]. Intermezzo provides linking as a mechanism not only for providing references between objects, but also as a mechanism for forming collections of objects.

Any attribute of a resource can refer to another resource by placing the name of the target resource into the attribute slot. Links are used throughout the coordination features: for creating associations between users and their activities, grouping all of the tasks of a given user, and so on.

Distribution

Intermezzo does not provide a facility for true distribution of data. Instead, a single centralized server coordinates access to data and performs consistency updates among client replicas.

True distribution is important primarily because it support scalability to much larger numbers of clients than a centralized approach.

However, a distributed infrastructure was not provided in Intermezzo for a number of reasons:

- Most importantly, the amount of programmer effort required to build a true, robust distributed system is very large.
- The issues involved with distributed infrastructure are in many ways orthogonal to the issues revolving around how a shared data store would appear to collaborative applications.
- It is possible to support the same programming model (involving consistency, persistence, and so on) with either a distributed implementation of a centralized-with-replicas implementation. I have been able to explore the programming models that seemed relevant to coordination without having to build a distributed infrastructure.

I do believe, however, that a true distributed infrastructure would be superior to the current one, given the resources to construct such a system.

Evaluation

The definition of the Intermezzo foundation layer was driven by the requirements of the higher-level coordination features it was designed to support. Therefore, the foundation layer must be evaluated in terms of how well the foundation feature set supports the coordination features visible to application developers, and in terms of how minimal the feature set is. Further, it is useful to attempt to define the “edges” of the space satisfied by the foundation layer. That is, while the foundation layer may be sufficient for constructing the coordination features outlined here, where does it break down?

The first goal, sufficiency, is fulfilled for the set of coordination features I have investigated. I have been able to provide toolkit-based solutions for awareness, session management, and policy by building on top of the foundation feature set presented in this chapter. The sufficiency requirement will be addressed in more detail in later chapters, as the specifics of the coordination features are presented.

The goal of minimal implementation is a hard one to prove. I have structured my arguments for the foundation layer by describing a basic framework for the sharing of data objects, and then enumerating a set of features of this framework, and the objects contained within it. All of the features of the framework can be justified: information sharing requires replication and consistency; awareness requires privacy and data protection, for example. But it is harder to prove that the framework itself is the best possible solution to the coordination problems applications must deal with. For example, it may be possible that a completely non-object-based framework based on a different conceptual model may provide a cleaner and more elegant solution to the problems addressed here. I believe, however, that my object model, comprising constructs for data sharing, code execution, and communication, is simple, easy-to-understand for developers versed in object-oriented design, and is powerful enough to support the coordination semantics I am investigating.

The edges of the space of problems that are satisfied by the foundation layer are indicated by the caveats presented in earlier sections. The system does not support distribution, so it cannot be scaled to large numbers of client systems or users. The foundation layer does not present a “true” object model in the shared information space. Thus, facilities like inheritance are not trivially easy, as they would be in a real distributed object system.

In the end, however, the foundation feature set provided by Intermezzo is certainly capable of constructing the coordination features outlined in this research. It provides a simple model on top of which a number of useful and powerful mechanisms for supporting collaboration applications can be built.

Chapter Summary

This chapter has presented a set of features that can be composed to construct a collaborative toolkit for supporting coordination in multi-user environments. We have examined a basic object framework, comprising components for data storage and sharing, code execution, and inter-thread, inter-application, and inter-machine communication. This object framework is used to build a runtime environment and a client-side toolkit for use by application developers.

The design of this framework, and the features used in it, were driven directly by the requirements of the higher-level coordination features investigated by this work. We have justified the presence of features such as authentication, access control, consistency, and persistence. A number of features common in other shared object environments (such as distribution) are not present because they are not explicitly needed for the coordination features I am interested in.

We have compared the Intermezzo foundation layer to other object models for distribution, data sharing, and collaboration, and evaluated how well the system satisfies its requirements. In the next chapter, we begin looking at specific coordination features constructed on top of the foundation layer.

Chapter III

Awareness

This chapter introduces the notion of awareness as an important component in coordination and collaboration at large. We begin by investigating prior research in awareness for collaboration, and from prior art, derive a set of components of awareness, against which a computer-mediated awareness system can be evaluated. Next, we present a model for computer-augmented awareness based on modeling the activities of users. This model is realized in terms of the Intermezzo foundation layer and provides a systematic approach to capturing information about users spread across a network. This chapter evaluates the activity-based awareness approach, and compares it against other models of awareness, both computer-augmented and non-computer-augmented.

Introduction and Prior Art

Collaborative software, by definition, involves the interaction of multiple people working together to accomplish some task. In most current collaborative settings, these users are likely to be distributed in space or, in the case of asynchronous systems, even distributed in time. To interact more effectively, users need to be aware of others: their presence, actions, and so forth.

In the "real world," users maintain an awareness of others through a number of sources: peripheral sounds, quick glances, information from coworkers. Likewise, a number of computer-augmented awareness approaches have proven to be useful for the dissemination of coordination information to participants and potential participants in a collaborative endeavor. In fact, the whole of mediaspace research has focused on computer-based solutions to providing and augmenting awareness in spatially-distributed parties.

In the application domain, programs must convey a sense of the actions of other users within the application. Some examples of applications that provide within-application awareness include ShrEdit [63][74], Quilt [35][60], PREP [72], and GROVE [32], and summarized by Paul Dourish and Victoria Bellotti's work [27]. For example, in the ShrEdit shared editor, users are presented with a "Gestalt" view of the application. This view shows the (virtual) location of other participants in the editing session, and provides information about their foci in the document.

In the coordination domain, awareness systems intend to convey a sense of the user’s place within the world, instead of the user’s place within a given application. The goals of coordination awareness closely resemble those of “real world” awareness: they provide information about location and activity. Most current examples of coordination awareness tend to be mediaspace-like systems in which the computer is used as a facilitating device for propagating and restricting video signals from cameras mounted in the physical space of work.

The original Xerox PARC “Media Space” system [5] provided a set of audio and video channels that continuously connected participants in the system. The designers describe the system as follows [108]:

> *Media space is a setting for design groups to work together without physical presence. It is constructed with video, audio and computing technology. Within a media space, a group’s members can hear and see each other as well as share drawing, text, calculations, and computational tools. A media space permits the members of a group to interact with each other and to share information, even when the members are spatially or temporally removed.*

This description is categorical for a range of audio/video-based awareness systems explored in the research literature. One of the important results of the media space work is the degree to which mediaspaces can promote awareness among participants: awareness in a mediaspace is analogous to awareness within the physical world and “directly” supports most of the cues for awareness that we are accustomed to dealing with.

Another system that augmented the physical environment with near-constant audio and visual awareness cues is Cruiser [60][87]. Cruiser provided mediaspace-like functionality embedded in the notion of a “virtual space” (which happened to closely mirror the physical work environment) in which participants could find themselves in serendipitous encounters with others. In this case, coordination awareness was facilitated and constrained by a virtual environment that used a spatial metaphor for controlling access to awareness information.

Awareness research at Rank Xerox EuroPARC has studied a number of systems, including Polyscope and Vrooms [6] (the later of which, like Cruiser, uses a spatial metaphor for situated awareness). The Portholes system [28], connecting researchers at PARC and EuroPARC, also uses networked audio/video connections to support a sense of community and awareness among a spatially-distributed set of users. The RAVE [37][38] environment augments a traditional mediaspace environment with non-speech auditory cues to provide awareness information about current activities.

Work at SunSoft includes the lightweight awareness system Montage [110], and the asymmetric conferencing tool Forum [50][51]. Forum, in particular, emphasizes the need for awareness in a range of settings, including the lecturer/audience setting addressed by Forum.

Other mediaspace-like systems include work at the University of Toronto [62][97], NTT [52], and Georgia Tech [101][102].

Both the sheer volume of prior research, and the richness of the work illustrate the perceived value of machine-augmented awareness within the research community. More directly, the reported experiences of users when participating in such systems is indicative of their value. Users *like* working with these awareness systems, and report richer interaction with coworkers because of them [34].

Facilities for supporting user awareness are a ripe area for collaboration support environments, particularly in the coordination domain. Further, as we shall see, the particular mechanisms for supporting awareness can be leveraged in interesting ways by the other facilities of a coordination-oriented support environment.

Components of Awareness

The term “awareness” has a common usage which indicates mindfulness, or being conscious or cognizant of surroundings, situations, or other people. The research literature has not, however, set forth a technical definition of the term as it applies to computer-supported cooperative work.

In the course of this research, I have used a working definition for awareness, which has proven useful. This definition is based on separating the concept into a number of components. By defining the components that constitute awareness, we can identify related research and evaluate the performance of our system: how well does it provide the various components of awareness.

The list of components is not meant to be exhaustive, and is not even completely orthogonal. Instead it is meant to be indicative of the specific areas of awareness research that have been undertaken under the general moniker of “awareness.”

Four components of awareness that have been investigated by the literature are:

- Location
- Current Tasks
- Potential for Interruptibility
- Presence of Coworkers

We shall now study each of these in some detail.

Location

This component of awareness indicates the current point, in either physical or virtual space, that a user is occupying. Physical location may be an office, or a conference room. Virtual location may be a network host that the user is using, or a “room” in an application that uses spatial- or location-based metaphors.

Knowledge about location is important to coordination for a number of reasons. First, location can provide indirect information about the other components of awareness. Knowledge of location can give an indication of the tasks a user is involved in (location in a conference room implies a conference, location in the boss’ office implies a meeting, so on). Further, location is often a prerequisite for communication. We must know the physical location of an individual to address a telephone call to them, and we often must know the virtual location (network host, “room”) to initiate a collaborative or conferencing session with a user.

The spatially-oriented mediaspace applications such as Cruiser [87] have indicated the importance of “location” (whether physical or virtual) in awareness, as has other research which has explored the use of space in human-to-human communication (including work by Reder [80] and Gaver [36]). Indeed, since the cameras are fixed in most mediaspace-style awareness systems, one of the key awareness indicators provided by these systems is whether a user is located in his or her office.

Current Tasks

The second component of awareness is the set of tasks in which users may be involved. These tasks may be computer-oriented (applications) or “real world-oriented” (talking to a coworker or answering the telephone).

Knowledge of current tasks is important to coordination because it provides a sense of what’s occupying another person’s time and attention. Knowledge of current tasks provides an indirect indication of interruptibility (“Tom’s trying to beat a paper deadline; I shouldn’t bother him.”) and, perhaps more importantly, provides information about whether two people are engaged in similar tasks (“I see you’re running Framemaker now. Are you having problems with the license server too?”)

Propagation of information about current tasks is particularly important to awareness in collaborative settings since it provide knowledge about users who may be engaged in an activity which is relevant to yours. As an example, if you’re proofreading a paper, it may be important to know if others are proofreading, or if the author is engaged in a full-scale rewrite at the time you’re editing. Knowledge of current tasks is crucial for many design tasks, and is used in such systems as SASSE, where a “Gestalt” view of other collaborators is provided [3], and ClearBoard, where eye gaze is considered crucial for discerning the focus of user engagement [53].

Potential for Interruptibility

The third component of awareness is the *interruptibility* of a given user; that is, the level of engagement of the user in his or her tasks. The notion of interruptibility is a complex one, as it subsumes many issues which are difficult to determine even in social situations, much more so when computer-based approaches are used.

Interruptibility is a determined by a number of factors, and we can see by examining these factors that interruptibility is a particularly hard concept to capture in a machine-mediated system. The first factor is the timeliness of the work. If a particular task has a deadline for completion that is approaching rapidly, then the potential for interruptibility is obviously low. A second factory is the amount of value associated with the work. If the consequences of failure to accomplish the task are high, then the potential for interruptibility is low. A third factor is the degree of mental or physical “context” that is required to engage in the task. If a programmer must become mentally immersed in a piece of code before beginning to work on it, or a painter must gather and mix a number of paints before working, these individuals will be reluctant to drop work on their particular tasks; the overhead of restarting the task is too high. A final consideration in interruptibility is social factors (particularly hierarchy) in the workplace. Typically people are reluctant to interrupt those higher in the social or work hierarchy than themselves.

Interruptibility is important because it determines whether one user will be willing to attempt to communicate with another, particularly in synchronous collaborative situations. (Asynchronous systems can avoid interruption by “batching” work until the busy recipient can attend to it.)

The importance of interruptibility information to awareness is highlighted by Prakash and Shim [80], who indicate:

> *Individuals in office settings must routinely trade off having uninterrupted periods of time in which to get their own work done against being accessible for communication to others with whom they work. Individuals are frequently observed attempting to manage or alter this tradeoff. Asking a secretary to screen one’s calls, closing one’s office door, forwarding one’s*

phone, and so forth, are all strategies (requiring enabling technologies or individuals) for managing the tradeoff.

This quote indicates that providing information about the potential for Interruptibility is essential to supporting effective collaboration, and also indicates that providing users with the power to explicitly control this aspect of their environments may be a fruitful area of research (and is one that is discussed in Chapter V later in this work). The importance of interruptibility to awareness is also discussed in [91].

Presence of Coworkers

The presence, either physical or virtual, of others is the final component of awareness. This component is inherent to some degree in the other components: location, current tasks, and interruptibility are all affected by or help determine the presence of coworkers. I have chosen to list copresence as a separate component of awareness since it is particularly important in collaborative situations that, necessarily, involve multiple people.

Much previous work has indicated the importance of copresence as a factor in awareness. Most mediaspace studies have noted the effects of copresence as being important to building a sense of awareness and community in a work group (DIVA [103] and VideoWindow [34] are two examples).

The Intermezzo Awareness Model

Given the goal of satisfying application requirements for the components of awareness enumerated above, can we define a model for the collection and dissemination of coordination-oriented awareness information?

The Intermezzo approach to awareness is *activity based* in the sense that it models user activities to provide awareness information to applications and other users. The notion of an “activity” is captured in Intermezzo as a tuple called an *activity record*:

$$A_n = \{S_n, V_n, O_n\} \qquad \text{(EQ 3-1)}$$

S is the *Subject*, or the user involved in the activity.

V is the *Verb,* or the current task the user is engaged in.

O is the *Object*, or the “thing” being operated on.

The activity record tuple captures the notion of one user engaged in a single particular task on an object (or set of objects, as we shall see). The abstract components of this tuple map directly onto the foundation object model. In the actual implementation, the subject, verb, and object constructs are represented by Intermezzo resources. Each component of the activity record defines it own type, and its own set of slots that maintain information relevant to the object.

By convention, all Intermezzo-aware applications publish activity records when they run. Thus, applications are responsible for keeping the “picture” of the user up-to-date. Users are known in terms of their activities, and the objects of those activities. In essence, users are modeled in terms of their behaviors. (Such a system might be termed a “Skinnerian” or “behaviorist” approach to awareness—we can only understand others in terms of their actions.)

Consider an example. The user “iansmith” edits a file in his home directory using an Intermezzo-aware editor. The editor retrieves the information needed to construct resources representing the user, the application itself, and the file the editor is operating on. These resources are combined into an activity record that is published to the Intermezzo runtime service. Once published, the activity record and its components are available to any client application which wishes to examine it, and has the proper authorization to do so.

The global “context” in which collaborative applications are operating comprise a number of resources; these resources represent a space that may be searched using fairly sophisticated searching operations (see Chapter VI for more information). Figure 3-1, “An Application Publishes an Activity Record.” shows an application program running on a client machine generating an activity record on the Intermezzo server.

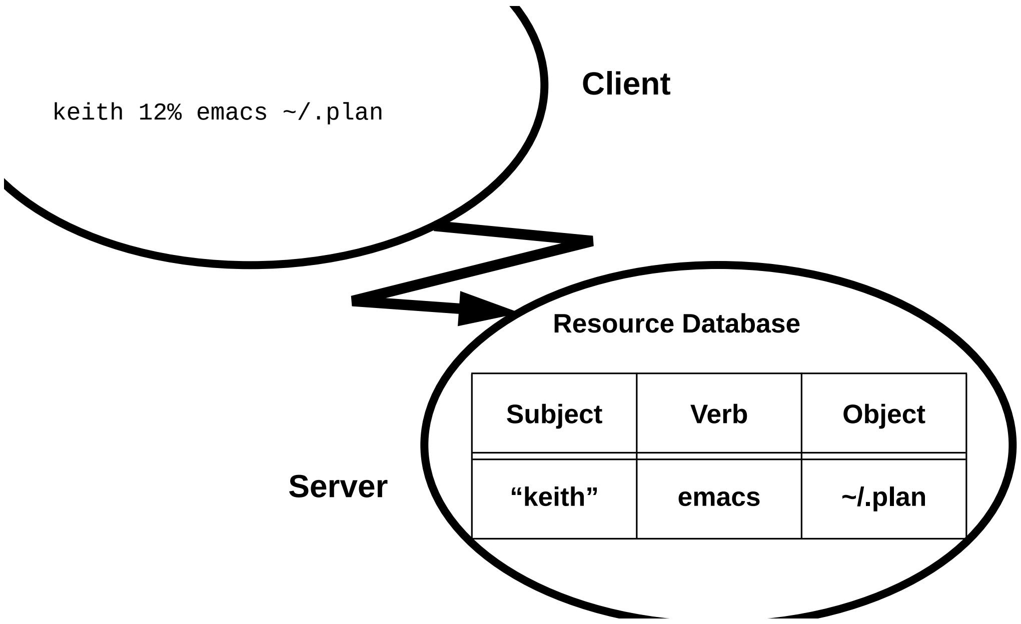


FIGURE 3-1 An Application Publishes an Activity Record

Activity Components in Detail

The Intermezzo awareness model is intended to provide information about user location, the current tasks users are involved in, potential for interruptibility, and presence of coworkers (either physically- or virtually-collocated). In large part, the degree to which the model satisfies these goals depends on two factors:

- The information associated with each resource.
- The conventions followed by applications and users for publishing activity records.

By pushing a large amount of information about user activity into the environment, the facilities provided by Intermezzo can satisfy a general array of awareness needs. A richer range of information means that more applications can benefit from the data collected by the awareness system. This section details the information present in the environment as represented by the resources used by the awareness system.

Activity Resources

The Activity resource captures the notion of a single user engaged in a single task on some particular artifact. Activity resources serve as containers for the individual components of an activity: Subjects, Verbs, and Objects. The individual attributes of an Activity resource are links to instances of these resource types, which are described in more detail below.

Table 3-1, "Activity Resource Attributes," provides a synopsis of these attributes.

TABLE 3-1 Activity Resource Attributes

Attribute Name	Description
Subject	A link to the Subject component of the activity.
Verb	A link to the Verb component of the activity.
Object	A link to the Object component of the activity.

Subject Resources

Subject resources represent single users of the system. The slots in a subject resource constitute a "single-source" repository of information about a user, which can be used by a variety of applications. Applications and other users (via client programs) are free to access and, modulo authorization constraints, update slots in the subject resource.

By capturing and representing as much information about users as possible, subjects serve as a contact point for storing and retrieving general user information. Programs such as directory lookup services, "finger" utilities, electronic mail address books, and the like can be implemented by accessing slots in the subject resource.

The slots present in the resource are extensible; that is, applications that need to associate new data with users can add slots as needed. By convention, the slots in Table 3-2, "Subject Resource Attributes," are always present.

TABLE 3-2 Subject Resource Attributes

Attribute Name	Description
Login Name	The string used by the system to identify the user.
Real Name	The user's given name.
Home Directory	The user's home directory in the filesystem.
Shell	The user's login command interpreter.
Office	The user's office location.
Calendar	The name and host of the user's calendar.

TABLE 3-2 Subject Resource Attributes

Attribute Name	Description
Work Phone	The user’s work telephone number.
Home Phone	The user’s home telephone number.
Fax	The user’s fax telephone number.
Home Host	The user’s “typical” login host (the machine where the user can be found most often).
Email Address	The user’s electronic mail address.
Organization	A string identifying the user’s affiliation.
Plan	A short string describing the user’s goals (based on information provided by the UNIX `finger` program).
Project	A string describing the user’s current project (based on information provided by the UNIX `finger` program).
Face	A graphical image of the user’s face.
Voice	A short sound clip of the user’s voice.
Activities	Back-links to the activities containing this subject resource.

The availability of these slots is guaranteed when the Intermezzo toolkit is used to generate subject resources. Note that the *Activities* slot provides a reversible link from the subject resource back to the activity resources that contain it. From the activity resource, current tasks and objects may be derived. Thus it is possible to generate a complete list of current applications and open files given a particular user.

Note that the slots in the subject resource represent fairly static data. With the exception of the activities slot which, by convention, is updated whenever new activities are created, the data maintained by subjects represent the non-changing attributes of a given user. The slots in the user object are typically modified by the user via a “controller” application designed specially for this purpose.

For this reason, and also because of the desire for subjects to represent a single point of access for information about users, subject resources are persistent by default. While persistence has a number of uses, it is particularly useful in the case of subjects; in fact the need for persistent subjects was the main reason that persistence was added to the Intermezzo object model. Without persistence it is impossible to refer to a user who is not engaged in any activities (that is, is not logged on).

Activity records maintain IDs (essentially handles or pointers) to the component resources within it. Typically there will be only one subject resource for a given user. The activity record will maintain a pointer to the one representation of that user. This ensures consistency and ease of location for information about users.

Other resources in the activity record (verbs and objects) are not persistent by default, since these represent transient activities in the system. Human beings, in contrast, are by-and-large long-lived entities (at least in the time span represented by computer applications). Other activity record components *may* be made persistent, if there is a desire for activity history, or asynchronous coordination.

The information maintained by Subject resources is gathered from a number of sources. Much of it is available from operating system-level facilities: the user's login and real names, directory, and shell are all typically available (at least on UNIX-like operating systems). Other information (such as office, telephone numbers, and email address) must be explicitly provided by the user via an application specifically used to update such information on Subject resources. The Activities slot is maintained automatically by Intermezzo.

Verb Resources

Verbs represent single tasks in the system. A user engaged in multiple tasks will have multiple activity records, each containing a unique verb representing the task. Multiple verbs will be instantiated even if the user is running multiple instances of the same task.

Verbs capture information associated with a single active process in the system. Like subjects, the slots associated with verbs can be extended by applications for application-specific needs. The commonly associated attributes are agreed upon by convention and implemented by the Intermezzo toolkit however. These are presented in Table 3-3, "Verb Resource Attributes," below.

TABLE 3-3 Verb Resource Attributes

Attribute Name	Description
Application Name	The name given to the application program.
Start Time	The time at which the application was started.
Idle Time	Amount of time the user has been idle in this application.
Process Host	The network host on which the application is being run.
Display Host	The network host on which the application is being displayed.
Response	Used by the Intermezzo session management service (see Chapter IV for more details).
Activities	Back-links to the activities containing this verb resource.

Some of this information is static and can be determined at application start-up time (application name, start time, process and display hosts). Other information, namely idle time, requires application intervention to maintain.

The notion of "idle time" is different for different applications, and the mechanisms used to capture it vary as well. For example, on a character-based application, idle times can be determined via keyboard activity. In graphical applications, idle times may be determined via mouse clicks, or even mouse motion over an application. In some circumstances, controller applications can capture idle time information for some applications. In the character-based application example, a controller may monitor terminal idle states and update the idle time slot on the verb resource. In other circumstances, gathering useful idle time information by a process external to the task application in infeasible.

Object Resources

Objects represent the focus of a task: the “thing” on which the task is operating. Objects can represent an essentially infinite array of entities on which applications operate. Common types of objects include files, calendars, database selections, and so forth.

Because of the requirement for flexibility in object representation, the object resource is more complex than either the subject or verb resources. For any given task, the Intermezzo runtime library must be able to generate an instance of an object which uniquely identifies the thing on which the task is operating, no matter what the type of that thing is.

We require objects to have the property that there is a one-to-one mapping between an instance of an object resource and the physical entity to which it refers. That is, given an object, one can determine the physical thing it is modeling; and given any physical artifact, it is possible to generate a unique object for it.

The Intermezzo toolkit provides structural support for objects; that is, it provides a framework for generation, comparison, and retrieval of objects. Individual applications must implement a subsystem to generate unique objects for the particular datum type on which they operate. Since the object of a given application is domain-dependent, Intermezzo cannot know generally how to interpret a given artifact, or even how to generate a meaningful one-to-one-mapping to an object instance.

Intermezzo does provide support for generation of object instances for the most common artifact type: files. The Intermezzo toolkit can uniquely generate objects for any file anywhere in the network. Note, however, that because of particular application requirements, even the interpretation of files varies from application to application. Developers are free to interpose their own meaningful interpretations into the object generation subsystem. See “Granularity and Hierarchy in Objects” on page 56.

Many of the slots used by objects are domain-dependent, based on the type of the object. Hence conventions about the definition of these slots are left to the applications that use objects of that type. The generic object framework established by Intermezzo does enforce a number of slots that are present in all objects and are used for generation and comparison.

The “namespace” of an object defines a particular type of object. For example, files, calendars, and database selections are all represented by a different namespace. Within a given namespace, the name slot uniquely identifies the object as distinct from “siblings” which share its namespace (that is, they are different instances of things with the same type).

Note that the representation of object resources is one area in which true subclassing would be useful in the foundation layer. Currently different object types are represented by aggregation: the base object slots are present, and code which implements subtypes (new namespaces) is free to add new slots that may be appropriate for the appropriate subtype. True inheritance would make the implementation of object subtyping easier, but since object subtyping is the only use (so far) for inheritance, it was left out of the foundation layer.

The common slots in the generic object framework are represented in Table 3-4, “Object Resource Attributes,” below.

Comparison of object resources requires a comparison of both the namespace and the name. The namespace ensures that no “apples and oranges” comparisons are performed (files and calendars for example); names ensure distinction within a particular type of object (“file1” and “file2” for example).

TABLE 3-4 Object Resource Attributes

Attribute Name	Description
Namespace	The type of the object, represented as a string.
Name	A domain-dependent identifier for the object, valid within its given namespace.
Printable Name	A human-readable string which can be used to identify the object.
Implicit SM	Used by the Intermezzo session management services (discussed in Chapter IV in more detail).
Activities	Back-links to the activities containing this object resource.

The printable name slot is a representation of the object that would be understandable by a human being. For a file, the printable name would represent the full pathname of the file; for a calendar, it would represent the owner of the calendar.

An Example: Files

Simple files are the most common type of artifact used by applications, and thus have “built-in” support provided by Intermezzo. We now examine how the runtime system generates unique names for objects in the *File* namespace.

The naïve approach to constructing one-to-one mappings between file object names and actual files is to use the pathname of the file. This approach is insufficient since it does not allow the possibility of symbolic links or aliases in the file system. Aliasing breaks the one-to-one requirement because the same physical file may have different names bound to it. The simple approach also breaks down when remote or networked filesystems are in use. Two different files on different machines may have the same filename. Thus the one-to-one requirement is broken here as well.

Intermezzo uses a name generation approach similar to that used by COeX [76] and Tooltalk [54][109]. We use a tuple that represents the physical machine the on which the file resides, the filesystem on that machine, and then a unique per-filesystem identifier (on UNIX systems this corresponds to an inode).

$$Name = \{Host, Filesystem, Identifier\} \tag{EQ 3-2}$$

The Intermezzo client-side library can generate these tuples for any file, and guarantee a one-to-one mapping between names and actual files. The one-to-one property is maintained since the name of the object takes the network into account (by representing the host on which the file resides), and also circumvents simple aliasing (by representing the file as a unique node within a filesystem, rather than its path name).

Granularity and Hierarchy in Objects

One consideration when generating a resource that represents the focus of a task is that different applications may have different notions of what component of that focus is relevant. For example, consider a user editing a file using a drawing tool. Other applications may consider any of a number of characteristics of the object (the file) salient. For example, a tool designed to track usage of network file servers may only be interested in the machine on which that file resides. A project manager tool may need to track when changes are made to any files within a certain directory (presumably representing the work associated with a certain project or group). Tools specifically built for collaboration may have an even finer-grained focus: a collaborative drawing tool may consider the particular figure within the file currently being manipulated a salient feature of awareness.

Because of the fact that different applications may consider any number of attributes of an artifact salient, simply representing objects as single entities in insufficient. We need a scheme that allows us to represent the artifact at a number of granularities: from the machine the artifact resides on, to any containers the artifact may be embedded in (directories in the case of files), to individual components of the artifact that represent the user’s current focus (figures in the drawing, in our example). Objects are represented as hierarchies, with each layer in the hierarchy providing a view of the object at a different granularity. As we descend the hierarchy, features hidden in the “large-scale” view become apparent.

In many cases, layers in the hierarchy will be domain-independent. For example, many applications manipulate files. An infrastructure that can generate a set of objects representing the host the file resides on, and the directory components of the path to the file, as well as the file itself, will be generally useful to a number of applications.

Other layers in the hierarchy will be domain-dependent. Practically anything below the granularity of “file” will only be understandable by applications that have the domain knowledge to interpret the object. For example, the notion of figures within a drawing only have meaning to drawing applications. Similarly, specific appointments within a calendar are only meaningful to calendar applications; other applications do not have the domain knowledge or the infrastructure to interpret such constructs.

The Intermezzo model allows objects to be represented as a list of resources that model the particular artifact at a number of granularities. The toolkit can provide support for a number of domain-independent layers of the artifact hierarchy: namely files, directories, and machines. Applications have the responsibility for providing support for new layers of the artifact hierarchy that represent domain-dependent views of the artifact (the toolkit *cannot* provide this support, since it is domain-dependent).

Table 3-5, “Example of Object Hierarchy,” shows an example of an object list representing the hierarchy used to model a particular artifact, in this case a figure in a drawing. The first column denotes the artifact,

TABLE 3-5 Example of Object Hierarchy

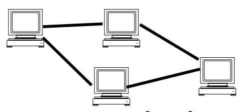
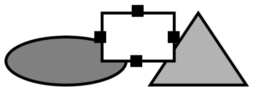
Artifact	Description	Domain
 **cc.gatech.edu**	Network domain of the host system.	Environment

TABLE 3-5 Example of Object Hierarchy

Artifact	Description	Domain
chagall	The machine on which the file resides.	Environment
/	Directory component of path.	Environment
tmp	Directory component of path.	Environment
paper.fm	Document.	Environment
 figure 3-7	Figure within file.	Application

viewed at a particular granularity. The second column provides a description of the artifact, viewed at this granularity. The third column represents whether the view is domain-dependent (that is, it must be generated by the applications) or domain-independent (it can be generated by the environment). The example shown here uses a number of layers to represent the artifact: the network domain containing the host the artifact resides on, the host itself, the components of the path leading to the file containing the figure (there are two of these), the document containing the figure, and the figure itself.

Note that there are any number of possible additional layers in this hierarchy. Coarser-grained interpretations are possible, as are domain-specific intermediary layers representing constructs such workgroups and projects.

The hierarchical representation of artifacts provides information needed by applications to interpret user activity. Applications can decide at what granularity a given artifact is relevant to them, and interpret the artifact at that granularity.

Evaluation and Comparison with Other Awareness Models

Our evaluation of the efficacy of the activity-based approach to modeling user awareness centers around determination of how well the approach satisfies the components of awareness discussed at the beginning of this chapter.

Location

The Intermezzo model of awareness is only capable of providing information about “virtual” location of the user. That is, what network hosts is the user logged in to, where is the display of applications being sent, and so forth. Fortunately, virtual indication is often a good approximation of physical location.

In situations where strong guarantees about physical location is a requirement, the Intermezzo approach can be augmented with information from other sources. See “Comparison with Other Computer-Based Awareness Approaches” below.

Current Tasks

Information about current tasks in the computer environment is completely provided by the Intermezzo model. Given a Subject resource that represents an individual user, the Activities link can be followed to list all of the tasks the user is engaged in.

The maintenance of task information is, of course, dependent on cooperation by the applications running on the system. The applications themselves are the sources and maintainers of activity information, and without their cooperation task information will be incomplete or out-of-date.

Potential for Interruptibility

Potential for interruptibility is a weakness of the Intermezzo model. This information must be inferred from other, more reliable, data such as the task list. Users can see that another is engaged in a high-priority task, seems to be collaborating with others, and so forth, and can use their own world knowledge to interpret this information in terms of interruptibility.

As stated before, interruptibility is a hard notion to capture mechanically. Other systems for awareness such as mediaspaces can provide some information about interruptibility by passing on details about the physical context of users. See “Comparison with Other Computer-Based Awareness Approaches” below for more details.

Presence of Coworkers

Much like location, the presence of absence of coworkers can only be interpreted in the virtual domain by the Intermezzo approach. Activity-based modeling can provide information about which users are co-engaged in some collaborative task, or running the same application, or working with the same data set. This information can be interpreted as “virtual” or “mental” copresence. But it cannot provide strong information about physical copresence.

Again, physical copresence (based on virtual location) can often be inferred. And, as mentioned previously, augmenting the Intermezzo approach with other mechanisms can provide stronger guarantees about the quality of copresence information, in situations that demand it.

Comparison with “Real-World” Awareness

Comparing the activity-based approach to awareness with the non-machine augmented, “everyday” awareness that we commonly experience with coworkers is instructive, because the comparison offers a common point of reference, and because it can help identify deficiencies in the activity-based approach.

The most problematic limitation with the activity-based approach is that information is not “free for the taking.” In the real world, users can simply look around to gather a rich array of information about user activities, locations, and so forth. Human beings are adept at determining potential for interruptibility, and other socially-influenced components of awareness. As Rouncefield, *et al.*, [91] state, “This ecology of the office provides, to those who know it, the ‘at-a-glance’ availability of what people are doing, what state they are at, how quickly they are getting the work done, and so on.”

In comparison, the activity-based approach used by Intermezzo requires extensive application cooperation to function effectively. In fact, awareness seems to be one of those situations in which nearly total “buy-in” is required for maximum utility (similar effects are seen in group calendaring applications, where virtually everyone in a work group must use the system without fail for it to function at all [48]). With the Intermezzo approach, applications must cooperate with the runtime system to facilitate awareness.

Interestingly, the requirement for application cooperation is not present in some other computer-mediated awareness systems. Mediaspaces, for example, do not have this requirement. There is only one stand-alone application which must be run to enable awareness. Users and other applications do not have to change their behavior to take part in the system, since mediaspaces essentially “transplant” the remote physical space (along with all its associated cues) to the user. (But there are some important consequences of this fact; see “Notes on Awareness” below.)

The requirement for application cooperation seen in the Intermezzo model can be mitigated to some degree by the use of “controller” applications which actively collect data and update the resource database on behalf of other applications. In effect, controllers function as Intermezzo-aware proxies for other non-Intermezzo aware applications.

A second important limitation is that some user states are not well-captured by activity. For example, user moods cannot be modeled by this approach. Systems which are more like mediaspaces may be able to capture some of these complex user states, although even they still have problems (due to artifacts in the quality and range of video and audio for example).

A third limitation is that the computer must still decide how to present and use awareness information once it has been collected. In the real world, collection and presentation of awareness information are synonymous, at least when performed first-hand: a user sees a coworker and is instantly aware of a wealth of information, in contrast to many computer-based approaches. The model provided by Intermezzo only specifies a structure for the collection and representation of awareness information. It does not specify how this information will be presented to the user, or even when or if it will be presented to the user. Again, applications must make intelligent decisions about the presentation of awareness for the information to be usable at all.

A final limitation is that, as discussed previously, some of the components of awareness can only be approximated. For example, physical location must be gleaned from where applications are running (the location of application displays is probably a good indicator of the location of their user, but is still only an approximation). Potential for interruptibility must be inferred from human knowledge about the activities a user is involved in (“Beth is editing the budget; I’d better not bother her”), and from idle times and copresence with others.

Comparison with Other Computer-Based Awareness Approaches

Comparison with other computer-mediated awareness systems is also useful. While the activity-based approach has certain limitations (and benefits) with respect to “real world” awareness, we shall see that it has a different set of strengths and weaknesses when compared against other computer-based schemes.

Two common approaches to computer-supported awareness have been investigated in the literature. The first and most common is awareness through computer-mediated audio and video. The second is the use of active badges. We investigate how the activity-based model compares with these two computer-based systems. Each approach has its own strengths and weaknesses relative to the activity-oriented scheme used by Intermezzo, and yet (as we shall see) these approaches can be integrated together to complement one another.

Video Monitoring

These systems, generally characterized as “mediaspaces,” support awareness of the locations and actions of others through a near-constant video presence (Media Space [5], Portholes [28], CAVECAT [62][97], and others), and perhaps “environmental” audio cues (RAVE [37]). Some systems go beyond the simple connected-office approach to provide a locality-based metaphor for interaction (Cruiser [87], Jupiter [15]). In these systems, awareness is regulated and filtered by copresence in a virtual space.

Video-based systems tend to provide excellent indications of physical location, as long as the user is in an area equipped with a camera. The cost of equipping spaces with cameras may be prohibitive in some situations, however, so provision of information about physical location may be limited by this factor. Virtual location is not provided by video-based systems.

Information about current tasks can be provided in the case of physical tasks: the user is talking on the phone, talking with a coworker, starting out the window. Again, video-based systems provide few or no cues as to computer-based tasks. Similarly, the presence of coworkers is reliably indicated in the case of physical copresence only.

The information about interruptibility that can be provided by video-based systems is limited to that which can be determined based on physical location, physical tasks, and physical copresence.

The primary strengths of video monitoring approaches are based on the high-quality cues about the physical environment that they provide. These systems can go far beyond approaches which merely sample the virtual environment and must infer physical information from virtual cues; physically-based approaches can provide information about facial expressions (an indicator of moods), and physical tasks and position, all of which are extremely useful social indicators of awareness. The limitations of video monitoring result from the fact that almost no direct information about the virtual (computer-based) world is provided; it can only be inferred from physical side effects.

While the quality of physical cues in a video-based system far surpasses that available from activity monitoring, the data is not easily machine-representable. Activity monitoring can provide more information about virtual activities than video schemes, and this information is in a format that is easily understood by applications. Of course, this characteristic has a downside in that information that is machine-parsable is often not easily understood by humans. In the Intermezzo approach, applications have the burden of making activity information understandable. In mediaspace-like systems, the information is trivially understood by human users and there is little or no need for machine interpretation.

Active Badges

The second common computer-based approach to awareness is the use of active badges [113]. Active badges are devices worn by users which actively update a global location database as the position of the wearer changes. Rooms in the work area must be equipped with sensor devices to receive the data transmitted by the badges.

Active badges give accurate location information to the system, so much so that users of active badges have found it practical to route their telephone calls to the location provided by the awareness system. In fact, information about physical location is the only *direct* information provided, and is the original reason for the creation of these systems. Likewise, information about physical collocation of users is readily available with active badge systems. They typically can be used more widely than video-based systems, however, since the cost of transmitters and receivers may be less than the cost of a complete video infrastructure.

Information about tasks and interruptibility is not provided by active badge systems, other than information which may be indirectly inferred based on location and copresence. For example, the location of a user in a conference room or in a boss's office may give some indication of interruptibility and tasks.

An advantage of active badge systems over video monitoring is that the information about location is available in a convenient machine-parsable format. In comparison with activity-based approaches, however, active badge systems cannot provide virtual task and location information.

Summary of Comparisons

A summary of the three computer-based awareness approaches, video monitoring, active badges, and activity-based, as well as "real world" awareness, is presented in table Table 3-6, "Comparison of Awareness Approaches," below. The last row in the table indicates how amenable the approach is to automated machine-parsing. Some approaches generate data that is more easily represented and stored by computer; the aspect of parsability will be discussed more fully later

Together, the strengths and weaknesses of the activity-based approach along with those of the other approaches suggest that an augmentative approach to awareness may be effective. By combining multiple approaches better "coverage" and more robust solutions may be obtained. The integration of other awareness input sources can be accomplished by controller applications in the Intermezzo model.

TABLE 3-6 Comparison of Awareness Approaches

	Activity-Based	Active Badges	Video Monitoring	"Real World"
Location	◍	●	●	●
Tasks	●	○	◍	●
Interruptibility	◍	○	◍	●
Copresence	◍	●	●	●

TABLE 3-6 Comparison of Awareness Approaches

	Activity-Based	Active Badges	Video Monitoring	"Real World"
Parsable?	●	●	○	○

● = Feature Included ○ = Feature Not Included ◐ = Feature Partially Included

As an example, a controller that is messaged whenever the location database maintained by active badge monitors is updated can write authoritative physical location information into the location slots in subject resources. Augmenting the activity approach with active badges removes one of the limitations of an activity-only approach: true physical location can be known with certainty, rather than merely approximated by virtual location.

As another example, video systems used in conjunction with the activity approach can provide more information about physical copresence, physical location, and perhaps some other user states (such as moods) that are not captured at all through activity-only modeling.

Notes on Awareness

Many of the benefits of bringing awareness of users into the computer-mediated domain have been investigated in some detail by the research literature. Awareness information helps users to coordinate their activities better, and can result in higher work quality [27][66].

As we have seen, the activity-based approach to awareness is capable of bringing a wealth of information into the application domain so that it can be presented to other users. The approach, particularly when used in conjunction with other schemes, can be particularly effective at capturing information about computer-based activity and virtual location.

There are several other interesting repercussions implicit in the activity-based approach which may be non-obvious however. This section investigates these in detail.

Input to Applications, Not Just Users

In most awareness systems seen to date, awareness data has *typically* been used only for the benefit of users. For example, in most mediaspaces, activity is not captured in a machine-readable form. Rather, it is displayed as video and left for humans to interpret. The salient components of awareness are not directly represented in any computer-based form; they must be extracted by the users of the system from the video stream.

Some mediaspaces have recently begun to focus on machine-parsable representations. Where Were We [65] and Marquee [114] both provide machine-parsability by annotating the audio and video streams as they are created, or after they have been recorded. But in general, the goal of these systems is to make activity information easier to interpret for humans, rather than for applications.

A number of “activity monitor” applications have been built which are similar in principle to the awareness approach used by Intermezzo [61], but still these systems generate data for human consumption only. Typically the activity data is sent to a client application which presents it in a window on the user’s desktop.

In contrast, one of the goals of the activity-based approach used by Intermezzo is to make it possible for applications to behave intelligently in the presence of information about user activity. That is, just as users are aware of the presence and activity of other humans, so can applications be made aware of their users.

As an example, consider a computer-based teleconferencing application that could be made aware when one of its users is on the phone. The application could mute the audio signal from the conference, and notify other participants that the user is currently engaged in another task which demands auditory attention.

As another example, consider a collaborative editing system in which a user requests another to join the session. The editing tool could be written to analyze the potential collaborator’s location, other tasks, and collocation situation in an effort to determine if the recipient should be disturbed, or if the request for collaboration should be delayed.

Further, and as we shall see in later chapters, the Intermezzo environment itself uses awareness information to govern its own policy and access control mechanisms. The power that can be vested in a collaborative environment by bringing machine-parsable awareness information into it is considerable.

Policy and Access Control is Essential

One obvious concern about bringing awareness information into the environment is that the availability of such information is potentially a huge privacy violation [8]. Capturing (and potentially storing) detailed information about the whereabouts and activities of users requires provisions for strong security guarantees about access to this information.

Acknowledgment of the security risks inherent in awareness systems has been noted many times [56], particularly in the case of mediaspace systems [102]. In almost all mediaspace studies, there are a number of potential users who refuse to use the system even though (a) there is strong physical security (users can simply cover the camera), and (b) storage and machine processing of mediaspace data is not feasible with current technology.

Certainly the security concerns present in mediaspace systems are present in an activity-based awareness system. If anything, the concerns are greater since a machine-parsable representation of the information is readily available.

Thus, security and access control schemes are of paramount importance in a coordination support system that makes awareness data available to users and applications. Chapter V discusses high-level access control mechanisms in depth.

Persistence Adds Value

A third point to make about activity-based modeling is that persistence can significantly increase the power of the system. The activity-based approach is particularly useful for capturing the real-time aspects of awareness: the instantaneous condition of the collaborative world and the users in it. The ability to support persistent subject resources is necessary, however, and in fact was the primary reason for the inclusion of persistence in the foundation layer.

There are other uses for persistent awareness information that have not been fully explored in this research. Below we enumerate two of these.

Usage Statistics

By archiving activity information it is possible to compile long-term usage statistics about the activities of users. The information captured by the activity approach can be used to answer questions about resource allocation, work trends, and usage patterns of data and applications.

For example, an administrator can track application usage and work patterns to determine when to add resources to a network. Group work flow can be analyzed to isolate inefficiencies in work processes.

Organizational Memory

The ability to store and search activity data gives the potential for a form of organizational memory “on the cheap.” Organizational memory is knowledge which resides in a multi-person group (such as a design organization or an entire enterprise) about previous design decisions and the location of personal expertise in a workgroup.

Some systems such as Answer Garden [1] have provided machine support for organizational memory by archiving help desk-style question and answer sessions. Persistent activity information can give many of the same benefits, but across a wider spectrum of usage (many applications, many users, instead of the single help desk context).

For example, the awareness system can “remember” who has accessed documents and make this information available to other users. Such a system would enable a means of determining a set of people who are likely to have expertise about a document.

Chapter Summary

This chapter has presented a general model for machine-generated representations of the types of information useful for user awareness. This model is based on logging of information about the activities of users across the network, and is sufficient for capturing many of the essential attributes comprised by the notion of *awareness*.

Activity-based modeling of users can provide information about current tasks, location (especially “virtual” location), the presence or absence of coworkers (again, especially in the virtual sense), and can give some indication of potential for interruptibility. The information available in the activity-based model can be leveraged by other approaches to awareness to “fill in the gaps” in the information presented to users and applications. We have seen that one of the strengths of the activity-based approach is that it makes awareness information available in a format that is easily machine-parsable and machine-storable (unlike some other media, such as video). This characteristic makes activity-based awareness useful as input to applications, not only as input to users (which has been the traditional use of machine-augmented awareness).

We have presented an implementation model, based on the Intermezzo foundation layer, which can be used to capture, store, and disseminate awareness information. This model captures activity as a set of tuples representing users, their actions, and the objects of those actions. We have seen that an effective implementation of this model must take issues such as object granularity into account.

Finally, we have compared the activity-based awareness approach to a number of other approaches, including the types of awareness information we are presented with in the physical world. The information available through the activity-based approach to awareness has a number of applications beyond simple user awareness. These applications, most of which are unexplored by this research, include organizational memory and the collection of user statistics.

In the next chapters, we will use the information provided by activity-based awareness as a foundation for more powerful collaborative tools.

Chapter IV

Session Management

This chapter introduces session management as one of the coordination features supported by Intermezzo. We begin by examining the need for toolkit-based solutions to session management, and then introduce a taxonomy of approaches to session management. This taxonomy includes some forms of session management commonly found in collaborative systems, as well as a number of novel forms. Next we examine an infrastructure that is capable of providing support for the forms of session management detailed in this taxonomy; this infrastructure is built atop the activity-based awareness methodology described earlier.

Introduction and Definitions

The defining characteristic of collaborative applications is that they, by definition, involve the interaction of multiple users. The manner in which the users of an application join together into a collaborative endeavor is determined by the session management system used by the collaborative application.

The term *session management* refers to the process of starting, stopping, joining, leaving, and browsing collaborative situations across the network [77]. Examples of collaborative situations include a group of users editing a text document in a shared editor, or a video conference among a number of participants.

Currently, developers typically implement subsystems to perform session management on a per-application basis. This trend has resulted in three characteristics of current session management systems which are problematic:

- Much work has gone toward “reinventing the session management wheel” in collaborative applications. Application developers typically implement subsystems to perform session management each time they build an application. There is little cooperation among applications or code reuse between developers.

- Since session management itself is subordinate to the centerpiece task the application facilitates (group-editing a document, for example), the session management mechanisms in a particular collaborative application are often not very robust, flexible, or powerful. Usually the session management facilities provide the minimal level of functionality to allow the application to perform in a collaborative setting.
- Per-application reimplementation of session management typically means that there are no facilities for altering session management behavior on a global (across-application) scale. For example, it may be difficult to turn off incoming requests for collaboration across *all* collaborative applications running on a user's desktop.

Side effects of the current application-centered practice have limited the flexibility of session management systems in some ways that are non-obvious, however. Since application-centered approaches, by definition, tend to limit cooperation between applications (since there are no common facilities to build upon), it is very difficult to build session management systems that permit the easy flow of information between applications using these approaches. Put more directly, the types of session management systems that are easy to build in an application-centered approach may not always be the "best" from the perspective of the user. Building collaborative systems around non-cooperating session management facilities restricts the options available to us as developers of collaborative applications: the tools influence the resulting system. As the state of the practice has influenced session management systems, sessions management systems have in turn affected the characteristics of applications built using these systems.

This application-centered development model has forced a certain mode of operation on collaborative applications. Since the session management services typically haven't provided a great deal of coordination between applications, users have been tasked with performing much of the coordination required to begin a collaboration themselves. We shall look more closely at the models of session management used currently by most applications, and at new models that can be created when more environment-centered approaches are used.

This work investigates two major research goals related to session management. First, we introduce a taxonomy of session management systems that defines a set of determining characteristics of such systems. This taxonomy categorizes not only the forms of session management in common use today, but also covers some non-traditional forms that (as we shall see) can prove useful in many circumstances.

Second, we introduce a theoretical model for the implementation of session management services. This model supports not only the traditional forms in our taxonomy, but also some of the non-traditional forms. The implementation of this model in Intermezzo is powerful and flexible enough to provide an array of session management services to applications.

Session Management Paradigms

To motivate our discussion of models of session management, we must first explore how session management systems typically operate in current applications, and introduce some terminology.

Most collaborative applications built to date have taken a "heavy weight" approach to session management. Two of these heavy-weight approaches are common:

- **Initiator-Based.** Through some sequence of dialogs the initiating user invites other users to the collaborative session. The number of invitations issued can be potentially large, depending on the application and the context of the task. Invited users can accept or reject the invitation. Examples of such systems include MMConf [12], Quilt [60], RTCAL [46][93], and the UNIX "talk" program. Initiator-based session management systems fulfill two goals: to notify users of the existence of the collaboration, and to provide a means for rendezvous with the others in the session.

- **Joiner-Based.** The initiating user creates a new session; users must find the session by browsing the list of currently active sessions (or know *a priori* that the session will be taking place). Once they know the session handle they can attempt to join the session. Examples of systems which follow this model include Collage [71], Rendezvous [77], and GroupKit [90]. Joiner-based systems typically only provide rendezvous facilities. The participants use some other means to notify each other that a collaboration is in progress.

Both of these approaches are somewhat awkward because they force *someone* (either the initiator or the participants) to do a significant amount of overhead work. In the case of initiator-based session management, the initiator must explicitly invite all participants to the session. In the joiner-based case, the joiners are required to find the session handle for themselves. If they do not remember that they are supposed to be taking part in a collaboration, it again falls to the initiator to invite the participants to the session (typically via a "low-tech" solution: a telephone call).

We call these two approaches instances of *explicit session management*. We use the term explicit because the participants in the collaboration are required to take some action (perhaps time consuming) to join the session. This joining action is often only peripherally-related to the task on which the participants are trying to collaborate. For example, for a group of users who wish to work together on a text document, editing the document is the primary focus of the activity. The peripheral task of either inviting group members, or of searching for a "conference" is an artifact of the implementation of the session management facilities offered by the collaborative applications being used.

Explicit session management seems to be useful in situations where there is a high degree of formality or where there is a natural name for the activity. An analogous "real world" situation might be *Faculty meeting Wednesday at 2:00 PM in room 302*. The task is widely known to the participants, is at a well-known location, and embodies a degree of formality.

It is important to realize that explicit session management (and indeed, any form of session management) can affect the way people collaborate. Because starting a collaboration is expensive, informal collaborations are less likely to take place when using an explicit system. Perhaps because of the formal nature of explicit session management systems, collaborative applications based on this technique tend to resemble meetings.

Unfortunately, more spontaneous collaboration is not likely to fit well into this model. Rather than the faculty meeting form of collaboration, what about serendipitous meetings in a hallway or in a breakroom? Some systems have been built to support this form of "chance meeting," although they have typically been communication-oriented "mediaspace" applications [5], rather than, say, shared editors. Examples of this class of applications include Montage [110] and Cruiser [87]. Mediaspace applications usually bypass the entire issue of session management by leaving users connected to each other (or to a virtual location) all the time. This approach seems appropriate for communication-oriented systems, but may be less useful for other types of applications (we do not wish to require users to leave one of every potential collaborative application on their screens in the chance that someone may wish to collaborate with them using that application). We would like to build a framework for session management that works for non-communication-oriented applications but is still lightweight, supports serendipitous meetings, and is as transparent as possible to the user.

Such a system would not necessarily supplant the explicit forms of session management, but would instead serve as a complementary form that might be useful for a variety of tasks which explicit session management might hinder.

A Light-Weight Model of Session Management

There are a number of situations in which a more light-weight, *implicit* form of session management that requires less initial overhead may be useful. We can see that in many situations invitations are not needed. For example, explicit computer-based invitations are often not needed if the collaboration is within a small group of people who know that they are supposed to be working on some collective task. Often, social pressures will serve to keep uninvited participants out of the collaboration, without the need for strict machine-enforced invitation protocols. And of course, serendipitous collaboration doesn't require invitations by its nature. Also, the requirement of joiner-based systems that they be named and created in advance is a detriment to systems that must support light-weight or very informal collaboration.

But without invitations and naming, how can we accomplish rendezvous? What can we do to make the process of joining the collaboration as light-weight as possible? In the models we examined earlier, rendezvous was accomplished by some mechanism orthogonal to the task at hand (requiring the user to browse a list of conferences before being able to enter a shared editor, for example). A more direct approach would be to have the act of opening the object of the collaboration provide the potential for collaborative activity itself.

Thus, to collaboratively edit a file, users would simply edit the same file. The system would detect the potential for collaboration inherent in the fact that multiple users are working on the same object, without the need for naming sessions or browsing lists of sessions to accomplish rendezvous. We call this process *implicit session management* because it avoids the overhead of the explicit session creation, naming, and browsing phase. In contrast to the explicit forms of session management (initiator-based and joiner-based), this form of implicit session management is *artifact-based* [31].

There are physical analogs for this type of collaboration:

- In "old fashioned" libraries with paper card catalogs, a person would know if another individual was interested in similar subject matter by the proximity to them when browsing in the stacks. A potential (although not required) collaboration exists ("I see you're interested in sailing too"). (Thanks to David Gedye for this example.)
- In a medical office, one worker may know that another is updating a patient file because the file is not in its place in the cabinet (in many offices a paper slip is left in its place to denote who has the file in question). If desired, the worker searching for the file may go to the one who has it to share information.

In both of these cases, participants know that they're working on the same or similar tasks because they are interacting with the same physical object (files or card catalogs in the above two physical examples). Rendezvous is based on the sharing of this common object. Collaboration is easy, although not required.

Artifact-based session management has existed in a number of applications in a specialized form, even though it has never been categorized as a form of implicit session management. Examples of artifact-based rendezvous include SUITE [20], CoEX [76], and the ToolTalk system from Sun [54][109]. In CoEX and ToolTalk, the artifact used for rendezvous is a shared file within the filesystem; in SUITE, the shared artifact is the name of a persistent object. Thus, whenever two or more applications access the same data object, the collaborative applications are connected and a new session is started.

Other Forms of Implicit Session Management

We have seen that the artifact-based approach to session management provides an alternative to the traditional, explicit approaches that is light-weight, supports serendipity, and is transparent to the user (that is, they require the user to take no action orthogonal to the task of collaboration). But are there other forms of implicit session management which have not been identified in the literature?

One mechanism of user rendezvous which seems to meet the definition of implicit session management is used by systems that support a spatial metaphor. Examples of such systems include Cruiser [11][87], MILAN [49], and various MUD systems including Jupiter [15] and LambdaMOO [13][14]. In such systems, users are afforded collaboration by virtue of their being in the same virtual location (the same room in a MUD or in Cruiser's virtual office metaphor).

These are forms of implicit session management that I term *locality-based*. The forms are implicit because, like artifact-based session management the system does not require the users to take any action orthogonal to the task of collaboration to rendezvous with each other. Such systems have "built-in" rendezvous mechanism which is inherent in the spatial metaphor that the use: virtual co-location affords collaboration.

A Comparison of Forms

As previously stated, explicit session management forms are not necessarily bad, they are simply different than the lighter-weight, implicit forms. Both forms of session management have their own respective strengths and weaknesses.

Explicit forms of session management seem to be most useful in situations where the collaborative endeavor has a name readily associated with it (to make location of the session easier), or if the collaboration is well-known to all participants ahead of time. This requirement of *previous knowledge* is present because of the nature of the rendezvous systems used in explicit session management. The participants must either be able to name the collaborative endeavor, or name the other participants, in order to collaborate.

The previous knowledge requirement means that explicit forms are perhaps more effective when used in formal situations: the higher the formality of a particular endeavor, the more likely it is to be known to participants ahead of time. The explicit forms do support a number of session management metaphors: initiator-based session management has an obvious metaphor in the telephone call. Collaboration is initiated by one user who must know the target user or users ahead of time. Joiner-based forms support metaphors such as yellow pages and broadcast media. To "tune in" to a collaboration you must be able to find it. Further, there is no explicit invitation. It is up to the user to know when to attempt to join (when a favorite program is aired, in the broadcast metaphor).

Implicit forms are useful in situations where the activity is not readily named, where invitations are unlikely or unwieldy, or where the activity is not known to all participants ahead of time. Because of the lack of previous knowledge, implicit forms are better able to accommodate informal interaction and can even support the most informal of all collaboration: serendipitous interaction.

Common metaphors for implicit session management are almost "built in" to the session management forms themselves. In fact, this is why they support light-weight interaction so well: the act of rendezvous is "hidden" within the metaphor in use. Artifact-based systems support object-sharing metaphors, much like the card catalog or file folder examples in the physical world discussed earlier. Locality-based systems support a virtual copresence metaphor which is already common in MUD-like systems.

Table 4-1, “Comparison of Session Management Forms,” presents a synopsis of the comparison between these forms.

TABLE 4-1 Comparison of Session Management Forms

	Explicit		**Implicit**	
	Initiator	**Joiner**	**Artifact**	**Locale**
Named/Unnamed	Unnamed	Named	Unnamed	Unnamed
Onus on System/User	User	User	System	System
Supports Serendipity	No	No	Yes	Yes
Formal/Informal	Formal	Formal	Informal	Informal
Metaphor	Phone call	Yellow pages	Shared document	Copresence
Requires Shared Artifact	No	No	Yes	No
Requires Shared Locale	No	No	No	Yes

Activity as a Foundation for Implicit Session Management

We have developed a taxonomy of session management forms which encompasses not only the “traditional” explicit forms of joiner-based and initiator-based session management, but also two forms of implicit session management that have not been widely studied in the literature. Both implicit and explicit forms of session management require help from “the system” to operate effectively. The explicit forms require some form of naming service for session location and selection, as well as inter-application communication facilities to support messaging potential participants.

The infrastructure needs of implicit forms have not been widely considered however. This section presents a theoretical model for implicit session management, particularly as it is manifested in the artifact-based approach. The next section, “Implementation” on page 73, gives details on how this theoretical model is provided by Intermezzo to satisfy application needs in the implicit domain, as well as how Intermezzo provides naming and inter-application communication facilities to satisfy explicit session management needs.

It is my thesis that information about user activity can serve as a foundation for building a powerful and flexible session management service, with application beyond that of implicit session management only. Recall that activity information contains details of the current tasks that are being run across the network: the users on the systems, the applications or tasks they are currently engaged in, and the objects of those tasks (that is, the data on which the applications are operating).

Essentially, the information provided by the activity-based approach to awareness can fulfill the infrastructure needs of an implicit session management service. Recall that in the activity-based awareness system, applications that partake in the service publish activity records that may be represented as the tuple:

$$A_n = \{S_n, V_n, O_n\} \quad \text{(EQ 4-1)}$$

where A_n represents the n-th activity, which is comprised of user S_n, performing task V_n, on object O_n.

When performing implicit session management, the session management service will automatically detect potential collaborative situations and take “appropriate” action (as described to it by applications or by user preferences). There is no need for users to explicitly issue invitations or create sessions.

The session management service detects potential collaborative situations by looking for overlaps or confluences in the total set of activity information published by all applications across the network. When two activity tuples exist that contain the same object token (that is, when both the namespace and the name of two objects match exactly), then the session management service can take action to allow the users to enter into a collaborative situation.

For example, if two users edit the same file, the session management service can notify the users of this fact and allow them to easily enter into a “spur of the moment” collaboration. The mechanics of joining a collaborative endeavor closely match the human dynamics of collaboration. When two coworkers wish to work together on a paper, one will typically say, “Let’s get together sometime after lunch and finish up the budget.” No *formal* invitations are issued, and no name is given to the activity. Instead, the coworkers simply begin working on the budget at or about the same time. The action of working on the same budget implicitly carries with it the notion of collaboration. Whereas in the explicit forms of session management the burden of labor is on the users of the system, in implicit session management the system itself can assume the task of detecting and handling potential collaboration.

Note that this form of implicit session management, because it is so transparent, requires applications or the collaboration support environment to provide powerful mechanisms for policy controls to allow users to enable, disable, or otherwise alter the behavior of the session management service. Obviously we don’t want to automatically be thrown into a shared editor anytime we happen to open a file that another person has open.

The mechanisms for publishing and retrieving activity information, for generating unique object names within a namespace, and for taking appropriate action upon detection of potential collaboration are defined by particular implementations of this model. The section below describes how Intermezzo implements implicit, artifact-based session management as well as traditional explicit forms of session management using the model of activity information presented above.

Implementation

This section presents the implementation of both implicit and explicit session management services in Intermezzo. Intermezzo builds mechanisms for both forms on top of the foundation layer services described earlier. The actual APIs used by developers for accessing session management services are described in Chapter VI later in this work.

Implicit Session Management

The facilities for artifact-based implicit session management are built atop the activity and awareness conventions described in Chapter III earlier. Once the basic activity collection mechanisms are in place, implementing an artifact-based session management system requires only a few additional features. This additional infrastructure is provided by both client and server features that cooperate together to implement session management:

- A set of conventions that the client and server agree upon for indicating whether the client wishes to take part in implicit session management.
- Support in the server for recognizing confluences in the activity database.
- Client support for responding to server messages which indicate potential collaboration.
- Facilities for describing how the system will respond to potential collaboration (policy controls).

Each of these features is described below.

Conventions

Clients can decide for themselves whether they choose to participate in implicit session management services. Some clients may only participate in explicit session management services, and some may not be collaborative at all. Therefore, it is important that clients be able to communicate to the server to indicate whether or not they will participate in the implicit session management service.

If a client wishes to participate, it augments the activity record it publishes to indicate that the objects it is using are “active” to the implicit session management system. It does this by setting the `Implicit SM` slot on the object resource to `TRUE`. Once set to `TRUE`, the server will “notice” confluences involving this object. Objects which do not have this slot set to `TRUE` are essentially ignored by the server-side implicit session management system.

If at any time a client does not wish a particular object to be visible to the implicit session management service, it can change the value of the `Implicit SM` slot to `FALSE`. The server will detect the change and disregard confluences involving this resource.

Note that this approach interacts well with the hierarchical object model used to represent artifacts of collaboration. Typically, applications will flag all of the objects constituting the hierarchy to support collaboration. If an application does not wish to support collaboration based on any particular “view” (granularity) of an artifact, it can indicate this by setting `Implicit SM` to `FALSE` for that object in the hierarchy.

Note that applications can create their own domain-dependent layers in the artifact hierarchy to represent any attributes of the artifact that have meaning only to that application. Thus, a collaborative drawing tool could base implicit session management on the sharing of a single figure within a document, rather than on the document as a whole.

Server Support

At startup time, the server spawns a new internal thread that supports implicit session management. The service registers new trigger functions in the activity database to inform it of changes to the database that may be relevant to implicit session management.

Specifically, the server registers triggers for the following conditions:

- Object Creation
- Value Change of Implicit SM Attribute
- Object Deletion

The trigger for object creation is fired whenever a new resource of type Object is created. The trigger function causes the evaluation of the confluence algorithm to determine whether other clients participating in implicit session management are accessing the same object. The algorithm, in pseudocode, is as follows:

```
//
// The object is just created; install a new trigger function on it to
// indicate whenever its ImplicitSM attribute changes.
//
object.InstallTrigger(CHANGE, “ImplicitSM”)
//
// If the new object is not participating in implicit SM, ignore it.
//
if object.ImplicitSM == FALSE
   return

//
// Retrieve a list of all objects from the resource database which match
// the new object’s namespace and name.
//
otherObjects = resourceDB.SelectMatches(object)
//
// Iterate over them, looking at each candidate in turn.
//
for each candidate in otherObjects
   //
   // If the candidate is not participating in implicit SM, ignore it.
   //
   if candidate.ImplicitSM == FALSE
      continue

   //
   // Otherwise, we have a match. Determine the verbs (applications)
   // associated with each object by accessing it’s back-link to the
   // activity record. Perform the specific action required by that
   // client (described in the next section).
   //
   thisVerb = candidate.Activity->Verb
   otherVerb = object.Activity->Verb

   DoClientResponse(thisVerb, “Implicit SM Potential”, otherVerb)
   DoClientResponse(otherVerb, “Implicit SM Potential”, thisVerb)
```

This algorithm scans the resource database looking for objects which match the new object. If found, and both objects are flagged as participating in implicit session management. an application-specific function for the application associated with those objects are performed to indicate that a potential implicit session management situation exists. The particular actions available to applications upon detection of a potential are described in the section below.

Note that the first thing the creation trigger does is install a new trigger function on new resource so that code will be run whenever the object's `Implicit SM` attribute changes value. Per-attribute triggers must be installed on *particular* resource instances. Thus, the trigger cannot be installed until an object resource is actually created (see Chapter VI for a more complete discussion of how triggers are used).

The trigger for value change on the `Implicit SM` attribute executes an algorithm similar to the one for object creation. If the value of `Implicit SM` is changing from `FALSE` to `TRUE` (that is, the object will now be participating in implicit session management), the server iterates over any matching objects and performs the client response function for the applications of those which themselves have `Implicit SM` set to `TRUE`. If the value of the attribute is changing from `TRUE` to `FALSE` (that is, the object will now be "invisible" to implicit session management), then no action is taken. The object will be unavailable for future matches performed by the implicit session management algorithms, however.

The trigger function for object deletion again performs an iteration over the resource database. If any matching objects are found which have been participating in implicit session management with this object, then their applications are sent messages to indicate that the object being deleted is no longer available. The basic structure of the algorithm is the same as that for object creation.

Client Response

Applications can decide for themselves how they will respond to potential session management opportunities. Some applications may be fully collaborative and may take some rich action when potential collaboration is detected. Others may be only minimally collaborative. Still others may not even be truly "Intermezzo aware" (the activity records corresponding to such applications may have been published by a "controller" application which updates the activity database on the behalf non-collaborative applications).

The number of possible actions to be executed when a collaboration is detected is large, and include:

- Pop up a dialog indicating that another user is accessing the object.
- Send a message to a shared window system to bring the application into a sharing mode. This action may be useful for applications which are essentially non-collaborative, but for which users still want to have some degree of collaborative interaction.
- Start up a suite of audio/visual conferencing tools to assist in collaboration.
- Go into an application-determined "sharing" mode. This action would be appropriate for fully collaboration-aware applications.
- Do nothing.

It is clear that the software infrastructure for supporting implicit session management needs to be able to support essentially arbitrary functionality when a potential collaboration is detected. Some of the functionality can (and must) be implemented by the clients themselves: a special sharing mode, for example, depends on application semantics known only to the client. Others may be implemented by the server, or by other "special" applications that serve as collaboration brokers on behalf of other applications.

The Intermezzo client response model supports arbitrary responses in reaction to potential collaboration. The model allows the actions that occur to take place in either the address space of the server or the address space of the client.

Applications that are participating in implicit session management modify the value of the `Response` attribute on their verb resource to indicate what action should be taken when a potential collaboration is detected. Legal values for this attribute are of either type `Boolean` or type `String`. The table below describes the action the server will take when the implicit session management algorithm described above detects a confluence.

TABLE 4-2 Implicit Session Management Client Responses

Value	Meaning
TRUE	Generate a message to the client indicating session management potential.
FALSE	Do not generate a message to the client.
“string”	Run the server-embedded code function named by “string.”

Clients have two broad classes of actions available to them. A boolean value of `TRUE` indicates that the server should simply generate a message to the client informing it that a confluence has been detected. The format of this message is fixed by the server and contains details on the object and the peer client that caused the confluence to occur. This response setting will be used by clients that implement specific internal functionality for handling collaboration (a shared editing mode for example).

A boolean value of `FALSE` indicates that no message should be generated to the client. This mode is rarely used, and is mainly useful only for applications that are essentially non-collaborative and participate in the Intermezzo activity and session management services in a minimal way only. Such applications, however, may desire that other applications be made aware of their use of a given object. The `FALSE` client response value allows an application to be “matched” in the session management algorithm (which will cause its peer to be notified of a shared object access), but without requiring the non-collaborative application to implement any special functionality.

Clients can also provide a string that names a code segment within the server to be executed. Clients can download code segments into the server to be run inside the server’s address space. Code downloaded into the server can provide essentially arbitrary functionality without requiring extra client intelligence. Examples include starting audio/video collaborative tools, or generating dialog messages. Code segments can also generate messages to clients, and can perform behavior specific to the peer client. For example, a segment may determine that its peer is not collaboration-aware and take a different action than it would for a collaboration-aware peer. If the code segment named by the response attribute does not exist, then an error message is generated back to the client.

Policy Controls

The notion of policy captures (among other things) the possible actions that clients may take when there is the potential for collaboration. Some users (and by extension, their clients) may only wish to participate in implicit session management activities with certain other users. The model presented above treats all clients as equals, in the sense that all clients which are accessing the same objects and are participating in implicit session management will be messaged when a potential exists.

This is often not desirable from the perspective of users. A given user may only wish to be notified of session management potentials when the other party is a member of his or her immediate work group. Problems of asymmetry also arise, where one user may wish to collaborate and another does not. The user who does not wish to collaborate should have the ability to keep the first from knowing that he or she is accessing some shared object.

Policy controls of this sort are a specific application of the general policy subsystem used by Intermezzo and described in Chapter V following this one.

Explicit Session Management

Explicit session management requires fewer services in the runtime environment than does implicit session management. In fact, the implementation of an explicit session management system requires only two mechanisms external to applications to support client needs:

- A naming service for locating sessions.
- A notification service for communicating with users.

A naming service is required to implement the session location and browsing functionality needed by joiner-based session management forms. A notification service is required by initiator-based forms to locate and generate messages to on-line users [30].

Note that these are only the facilities provided by the infrastructure; applications must still decide how to handle any domain-dependent session management functionality which may be required. For example, applications must implement whatever behavior must be executed when an explicit session management opportunity exists (when a user accepts an invitation, or when a user joins a current session). Like the infrastructure for implicit session management provided by Intermezzo, the explicit session management infrastructure only supports the coordination goals of detecting and facilitating rendezvous and connection. Applications are responsible for whatever domain-specific functionality must be provided to operate in a collaborative mode.

The support for explicit session management is somewhat limited in the current implementation of Intermezzo since (1) explicit session management has been studied in other literature, and (2) the activity-based facilities in Intermezzo provide few “value-added” services to explicit session management systems. More work could be done in the area of application conventions for describing how to conduct naming and notification, and supporting easier interaction between the facilities described below and application code.

Below we present the Intermezzo facilities for supporting naming and notification for explicit session management.

Naming Services

Joiner-based session management requires a directory service for locating sessions on the network, determining the active users in a session, and creating and deleting sessions. Intermezzo provides support for these operations on top of the foundation layer mechanisms.

A special resource type called `Session` is used for explicit session management. `Session` resources may be created, modified, and destroyed by applications that are participating in explicit session management.

These resources are used by applications to represent one active collaborative session on the network. The attributes of the resource are enumerated in Table 4-3, “Session Resource Attributes,” below.

TABLE 4-3 Session Resource Attributes

Name	Description
Session Name	The human-readable name of the session (presented to users via session browsers in joiner-based session management).
Activities	The list of activities constituting this session.

To create a new session, applications create a new resource of type `Session` in the shared resource database. Applications can search the resource space to locate existing sessions and read attributes of the `Session` resources corresponding to those sessions.

`Session` resources contain links to the activities that make up the session (that is, the set of users, applications, and data objects that are being used cooperatively to facilitate some joint work goal). By accessing these links, users can determine membership in a session; by updating the links, the membership of a session can be modified. Note that the standard access control model applies to attributes on `Session` resources just as it does for other resources. The creator of a `Session` can protect its membership by setting its access control lists accordingly. The policy mechanisms described in Chapter V can also be used to generate access control lists specifications for resources.

Note that the data model provided by Intermezzo makes it easy for clients to detect changes in the state of the session management world. By installing triggers for the creation, deletion, and modification of Session resources, applications can by asynchronously notified of changes to the session space.

When supporting explicit session management, the services provided by Intermezzo are only used to support data storage and searching functions. The extensive infrastructure available for supporting activity-based collaboration is not used.

Notification Services

Initiator-based explicit session management has a requirement for messaging users in order to invite them to a collaborative session. To notify a user of the existence of a session, the support infrastructure must provide facilities for locating users on the network, and for generating output to them.

Much like its support for naming, Intermezzo provides only facilitating support for notification. Applications must still decide how to use notification and implement the style of notification they desire.

To perform notification, a client application searches the resource database space for `Subject` resources which match the desired users. The `Subject` resource contains information needed to contact the user: idle time (to determine whether the user is in fact available), location, display host (the machine where the visual output of the user’s applications is directed), and so forth. Applications that wish to notify a user can do so by displaying a dialog to the user’s display host, generating audio output to their location, or by providing his or her telephone number to another user.

Again, the access control system can be used to restrict the ability of applications to conduct notification by regulating access to the relevant attributes needed by explicit session management.

Evaluation

This chapter has outlined a set of approaches to satisfying application needs in the area of session management. Intermezzo provides facilities for supporting both explicit and implicit session management paradigms. For explicit session management, the system provides naming and notification features that application writers can leverage to ease the task of participating in session management. Naming and notification are required for the explicit forms of session management, so their presence in a software infrastructure can speed the development process.

The features provided by Intermezzo to support implicit session management are much more novel, however. To address problems of implicit session management, particular artifact-based implicit session management, the runtime environment must provide a wealth of information. This chapter has shown that it is possible to construct a generic facility for rendezvous based on shared virtual artifacts by using the activity-based approach to awareness. Further, as the awareness system is extended to encompass new types of artifacts (objects), the session management system can automatically take advantage of their presence, detecting new potential collaborations.

The system provides a flexible set of conventions, server support, and client-side facilities for responding to implicit session management opportunities. These facilities are both easy to use by developers (as we shall see in Chapter VI, which discusses developer perspectives on the system), and capable of addressing a range of application needs for fine-grained control over client response.

Chapter Summary

Session management is an important part of collaborative systems: all collaborative applications must provide some mechanisms for bringing users together in the joint task. Yet too often, session management is an afterthought to application developers.

This chapter has presented a taxonomy of session management paradigms. We have investigated the “traditional” explicit forms (joiner- and initiator-based), as well as the more novel implicit forms (artifact- and locale-based). Each of these forms has its own particular strengths and weaknesses, and its own set of easily-supported metaphors for collaboration.

We have seen that activity-based awareness can provide a powerful tool for implementing “intelligent” session management services, particularly in the domain of implicit session management. By moving information about user activity into the environment, we can respond to ephemeral collaborative situations, thereby making the collaborative process lighter-weight, less intrusive, and more responsive to users. The models provided by Intermezzo also can be used for explicit forms of session management. The shared data framework and programming models provide a substrate on top of which naming and notification services can be constructed.

As we shall see in the next chapter, the session management mechanisms used by Intermezzo can be governed and regulated by a policy subsystem, which itself leverages the presence of awareness information in the environment.

Chapter V

Policy

This chapter introduces the notion of policy in collaborative systems by providing a number of “real world” scenarios typical in collaborative work settings. In Intermezzo, policy is defined in terms of access control. The access control model is sufficient to capture many aspects of real world policy, allowing the system to approach the fluidity and responsiveness required by human-to-human collaboration. The chapter outlines a set of formalisms for mapping from roles (descriptions of users) to policies (collections of access control rights). These formalisms are capable of producing extremely dynamic access control interactions at runtime. The implementation of the policy system, as well as a specification language for expressing roles and policies, is described, as are a number of common and useful policies.

Introduction

Collaborative systems are potentially chaotic environments: multiple users creating opportunities for collaboration, and rich (and unexpected) interactions between users and applications all contribute to the dynamic nature of collaborative software. The dynamism present in collaborative systems approaches (intentionally) the fluidity and richness of interaction of people in the physical world.

Collaborative systems represent a much more chaotic environment than typical single-user software systems. With single-user applications and environments, users engage in a “one-to-one” dialog with the application; users direct their attention and input to the application, and the application responds in (usually) expected ways. At least for experienced users, the style of interaction is almost choreographed; there are few surprises.

In contrast, collaborative environments present users with a potentially unbounded number of points of attention. The responses of a collaborative application are not as predictable as a single-user one, since the presence and input of other users of the application cannot themselves be predicted. Put simply, the presence of other users, with their own set of goals, desires, and experience levels, introduces the potential for uncertainty, unpredictably, and surprise into collaborative work sessions. Further, in collaborative systems there are many dimensions of unpredictability: user input, application-domain features, access control, rendezvous...all are made more complex by the presence of other users.

The presence of unpredictability within such a large and rich space of possible options indicates that there is a need for infrastructure-level features to “rein in” some of the unpredictability present in collaborative systems. If a collaborative environment throws up its hands and passes control to the user whenever the interpretation of a particular event may be made in several ways, users will very probably be overwhelmed by the task of “baby-sitting” the environment (accepting or rejecting requests for collaboration, modifying access control parameters), rather than getting work done.

Pushing intelligence about how to respond to particular occurrences in a collaborative situation into the infrastructure implies that mechanisms for describing and implementing *policy* must be provided by the infrastructure.

The American Heritage Dictionary defines policy as “a general principle or plan that guides the actions taken by a person or group” [2]. In a collaborative environment, policies govern the particulars of how users and applications interact with one another. A policy describes a general contingency against which specific events are evaluated and handled [67]. The goal of a policy system should be two-fold:

- *Reduce unpredictability in the system.* Allow the system to respond in expected ways to the actions of users and other applications, and support the principle of “least surprise.”
- *Require less effort from users.* Move the burden of dealing with most user and application actions from the user to the system.

We shall see some examples of policies in the next section.

Certainly the highly dynamic interactions supported by this research emphasize the need for policy support in the collaborative infrastructure. For example, the transparency of the implicit forms of session management require strong policy controls so that users aren’t constantly burdened by requests for collaboration every time they access a file. Also, the availability of awareness and activity information in the environment requires policies to govern who and under what circumstances access is allowed to the data.

In essence, a policy scheme is used to selectively limit the dynamism inherent in a collaborative system. The “trick” is to support a set of policy controls that can make a rich collaborative system manageable, and yet still preserve the essential components of dynamism that are necessary for human-to-human communication and interaction.

Policy As Access Control

The notion of policy is very broad, even in the non-technical usage of the word. How can we devise a formal system that allows us to capture the very general set of controls that users may wish to place on their environments?

It is useful at this point to present a set of policies with which to drive the presentation of the formal policy system. The policies below are defined in common, everyday language and are immediately obvious as policies in the non-technical sense of the word. Further, they represent common desires of users and have obvious utility in collaborative settings. If we can capture these sorts of common, easy-to-express policy desires in our system, then we will be close to supporting the rich forms of policy usage that we see in the workaday world.

- *"I don't mind people in my workgroup knowing what I'm working on, but others..."*
- *"Don't let anyone bother me when I'm working on my thesis. Unless it's my advisor of course."*
- *"I don't mind sharing my workspace with others during demo day."*
- *"If anyone calls for me, tell them I'm not here unless they've called with the new budget numbers."*
- *"If I'm requested to join a collaborative session, notify me with an auditory cue if it's Beth, or a pop-up dialog if its anyone else."*

All of the scenarios above are examples of policies which are common in everyday use. Users want the ability to regulate access to their information and personal space, and to govern how the system will respond to events that occur.

One of the contributions of this research is that *a wide array of policies like the ones above can be defined in terms of access control rights on data objects*. While perhaps not all policies can be captured by an access control-based model, many of them can be. Of course, making such a scheme broadly applicable to policy concerns requires that we wisely choose not only the access control primitives we will support, but also the granularity of the data objects to which we will be restricting access.

Several of the scenarios above represent what might be termed "first order" applications of access control rights to policy. There is a direct and obvious mapping from the policies, as defined in simple language, to their implementations in terms of access control. For example, in the first scenario, "I don't mind people in my workgroup knowing what I'm working on, but others...," the data object is "what I'm working on", and the access control action restricts access to a set of users. In first order policies there is a simple translation between policies and their implementation.

Other scenarios cannot be so obviously mapped onto access rights however. Consider the user preference example above, where the user expects the system to behave differently based on who is contacting him or her. Creating a policy system based on access control rights that can support such scenarios requires careful selection of both the access rights and the data objects in the system. If we can create data objects that serve as *proxies* for real world artifacts and capabilities then we may be able to implement such a range of non-first order policies like the above example.

To be an effective approach to supporting policy, our access control-based scheme must be able to (1) implement an array of non-first order policy scenarios, like the ones above, and (2) must be able to fit into a richly collaborative, interactive environment which is characterized by rapidly shifting actions and user roles.

The next section lays the groundwork for a policy system, based on our foundation-layer access control primitives, that provides a fluid, flexible, and extremely powerful model for implementing policy in a collaborative environment. This policy system is able to represent the scenarios described above, as well as others, by using a simple and consistent model of access control to data objects.

Supporting Fluid Policy: A Requirements Analysis

Human activity is a highly dynamic medium, rich in subtle interaction and constant shifts in the focus, priorities, and roles of the participants. As Moran and Anderson [68] state, “fluidity is a fundamental feature of work activity, and we need to be attuned to how technologies of various kinds can play a role here.” To be useful (or even usable) in such a setting, a policy system for collaborative applications must be able not only to respond to such changing interactions, but must also be able to support a form of policy that does not restrict the interaction artificially. An overly static policy system will serve only to cripple the dynamism which is inherent and beneficial in unconstrained human interaction. Policy systems must not only allow, but *support* the fluidity of interaction seen in “real life” human situations.

What work has been done to-date in policy systems for collaborative applications? It is instructive to look not only at policy controls for systems with a coordination focus, but also at controls for application-oriented systems and policy and access control schemes which exist outside the focus of collaboration.

The research literature has a number of examples of policy control in application situations, but only a modest amount of work in the coordination domain. Many systems use roles to generate application-level access rights, particularly in shared editors such as ShrEdit [74]. Work by Shen and Dewan on application-level access control has resulted robust model of the access control needs of applications [98]. In the coordination domain, many mediaspace-like systems provide some form of access control over the shared audio/video stream (see below).

Some general trends in these systems are clear. First, in the application domain, work on policy and access control is powerful and flexible. It supports fine-grained access control to allow control over objects such as segments of text in a shared editor. Application-domain policy support must also be very fast since it is evaluated nearly continuously as users to interact with the application. Some of the properties of application-level access control systems may make them inappropriate for coordination needs however: coordination may require access control that is not as rigid, more affected by situational influences, and supports more ephemeral changes of access control rights.

The second general trend is that in the coordination domain, policy support tends to be an “all or nothing” affair. A common metaphor is the “closed door.” CaveCAT [62], DIVA [103], Montage [107], and Cruiser [87] all provide a policy system in which users can allow or disallow access globally. It is impossible to specifically grant or refuse access to individuals or classes of individuals. Some of these systems expand incrementally on this model by providing several categories of access restriction (door open, door closed, door ajar, and so on). Only recently have some coordination-oriented systems begun to look at more novel and fluid support for policy (the ability to support “low-disturbance” forms of audio/visual interaction in Smith’s work for example [101]). Still, the general pattern we have seen is that weak policy support exists in coordination systems.

It is clear that for an access control system to be able to capture the wide range of needs expressed by the example policies discussed earlier, it must go beyond the rudimentary boolean access schemes typically found in coordination systems. Previous work in application-level access control schemes provides more flexibility and expressive power, but these systems were not designed for and have not been applied to coordination situations.

Many of the simple-minded or overly-formal access control schemes found in non-collaborative systems may be too restrictive for use in collaborative environments. Such systems typically support a fixed set of access rights to a fixed set of users and have no notions for the expressibility of the broad range of situations which can occur in collaborative settings.

By examining the scenarios for policy usage outlined earlier, we can begin to see some of the deficiencies with overly restrictive policy and access control systems.

- *"I don't mind people in my workgroup knowing what I'm working on, but others..."*

  This is an example of a policy which can be easily captured by traditional access control systems. The policy grants access to a set of information ("what I'm working on") to a constrained set of users which is long-lived and statically defined ("people in my workgroup"), and disallows access to others. A coordination system founded in any of the classical access control schemes (access control lists, capabilities, and so on) would be able to implement such a policy.
- *"Don't let anyone bother me when I'm working on my thesis. Unless it's my advisor of course."*

  Classical access control schemes would be able to identify "advisor" as a privileged user who should be allowed special access to the data. But traditional schemes typically support group membership that is changed infrequently if at all. Such systems would only be able to represent the instantaneous condition that this policy depends on (whether the user is working on the thesis) if the user explicitly modifies the access control settings when work starts or stops.
- *"I don't mind sharing my workspace with others during demo day."*

  Again, this policy depends on detecting some "real world" condition of the participants in the scenario.
- *"If anyone calls for me, tell them I'm not here unless they've called with the new budget numbers."*

  This scenario may seem gratuitous, but this is precisely the sort of thing that occurs regularly in the workplace. A user will delegate responsibility for access control to another human being. This delegate (a good secretary typically) must digest the salient situational variables and decide in real time whether or not to grant access. Today, users must delegate tasks such as this to other people since machine-based access control systems do not have the intelligence or situational awareness to act properly in such situations.
- *"If I'm requested to join a collaborative session, notify me with an auditory cue if it's Beth, or a pop-up dialog if its anyone else."*

  This is an example of a "non-first order" application of access control to a policy situation. In this case, we need to create our data objects that represent session management parameters in such a way that, by controlling access to them, we can achieve the desired result. This situation (and others like it) will be discussed more fully in the section "Policies and Session Management" on page 93.

In all of these cases, we can see that the presence of situational information about the state of the "real world" would be valuable in deciding whether or not to grant access to a particular piece of information. In fact, bringing situational awareness into the environment goes a long way toward changing a simple, low-level access control mechanism into something that can be used to implement general policies.

The characteristics of the scenarios described above indicate that a traditional system augmented with information about situational variables will provide a powerful infrastructure for describing coordination policy. This is the approach taken by Intermezzo: we combine the formality and speed of a classical access control system (specifically, access control lists as described in Chapter II in detail) with a scheme for sampling the "instantaneous" state of the users' world. The higher-level awareness system drives the low-level foundation access control system. The next section describes the implementation of a flexible set of policy controls for coordination.

Policy in Intermezzo

Policies are typically applied to sets or classifications of users. The literature has adopted the term *role* to indicate a particular category of users with a set of access control rights applied to them, and in fact many collaborative systems have found the notion of roles useful for delineating the application of policies to users [40].

A role is typically defined as a classification of a group of users within the user population of a particular application. All users in a certain role inherit a set of access control rights to objects within the application. A typically example of the usage of roles in the application domain is ShrEdit [74]. ShrEdit uses a number of roles to define the actions participants are allowed to take. Examples of ShrEdit roles include *editor* (allowed to change any portion of the document), *writer* (only allowed to add and modify text created by that user), and *commentator* (only allowed to add "margin notes" to the document).

Other systems which employ roles as a grouping mechanism for policies include ConversationBuilder [55], MPCAL [46], Quilt [60], Collaborative Editing System [74], SASSE [3], Colab [106], ICICLE [7], WORLDS [111], SUITE [98], and PREP [72].

A role provides a conceptual entity that can be mapped onto a policy. The notation:

$$R_a \rightarrow P_a \qquad \text{(EQ 5-1)}$$

Indicates that all of the members in role R_a will inherit the access control rights specified by policy P_a.

Intermezzo adopts the terminology of roles and applies the concept in a new direction. This section describes how policies are defined in Intermezzo, how they are applied to sets of users (roles), and how they are implemented in terms of the foundation layer access control policies.

Traditional Roles

Systems that use roles (which are typically application-oriented, rather than coordination-oriented) have a few characteristics in common in their role usage:

- The set of roles in use is determined *a priori* by the application or environment.
- The membership of those roles is typically determined early in the lifetime of the session.
- Membership is specified in terms of users.
- There are few (if any) changes to role membership during the course of the session.

In traditional use, roles are typically *static* in the sense that membership of a given role is established early and rarely changed. Further (and perhaps obviously) a role is defined in terms of a set of users who are members of that role.

Such static roles are useful for a broad range of applications, especially for the application-oriented aspects of collaboration. But they are perhaps not as useful for coordination. In static role systems, the set of users is predefined and typically fixed for certain access rights and the users themselves are responsible for updating role membership whenever policy mappings need to be changed.

In essence, users digest the situational variables that are salient to coordination and use their world knowledge to specify a set of access rights. Such a system places a burden on the users to "model" the state of the collaboration, and may impede the fluid flow of information that is essential to coordination. There are number of factors that limit the utility of static roles in coordination:

- Static role systems require explicit overhead on the part of the user to set up role membership. In fact, they require *anticipation* of potential coordination situations by the users.
- Static roles ignore situational dynamics ("the real world") in deciding group membership.
- Specification is rigid: there is no flexibility or "slack" in the system.
- Role membership can only be defined in terms of user names, not other attributes of users or the environment.

The problems related to predefinition of role membership have been identified in the literature. Neuwirth, *et al.*, have commented,

> *There is a potential problem in systems which support the definition of social roles: "premature" definitions of these roles could lead to undesirable consequences. For example, it is not always clear at the outset of a project who is going to make a "significant contribution" and therefore who should get [the] authorship [role]. But if authorship is defined at the outset, then it may reduce the motivation of someone who has been defined as a "non-author" and the person may not contribute as much.*
>
> *Despite potential problems, role specification is likely to be a useful strategy for managing some coordination problems.*

In [72], p. 185, Neuwirth, *et al.*, illustrate an problem arising from static role definition in the application domain. And yet, in the PREP editor, early-defined roles are used, presumably because of the lack of a role infrastructure or conceptual model to build upon.

Likewise, Dewan, *et al.*, in [23], point toward more flexible and fluid specification of roles as being a requirement for effective collaboration:

> *...users should be allowed to take multiple roles simultaneously. For instance, a teaching assistant should be able to simultaneously take the roles of "student" and lecturer." ...It should be possible to dynamically change collaboration rights, roles, owners, and ownership semantics. This requirement allows, for instance, a user of code inspection tool to graduate from an "observer" to an "annotator." ...Thus, like coupling and concurrency control, access control must be performed flexibly in the ways described above....*

We have seen that static roles are useful: they provide a conceptual model for mapping policies to users that is sufficient in most circumstances, they are well understood by developers and users, and are efficiently implemented atop common access control mechanisms. Intermezzo provides traditional, static roles for collaborative systems that can benefit from them. But static roles are limiting in many collaborative situations, particularly in the coordination domain. Of course, the limitations of static roles are not problematic only for coordination: many application-domain collaborative problems may benefit from a looser and more dynamic set of role mechanisms.

Dynamic Roles

Intermezzo also supports a more flexible (although more expensive and somewhat harder to manage) form of roles that do not have the limitations of traditional static roles. These roles are called *dynamic roles*. The defining characteristic of dynamic roles is that *membership in the role is determined at runtime, as requests for access are made*. This characteristic means that dynamic roles have an important power that static roles do not: instead of defining roles in terms of their members, roles can be described in terms of their attributes.

Membership in a particular role is not determined by a membership list; instead it is determined by a predicate function which is evaluated whenever an access request is made. The use of potentially arbitrary predicates to determine role membership lends great expressive power to dynamic roles.

By simply moving the determination of membership from session startup time to evaluation time, and by the use of predicate functions rather than membership lists, dynamic roles acquire several interesting properties:

- Dynamic roles allow role membership to be based on attributes other than “user name”.
- Potential membership can vary from moment to moment during the lifetime of a session.
- Access can be granted based on the instantaneous state of the user’s world.
- By describing role membership, rather than specifying it, users can be relieved of some of the burden of tracking, updating, and anticipating role membership explicitly.

As an example, via dynamic roles, you can not only specify “people who share my lab,” but “*people who are in my lab right now*” as a role or category of user.

Dynamic roles extend one of the common threads in this research: that by bringing information about user awareness into the collaborative environment, applications and the environment itself can be made more responsive to subtle changes in the state of the world. Further, users can be relieved of some of the burden of “manually” understanding and responding to situational variables; the system can take over some of that work when embodied with the required facilities and intelligence.

Although dynamic roles build on awareness information, they also regulate the awareness and session management services: the system is a closed loop in the sense that all three coordination features not only build upon each other but also feed each other. The synergistic effect which can be obtained by these three aspects of coordination used together is one of the contributions of this research.

Implementation

This section describes the policy services provided by Intermezzo in detail. Intermezzo provides mechanisms for creating both static and dynamic roles, an implementation of policies on top of the foundation layer access control system, and a specification language for roles and policies that can be used by users or applications to control and configure the policy subsystem.

A Specification Language for Roles and Policies

Intermezzo provides a software substrate that implements static roles, dynamic roles, and policies in terms of access control. The “view” of this substrate from the perspective of applications and users is a declarative language that is used to specify policies and roles. Users and client applications can create descriptions of the roles and policies which will be used.

At startup-time, the Intermezzo client-side code library loads a set of description files that contain specifications of policies and roles, and establish a set of role-to-policy mappings. These specifications govern the access rights which are granted to resources created by that client. The predicate functions used to describe membership in a dynamic role are expressed in an extended version of Python [112].

Specifying Roles

The Intermezzo role specification language supports three types of role specifications. Each role has a symbolic name associated with it that is used to bind the role to policies.

- Simple
- Dynamic
- Aggregate

Simple roles map a certificate name to a symbolic role name. These simple specifications are used to denote static roles. Note that certificates are consistently used to represent users throughout Intermezzo:

```
role keith = “Keith Edwards <keith.edwards@cc.gatech.edu>”
```

Dynamic roles declarations bind a predicate function to a symbolic role name. Predicates are specified by providing a full path name on the client to the location of the code file that implements the predicate. Predicates are written in an extended version of the Python language:

```
role demoday = [/users/k/keith/.policies/demoday.py]
```

Aggregate roles provide a mechanism for grouping other roles, whether simple, dynamic, or aggregates themselves. Aggregate roles can be nested arbitrarily deeply in a directed acyclic graph.:

```
role multimedia = {keith, beth, iansmith, demoday}
```

There is one role namespace within each parse-unit of the specification language. Note that inheritance of roles may be a useful feature. Inheritance would support the definition of classes of users, with specialization through derivation. This feature has not been explored in Intermezzo however.

Specifying Policies

Policies are specified declaratively and, like roles, are named. A policy specification consists of a set of access control specifiers that denote the access rights that will exist for the set of resources, and resource attributes, covered by that policy.

For purposes of assigning access rights, attributes are named by their key (the name of the attribute as represented by a string). Resources are named by type. As discussed in Chapter II there is a separate set of access rights for attributes and resources. The allowable access rights have human-readable names and are bound to the attribute or resource they correspond to. Each data object (either resource or attribute) can have default or fallback access rights associated with it, which will be used if no more specific access right is provided.

Below is an example of a simple policy specification:

```
policy Restricted {
   resource Subject {
      attr Name = AREAD
      attr Location = AWRITE
      attr * = ANONE
   } = REXIST
   resource Verb = RNONE
   resource * = REXIST
}
```

This specification creates a new policy named `Restricted`. The policy provides two by-type resource specifications for `Subject` and `Verb` resources. `Subjects` are associated with the `EXIST` access right for any users evaluating to this policy; `Verbs` have `NONE` rights. A wildcard specifier denotes that this policy associates `EXIST` rights with all resources with types not explicitly named.

The policy also provides two attribute access specifiers for attributes with keys `Name` and `Location`. `Name` attributes have `READ` rights; `Location` attributes have `WRITE` permissions. Note that since the attribute specifiers are nested within the specification for resource `Subject`, these rights will only be used for `Name` and `Location` attributes which are present in `Subject` resources. The wild card specifier indicates that any attributes in `Subject` resources that are not explicitly named by the policy inherit the `NONE` right.

Mapping from Roles to Policies

The Intermezzo specification language provides constructs for defining new roles and policies, and also provides a means for establishing mappings between roles and policies. Symbolic role names are mapped to symbolic policy names; the system supports many-to-many mappings.

```
multimedia      ->    PermissiveAccess
gvu_lab         ->    RestrictedAccess
animation       ->    RestrictedAccess
friends         ->    GeneralAccess
stasko          ->    AdvisorAccess
*               ->    Anonymity
```

Note the use of wildcard specifications for roles: the specification language allows a default policy that will be applied to all users not represented by any of the provided roles.

Many-to-many mappings allow several roles to share the same policy. In the example above, the policy `RestrictedAccess` is reused by two different roles: `gvu_lab` and `animation`. In the other direction, many-to-many mappings allow multiple policies to be associated with a given role.

Commonly, in this case each policy would specify a set of access rights for a different set of objects. Thus there would be no “overlap” between the access rights specified by the policies that are associated with a role. A second reason for allowing multiple policies to be bound to a single role is to reduce confusion that may result from aggregation. With aggregation, a given user may be in multiple roles simultaneously. The presence of the user in multiple roles may not be immediately clear because of either aggregation or dynamic roles. Since the system is well-behaved in the face of multiple policies associated with a given role, the presence of a user in multiple roles, each bound to perhaps several policies, will not present a problem.

If a user is in multiple roles simultaneously, Intermezzo uses a liberal policy for assigning effective access rights: the strongest right allowed by any policy of which the user is a member is applied.

Static Role Implementation

The notion of static roles is constructed by aggregating features found in the Intermezzo foundation layer; static roles are implemented using simple access control lists. In fact, the notion of a “role,” whether static or dynamic, is present only in the higher-level coordination features provided by Intermezzo (the parser for the specification language, for example).

When an Intermezzo client is run, the role specifications are parsed at application startup time. The client-side toolkit “digests” these specifications into a compiled format that specifies the access control lists that resources and attributes will inherit. This processed format is essentially an inverse of the role (user) to policy (access control) specification supported by the specification language. The system builds a list of all of the resources and attributes that are specified in the policy descriptions. It then “works backwards” to find all roles that map to policies that mention these resources and attributes. From this information it is possible to construct a set of prototype access control lists. These prototypes are mappings from resource types or attribute names to the access control lists that will be associated with those resources or attributes when they are created.

This mapping is maintained internally by the client-side library. Whenever the application creates a resource, or an attribute on an existing resource, the access prototype mapping is queried to retrieve an access control list to be applied to the new data object. Via this scheme, the penalty of parsing and internalizing the policy descriptions is only paid once, at application startup time. After startup, the internalized policy descriptors are used to automatically generate the access control lists for resources and attributes created by the application. The “generation” of access control lists is essentially just a table lookup in the prototypes mapping, and is thus quite inexpensive.

Since static roles are only manifested as access control lists at runtime, determination of access rights is trivial. There is no need to search role membership lists to determine access; instead, the system merely retrieves the access right associated with a given user when that user attempts to access an object. Determination of access at runtime is as simple as a dictionary lookup to retrieve the right from the access control list, and then an arithmetic comparison to determine whether the right will be granted.

Note that the owner of a resource always has full rights to it. Updates and reads of resources by their owners is the common case and bypasses the access control mechanisms. Since ownership is authenticated this poses no security risk.

Dynamic Role Implementation

Whereas static roles are implemented by associating a list of users with a set of access control rights, dynamic roles are implemented by associating a predicate function with a set of access control rights. When a client starts, it generates a “map” of access rights for certain resources and attributes that are represented by the policies it loads. In the case of dynamic roles, each resource and attribute keeps a list of access rights and the predicate functions which map onto those rights.

When a request for access is made, Intermezzo first evaluates the access control list for the object to see if the requested right is explicitly granted to the user making the access. If it is not, then the system scans the list of access rights associated with predicate functions. For any access rights that provide the requested access, the system evaluates the associated predicate functions until either a predicate returns true or no predicates remain that might be able to grant the requested access.

In essence, this scheme means that the system first applies the static roles to see if access may be granted. Only if they fail does the system attempt to apply dynamic roles. Further, only the predicate functions that have the potential to grant the access are evaluated, and they are only evaluated until the access is granted. This is consistent with the “liberal” application of the access control system throughout Intermezzo: if *any* policy would grant access to a particular object, then access will be granted.

Predicate functions are evaluated inside the server and execute with the permissions of the user requesting access. Evaluation occurs in the server because the alternative, execution within the client, would be a security risk: rogue clients could claim to have executed the predicate and return a successful condition to the server. Predicate execution occurs with the user’s permissions to ensure that a predicate function cannot be used as a “Trojan horse” to gain access to resource data that would otherwise not be accessible to a given user.

Note that predicates may execute arbitrary code. As a result of this property, predicate functions can (and indeed often are) chained together: the evaluation of a predicate function causes an access to some other data item which may be protected by another predicate. The system will evaluate these chained predicates and return the result to the top-level function.

Below is an example of a predicate function expressed in the form of extended Python used by Intermezzo. This predicate defines a role for the “demo day” scenario discussed at the beginning of this chapter. There are a number of possible ways to define a “demo day” predicate; this one considers the following characteristics salient:

- The day must be demo day.
- Only extend access to others when the owner (“me”) is in the lab.
- The user must be running the demo application (“Montage”).
- Access is only extended to those who are themselves in the lab.

```
def predicate(subject me context):
    if date.today() == “August 28”:
        if me.Location == “GVU Lab”:
            if me.Activities.member(“Montage”):
                if subject.Location == “GVU Lab”:
                    return TRUE

    return FALSE
```

The predicate consists entirely of a set of conditions to determine (1) the date, (2) the location of the resource owner, (3) the current activities of the owner, and (4) the location of the requesting user.

While some effort is made to speed dynamic predicate evaluation (by evaluating static roles first, and by shortcut evaluation, for example), there are other optimizations that are possible but are not implemented currently. First, predicates can be cached on the server; clients would then only transmit an identifying token for the predicate, rather than the predicate itself.

Second (and more difficult to implement), it may be possible to pre-process the predicates to determine which objects are examined by it. A predicate result can only vary depending if the “world state” represented by its objects has changed.

Policies and Session Management

The coordination features supported by Intermezzo—awareness, session management, and policy—all interact with and sustain one another. Information about users and their environments is used by the session management system to enable light-weight collaboration, and by the policy system to enable dynamic assignment of users to roles. The policy system also interacts directly with session management, by providing users with fine-grained controls over session access.

The policy system used by Intermezzo can restrict access of users to a collaborative session by restricting the information used by the session management facilities to do its job. Policies for session management are useful in a number of situations:

- User A may only wish to participate in collaboration with User B.
- User A may only wish to be informed of artifact sharing with members of his or her immediate workgroup.
- User A may only wish to collaborate with others at certain times of the day, or when some external condition in the environment is satisfied (the potential collaborator has information regarding the latest stock quotes, for example).

It should be clear from our previous discussion of policies for restricting access to user data that session management controls simply represent a specific domain of restricted data. As discussed in Chapter IV in the section “Policy Controls” on page 77, the basic awareness information model is extended by the session management service with a number of attributes designed to support collaboration. These attributes are actively used by the runtime environment and by clients to perform both implicit and explicit session management.

By vesting a certain power in these attributes (the power to conduct session management services), we can gain control over session management by restricting access to them. In the case of implicit session management, access to the `Implicit SM` attribute can be placed under policy control. This measure allows users to determine who, and when, access to session management will be granted.

In the case of explicit session management, access is regulated by restricting the `Name` attribute of `Session` resources. Without knowing the name of a session, joiner-based explicit session management is impossible. For initiator-based models, attributes used for notification on `Subject` resources may be restricted to make session invitation impossible.

Like any other data attribute, the attributes used for session management can be given dynamic access. Thus, session management access can be based on a restricted set of users, some characteristics of those users, or characteristics of the environment.

Common Policies

While the policy system used by Intermezzo is flexible and powerful enough to capture a range of complex, “real world” situations, it is also useful for describing a number of common general policies. This section describes two general policies which seem to be not only highly useful, but also “in demand” by users.

Anonymity

In some collaborative situations it may be useful to prevent broad access to a set of information about users. For example, in brainstorming sessions such as those provided by group decision support systems (GDSS), research has shown that often better results are achieved if the participants do not know who is submitting ideas [78][105]. By keeping participants from being able to identify one another, social barriers to contributing to the session are reduced.

This sort of information restriction is called *anonymity*. There is no information associated with users that could be used to identify them. Further, it is impossible (or at least unlikely) to track user data over a long period of time in an attempt to glean enough information to identify a user.

Intermezzo makes the construction of policies that correspond to anonymity possible. The precise definition of such policies will vary from site to site because of the need to restrict different pieces of information based on common user behaviors and tasks. For example, at a site where users are typically mobile, it may be desirable to allow access to information about location ("I know someone's in the coffee room, but I don't know who it is.") At a site where users typically sit at their desks all day, information about location is in practice equivalent to direct information about user identity.

Below is one (perhaps overly simple) definition of a policy to support anonymity.

```
policy Anonymity {
   resource Subject = {
      attr * = ANONE
   } = REXIST
   resource Verb = {
      attr * = ANONE
   } = REXIST
```

This policy does not allow any user to read or even determine the existence of any attributes on the `Subject` or `Verb` resources. It does allow the determination of the existence of these resources as a whole, however, and it does allow access to `Object` and other resources.

The anonymity policy as defined above is not the same as "invisibility:" invisibility would remove the `EXIST` rights from resources which would prevent even the detection of the presence of users. Anonymity is weaker in that it permits questions like, "How many users are there?" and, "What applications are being run?" while still removing identifying characteristics from that information. Still, however, anonymity is so restrictive that it prevents many of the activity-based interaction features of Intermezzo from operating: since `Subject` and `Verb` slots cannot be determined, it is impossible for a session management system to rendezvous with a user running with anonymity enabled.

Pseudonymity

Often, anonymity is too restrictive in collaborative situations. While it is useful in certain constrained environments (such as the brainstorming example), it severely limits the flow of information that may be useful to coordination. *Pseudonymity* is a policy that protects privacy but still allows access to information that can support coordination between groups of users.

Pseudonymity is similar to anonymity in that it does not allow information published in the shared information space to be associated with an actual, human user of the system. It *does* however allow the use of identifying “handles” which can be used to track users in the abstract. Pseudonymity allows questions like “What is user X’s typical work flow?” and, “Which users that run Framemaker also run Photoshop?” Pseudonymity supports the collection of user statistics and allows the system to track the paths of individual activities, while preventing those activities from being associated with an actual person. A common analog in the social sciences is the use of code numbers or names for experimental subjects. These codes are typically used over the course of a long-lived experiment (sometimes over a period of years), but still provide no information about the true name of “User X.”

Again, like the implementation of anonymity in Intermezzo, a policy for pseudonymity may vary from site to site. At some sites it may be possible to release certain pieces of information while preserving pseudonymity; at others, more information may be restricted.

Below is an example of a policy for pseudonymity:

```
policy Pseudonymity {
    resource Subject
        attr Location = AREAD
        #...other attributes we may wish to allow access to...
        attr * = ANONE
    } = REXIST
    resource Activity {
        attr * = AREAD
    } = REXIST
}
```

This policy varies from anonymity in a number of respects. First, it selectively allows access to certain attributes of the `Subject` resource (`Location` in this example). Next, it allows read access to the attributes of `Activity` resources. Allowing reads of `Activities` allows applications to *read the links* (meaning, the actual hashed resource identifier) in the `Activity` slots for `Subject` and so on, yet prevents the reading of the actual resources pointed to by these links (since `Subject` only supports limited read access).

By allowing applications to read the links, but not the data, in an `Activity` resource, applications can detect when several `Activities` “point” to the same user, while restricting the reading of potentially identifying information about that user.

Pseudonymity is useful for the (perhaps very common) case of access control where a user may say, “I don’t mind people knowing what I’m doing, as long as they don’t know it’s *me* doing it.” Note that users with pseudonymity enabled are able to interoperate with the Intermezzo implicit session management services since the policy does allow access to activity and some user information.

An Aside: Controlling Access to Personal Space

With the appropriate application conventions, a policy system based on access control can regulate access to personal virtual space, in addition to personal data. While my research has not explored this ability fully, any operating system with the concept of distinguishable users and access control can be used to prohibit unauthorized users from gaining access to machine-based resources: workstation displays, audio input and output, and other devices that may be under computer control. This is again much like the use of “special” data to represent artifacts and capabilities, as in the session management policy control system.

The procedure to regulate such access is simple: a server process is executed that opens the resource in question. Access to this server is restricted to client applications who hold a valid digital key. The keys can then be distributed securely via the Intermezzo sharing model as attributes on Subject resources: Subjects maintain all relevant information about a given user, including the information necessary to open windows on their displays, generate audio notifications, and so forth.

Once the keys are present as Subject attributes, their distribution falls under the control of the policy systems described earlier. Access to keys (and hence to workstation resources) can be selectively provided to individual users. Further, the user-based authentication mechanisms used by Intermezzo can be used to strengthen and refine security beyond what may be present in certain host-based access control systems.

While this solution again requires application cooperation, it holds the potential to provide fine-grained and highly-dynamic access control over the machine-controlled resources on users' desktops.

Chapter Summary

The ability to describe to a system how it should behave in the face of a potentially chaotic environment is important for collaborative work. In this chapter, I have presented a system for describing policies in terms of access control rights on data objects. With the proper conventions governing the types of data present in the system, an access-based model of policy is sufficient for describing and capturing many of the workaday situations in which policy is needed. We have examined a number of scenarios that the access-based privacy model can represent.

We have also seen that policy systems, at least in the forms typically found in current collaborative systems, may be insufficient for coordination needs: they often lack the flexibility, responsiveness, and ease-of-use necessary to make them applicable in the many "ephemeral" access control situations that arise during collaboration.

To address this issue, we have presented a formal model for representing policies, and how those policies are mapped to users, that can make access control more able to respond to the fluid needs of interpersonal collaboration. In the Intermezzo model, roles representing users are mapped onto policies representing a set of access control rights. The definition of roles (that is, the set of users constituting a role) can be performed either statically (as in traditional systems), or dynamically (at runtime).

The dynamic mapping of users to roles accomplishes a number of goals. It avoids the problems inherent in the premature assignment of users to roles, and discussed in the literature; role membership can vary during the lifetime of a collaboration; and finally, membership can be based on arbitrary attributes of the users or the environment.

Dynamic roles expand on one of the central themes in this work: by bringing information about users and their environments into the system, we can make computer-augmented collaboration more responsive, and we can free the users of many of the burdens implicit in working with today's collaborative systems.

Finally, this chapter has presented an implementation of the policy system as based on the Intermezzo foundation layer. This implementation provides a language for specifying roles and policies, and an efficient runtime infrastructure for evaluation of policies.

Chapter VI

Developer Perspectives

This chapter provides an overview of the programming models, programming interfaces, and developer support implemented by Intermezzo. The programming models used by the system support the easy construction of collaborative applications that use the coordination features implemented in the Intermezzo runtime environment. This chapter discusses notification services used to communicate changes in world state to applications. Intermezzo provides an array of notification mechanisms to developers. Next, the notion of embedded computation is introduced. Embedded computation provides a way for clients to download code into the runtime environment or into other clients for remote execution. Finally, the actual programming interfaces available to developers are explored.

Introduction

A runtime infrastructure for supporting collaboration may never be used if there is not strong development-time support for constructing applications that use the runtime environment. So far this dissertation has provided justification for runtime features, and investigated a framework for supporting the coordination needs of collaborative applications at runtime. Developmental support has only been mentioned in passing however.

One of the aims of this work is to provide not only runtime infrastructure, but a developmental interface to that runtime infrastructure that satisfies the needs and expectations of application developers. Builders of collaborative systems need toolkits that can allow them to access the runtime facilities in the environment easily, efficiently, and with a generality that makes the runtime facilities applicable to a range of applications.

This chapter investigates the developmental facilities (that is, the toolkit component) provided by Intermezzo. We have seen that the runtime environment for coordination provides a very dynamic and rich set of functionality for supporting collaborative activities. Thus it is essential that the constructs available to developers be able to capture this richness; an overly restrictive or constraining programmatic interface would subvert the goals of the project since the dynamism of the runtime environment would be inaccessible.

This chapter focuses on several aspects of the programmatic interface in particular:

- Notification Services.

  Notification is the process of informing clients of changes in either their state or the state of the "world." Intermezzo provides an array of notification services that span a number of dimensions, including ease of use and generality.
- Interfaces to Shared Object Services.

  For maximal expressive ability, the developmental system provides direct access to the shared object services implemented in Intermezzo. Essentially this is an application programmer interface (API) to the foundation layer services and is intended to provide generality and power at the expense of ease-of-use.
- Interfaces to Coordination.

  Intermezzo provides a number of higher-level, "coordination-specific" APIs that are an attempt to address common-case application needs for collaboration. These interfaces are used to interact with specific features of the coordination system (session management, for example). They are easy to use but somewhat restrictive since they do not provide full access to the foundation object model.
- Scripting.

  The runtime environment is fully scriptable using a high-level interpreted language. Scripts can execute either within client or server address spaces. The scripting ability is useful for application building (applications can either be written entirely in the scripted language, or they can be hybrid systems written in both the scripting language and a compiled high-level language). Scripting is also useful for performance in a number of situations, and can also serve as an extremely general mechanism for notification.

These features are addressed in depth in the following sections.

Notification

Application programs often have a need to be asynchronously informed of changes in the state of their environments. Applications typically request such asynchronous updates by telling the environment the types of occurrences the application is "interested" in. The environment then informs the application when the desired condition is fulfilled.

This process is called *notification.* Intermezzo provides a range of notification services to satisfy application needs. These services range from mechanisms with extreme generality and corresponding difficulty of use, to still powerful, but somewhat less general systems which can be used easily.

Notification is an important issue to address in any collaborative toolkit because the conditions which applications may deem interesting are especially broad-ranging. A common thread throughout this research is that collaborative infrastructures must be non-constrained to the point that they can deal with the fluidity and dynamism of human interaction. This motif has manifested itself in implications for awareness services, session management, and policy. Likewise, the notification services used by applications must also be able to capture dynamic interactions and not inhibit the application's ability to react to such conditions.

Furthermore, the notification systems in use must be able to capture the free-form nature of the information which is present in coordination situations.

The notification mechanisms provided by Intermezzo make up a spectrum of generality. From the least general to most general, the notification mechanisms include the following:

- Events.
- Triggers.
- Embedded Computation.

Events and triggers will be addressed fully in this section; embedded computation will be addressed later in the chapter, since this particular mechanism has uses beyond notification.

Events

Events are a common concept in toolkits for a number of application domains. Typically, an application writer makes some function call to *solicit* interest in a range of occurrences which he or she deems interesting. The "environment" (which, depending on the implementation of the development infrastructure, may include toolkit code or some other runtime feature such as a server) responds by generating messages called *events* to the application whenever the specified occurrence takes place.

The application can, depending on its structure and flow of control, respond to the messages in one of two ways. First, the application can poll or sample the stream of incoming events at regular intervals, and take appropriate action when a particular event arrives. Second, the application can install functions (sometimes called callbacks) which will be triggered automatically by the toolkit whenever the event arrives. Polling places the control and responsibility for event dispatch in the application code; callbacks place the dispatch mechanisms in the toolkit.

From the perspective of coordination, one of the problems with most event systems lies with the mechanisms for event solicitation. Most toolkits provide only a limited range of event types which may be specified. As an example, in the X Window System [94], event types which may be solicited include key presses, button presses, and window maps (which indicate that a window has appeared on the screen). It is impossible to specify constructs like, "Only notify me if the 'B' key is pressed," or, "Only notify me if a Framemaker window is mapped."

To overcome this limitation, Intermezzo uses a pattern matching scheme for event solicitation. Clients *describe* occurrences that are of interest to them. The toolkit provides mechanisms that allow developers to construct fine-grained descriptions of occurrences based on the state of the global database. The scheme is powerful enough to allow constructs like, "Notify me if Ian runs the Videometer application in the lab this afternoon," and, "Notify me if anyone in my project checks in code." The first is an example of a highly-specific specification involving a particular user, a particular application, and a well-specified point in space and time. The second is more general; it describes an occurrence involving any of a number of users and a variety of circumstances. The use of pattern matching to describe, rather than completely specify, events of interest is similar to message patterns in the Field system [81][82].

Implementation

A potential weakness of pattern matching event schemes is that performance may suffer if the implementation of the system is not chosen wisely. It is necessary to determine the "granularity" of the components of the global state that may be matched against to prevent potentially unbounded execution times.

Intermezzo uses a “key point” scheme to evaluate solicitation patterns. When a client initially subscribes to a class of events, it provides the runtime system with a pattern to match against and also specifies a *key point* that indicates the circumstances under which the pattern matcher will be run. At certain predefined points when the server changes state, all of the pattern specifiers associated with that key point will be evaluated. All patterns which evaluate to `TRUE` cause the generation of an event back to the client which registered the pattern.

There are three key points used by the server, corresponding to object creation, object deletion, and object modification. Changes in server state cause the evaluation of the pattern lists associated with one of these key points. This selection of three key points is useful because it corresponds to semantically meaningful constructs (the start of a new activity corresponds to object creation, for example), and it is also efficient since it maps directly onto low-level database functions.

Patterns are evaluated against the characteristics of the particular resource being created, modified, or deleted. Patterns are specified via as a list of tuples, called *specifiers*:

$$spec = \{key, operator, value\} \qquad \text{(EQ 6-1)}$$

Each specifier consists of a *key*, an *operator*, and a *value*. The key denotes an attribute of the resource to be examined. The operand is a unary or binary comparison function which evaluated to either `TRUE` or `FALSE`. The value is an arbitrary second operand used by the operator function.

The server performs short-circuit evaluation at its key points. The instant a specification operator returns `FALSE`, the evaluation of the specification is halted. Pattern matches are still potentially expensive however.

The following set of operator predicates are provided:

TABLE 6-1 Event Specifier Operations

Operator	**Description**
Noop	Perform no action (always returns TRUE).
Equality	Test for “equality” (as defined by the types of the operands).
Type Equivalence	Test that the operands are of the same type.
Subset	Test that the provided value is contained by the matched attribute.
Less Than (Before)	Test that the provided value is less than the matched attribute (for some types of attributes such as times, this is interpreted to mean “before”).
Greater Than (After)	Test that the provided value is greater than the matched attribute (for some types of attributes such as times, this is interpreted to mean “after”).
Less Than/Equal To	Test that the provided value is less than or equal to the matched attribute.
Greater Than/Equal To	Test that the provided value is greater than or equal to the matched attribute.
Type Name Test	Test that the matched attribute is of a type specified by the provided value (which is interpreted as a string).
Existence Test	Test that the specified attribute exists. The value operand is ignored.

TABLE 6-1 Event Specifier Operations

Operator	Description
Nonexistence Test	Test that the specified attribute does not exist. The value operand is ignored.

This set of operators allows fairly complex constructs to be created and pattern matched against the data in a particular resource being created, modified, or deleted.

Example

Below is an example event specification which describes one of the conditions mentioned earlier: “Notify me if Ian runs the Videometer application in the lab this afternoon.”

```
RESOURCE_CREATE

Type         EQ        “Verb”
Name         EQ        “Videometer”
Time         AFTER     12:00
Host         SUBSET    {warhol, dali, picasso, chagall, magritte, haring}
*User.Name   EQ        “Ian Smith”
```

The specification is shown in a stylized notation; developers interact with event specifications as data structures in either Python or C++. There are several possible ways to implement this “Videometer” condition; the one shown uses the `RESOURCE_CREATE` key point: whenever any resource is created, the specification will be evaluated for a possible match. Other implementations may be built by watching for changes in the set of `Activities` slot of Ian’s `Subject` resource. The specification shown here checks for creation of resources of type `Verb` with the application name attribute of “Videometer.” The application start time must be noon or later, and the host must be one of the systems in the multimedia lab. Finally, the name of the user must “Ian Smith” (the “*” notation indicates an indirection: the traversal of a link).

Client-side Event Processing

The client-side toolkit library provides mechanisms to facilitate event processing. When a client starts, the library spawns a new thread dedicated to event processing. The threaded processing of events obviates the need for “main loop” style processing in which the application writer must return to toolkit code periodically for event service. Essentially the client event processing happens “in the background” and the main client thread can proceed without interruption from the event processing mechanisms.

Clients register “Callback” objects with the toolkit event processing system. Callbacks are objects that provide a call interface for invocation when an event arrives; clients subclass the Callback object to provide any specific data that they may require to perform their event processing.

The Callback base class is also subclassed to provide multiple language support. For example, a PythonCallbck subclass provides facilities for execution of Python scripts in response to events. Callback subclasses allow application writers to specify their callbacks in whatever language is convenient.

Searching

The event specification mechanisms used by Intermezzo provide a generalized method for describing world states that may occur in the future. But the pattern matching system also can be applied to searches of the existing data space. Resource specifications can either be applied in a *time-forward* match (for describing future matches that will cause events to be fired), or in a *time-backward* match (for describing existing resources that will be matched and returned to the client).

The format of the specifications used for both searching and event solicitation are the same. Using pattern matching for searches provides a general mechanism for scanning for desired resources in the shared data space.

The client-side APIs provide two distinct interfaces for searching and event solicitation. The event solicitation API passes a specification to the server and returns no immediate value to the client. Instead, a message will be generated whenever the pattern is matched in the future. The search API passes a specification to the server and immediately returns a list of the resources matching the specification.

Triggers

Another notification mechanism provided by Intermezzo is *triggers*. Triggers are a notion from access-oriented programming. A trigger is a function that is executed whenever a certain data object is accessed. Triggers provide general notification mechanism because they allow arbitrary client-provided code to be associated with a data object. There are a number of important differences between triggers and events however:

- Triggers may be installed at the same resource key points as events; further, they can be associated with *specific* attributes on *specific* resources. Thus, triggers are installed on particular existing objects of interest, and can thus be more efficient than events (since event specifications are evaluated each time the data store is modified).
- Triggers can be processed in either the client or the server. Events can only be processed in the client. The server-side event processing available to triggers may be more efficient in some cases, depending on the location of other data required by the code.
- Triggers cause the immediate execution of code, rather than execution in response to a message transmittal. This is another reason that triggers may be more efficient in some circumstances than events.
- Triggers can be tied to either resources or to specific attributes. This allows finer granularity than events, which can only be tied to key points associated with resource changes.

Traditional access-oriented systems often use triggers to maintain global data consistency. Intermezzo supports the execution of arbitrary, client-supplied code via triggers. Once a trigger function is associated with a data object, the call to that trigger is automatic whenever the data object is updated. Triggers may either be associated with a resource (as specified by its type or its ID), or with an attribute (as specified by the ID of the resource containing the attribute, and then the name of the attribute itself). Resource type triggers are functions that are evaluated whenever a resource of the particular type is created. This is similar in principle to the event key point scheme.

Typically a developer will install a resource creation trigger, which then installs more efficient per-attribute triggers where needed. The creation trigger is only used to instrument “interesting” data objects with specific trigger functions. For example, a creation trigger may be installed for `Subject` resources. This trigger may install a per-attribute trigger on the `Location` attribute of that resource, which can then execute some arbitrary code when the location of a user changes.

Triggers are installed on replica objects; recall that replicas can exist in either the client or server. Thus it is possible to place triggers “near” any other data they may require access to for performance reasons. Typically client-side triggers are useful only for tracking changes to objects in the local data store; they are primarily used as a code structuring device to keep code dependent on data updates associated with data objects. Server-side triggers are useful for notification of “global” changes to the resource database.

The combination of client- and server-side trigger processing, as well as the variety of methods for binding triggers to data objects, makes triggers exceptionally powerful. While the pattern matching scheme used by events makes them broadly applicable to a number of common application programming situations, triggers “fill in the gaps” left by events; in particular, triggers are useful in any situation where the circumstances of interest are not easily expressible via specification tuples. This is why tuples are not triggered via pattern matching: such a system would merely duplicate the strengths and weaknesses of the existing event system.

Instead, triggers provide a flexible means for allowing arbitrary code to be executed in response to changes in the data space of the system. This code may function as an application callback, or it may generate an event back to the client for further processing by the application (in effect, functioning much like an event dispatch system).

Embedded Computation

One potential problem with highly-networked and distributed systems is the performance penalty that is inherent in the fact that computation is often separated from data. Typically, most application-specific computation occurs in the client address space. The client exchanges messages to and from a server, which can perform some specific and constrained (and typically limited) set of computation on the client’s behalf. Some performance problems can be overcome by using locally-cached data (and in fact Intermezzo uses this approach) but there is still a significant overhead required to maintain global data consistency of the replicated data objects.

A possible solution to some performance problems inherent in distribution is to move the computation closer to the data it is operating on. Since moving (replicating) data still involves the overhead of locking and consistency checking, code distribution may be a better alternative than data distribution in some circumstances.

Intermezzo supports the notion of *embedded computation*. Embedded computation means that segments of code may be downloaded, or embedded, in the server or even other clients for execution. Moving code close to the data it manipulates has a number of performance advantages in Intermezzo. First, if an application needs to access large amounts of data, server embedding obviates the needs for large numbers of round-trip requests between the client and the server. Second, read/write locking can be done internally to the server for heightened performance.

Originally, embedding was added to the Intermezzo programming model strictly for performance reasons. Embedding has proven to have many more uses however. The most important of these is that embedding supports the construction of “hybrid” applications constructed both from compiled code and from the intepreted scripts meant for embedding. In essence, hybrid clients embed an executable script within themselves to augment their functionality.

This approach allows developers to use different tools for different jobs. Performance-critical portions of applications can be written in a compiled language. Code that is not performance-critical, that is experimental, or that requires extreme flexibility, can be expressed via the scripting language and embedded.

Embedding also serves as an extremely general notification mechanism for clients. If the features provided by events and triggers are not sufficient, clients can download scripts into the runtime environment that will act as “agents” on their behalf. These scripts can continually monitor for interesting changes in the global data state and take appropriate action as necessary (typically by notifying the client).

Another use of embedding is for server monitoring. When the runtime server changes state, it will update a number of variables that are accessible to embedded code scripts. These scripts can serve as administration and monitoring tools and are in effect a means of extending the capabilities of the server at runtime.

A final important use of embedded computation is the support of user proxies. There may be many circumstances when users may wish to be notified of changes in the coordination space that occurred while they were away. For example, a user may wish to be made aware of certain changes that took place when he or she was on vacation or otherwise out of the office. By downloading scripts that act as proxies for the users, it is possible to collect data directly from the server, even when no applications are being run by the user. Disconnected or rarely-connected users can download code to log relevant occurrences until the next reconnect. This form of scripting seems to have particularly important uses in asynchronous and autonomous collaboration.

To summarize, embedded computation supports a number of facets of the runtime environment:

- Performance in a distributed setting.
- Notification.
- Hybrid clients.
- Efficient monitoring of server state.
- Proxies for disconnected or autonomous users.

Implementation

Embedded code is handled in essentially the same way, whether it executes in the server address space, or the address space of one of the clients. All embedded code is executed in a thread that runs with the Intermezzo permissions of the user which downloaded the code. This threaded approach has a number of benefits: first, it provides a secure environment in which users can be assured that their privacy will not be compromised by the scripting facilities. Second, since each script runs in its own thread, the performance impact on other clients’ scripts is minimal. Although the Intermezzo thread scheduler does not do this, it is possible to implement the system so that scripts have a limited amount of CPU resources available to them. This feature would make denial of service attacks more difficult.

The scripting language used by the system is based on an extended version of the Python language [112]. Python is an interpreted object-oriented language that supports threads and exceptions, is easily extensible, and supports dynamic typing. I chose Python primarily based on its ease of integration with Intermezzo, and its strengths as an object-oriented language. The core Python facilities have been extended with a new collection of objects that represent the constructs available in Intermezzo: resources, threads, and so on. Scripts written in this language can manipulate the global object space just like any other application (see the next section for more detail on the specifics of the Python extensions).

To summarize, the use of embedded computation has many uses in coordination and collaboration settings, but there is still much work to be done. In Intermezzo, embedded computation is primarily used only to gain performance and support the construction of hybrid applications for rapid development. The other uses of embedding, particularly for proxies to support asynchronous and autonomous collaboration, seem to be fruitful areas for future research.

Programming Interfaces

The actual programming interfaces used by developers are available in two languages: C++ and Python. Further, there are several “layers” of APIs available in each language. This section presents an overview of the programming model supported by Intermezzo. A more full presentation of the programming interfaces can be found in Appendix A and Appendix B later in this work.

C++ Programming Interfaces

The C++ API to Intermezzo provides three separate layers at which developers can work. The first layer is an extremely low-level API that supports concepts like replicas and server protocols directly. The middle layer is a “wrapper” around the low-level API that deals in terms of resources with automatic consistency control, event specifications, and most of the constructs described in the foundation layer of the system. Finally, a “feature-specific” API provides a high-level interface directly to certain collaborative features, such as awareness and session management.

The lower-level APIs provide more performance, and greater control over the interaction with the runtime system. The higher-level APIs are much easier to use, however, and represent the common coordination situations encountered by applications.

Low-Level Interfaces

The low-level programming interface in C++ provide direct access to replicas of objects and the protocols used to exchange data with the runtime server. This API is almost never used by clients; instead it is primarily used to construct the higher level APIs and the runtime environment itself.

When using the low-level API, replicas are manipulated directly. Thus there is no inherent data consistency provided automatically; if clients desire consistency they must implement it themselves using the proper server protocols.

The most important construct in the low-level API is the `IMClient` class. `IMClient` represents a connection to an Intermezzo runtime server, and encapsulates the data required by the client, as well as the functionality needed to communicate with the server. The `IMClient` class spawns threads as needed to deal with incoming messages from the server.

Resource replicas may be created directly, They are “published” to the server by calling the `Insert` method on an `IMClient` instance. `Insert` builds a message for transmittal to the server and instructs the server to construct a replicas of the resource. All of the protocol messages available to the server are encapsulated by methods on the `IMClient` class.

Here is a short program which constructs a new resource and transmits it to the server using the low-level API:

```
IMClient client(“localhost”, 1966);     // Provide host and port of server
Resource r = new Resource(“Subject”);   // Construct a resource
r.SetAttribute(“Foo”, Int(72));         // Create a new attribute on it.
Status s = client.Insert(r.GetID(), r); // Publish it
```

Mid-Level Interfaces

The mid-level interfaces are considered the “usual” APIs for clients that require more direct access to the shared data space than is available through the feature-specific APIs. In the mid-level programming interface, developers manipulate collections of replicas called *contexts*. Each context represents a namespace of objects available in certain address spaces. Clients can create their own contexts (called a *local context*) as a private “work space” for operating on objects. Local contexts also serve as caches for replicas in use from the global data store. Clients also typically create one *shared context* that represents the global data space they are sharing with other clients on the network.

Clients obtain access to data by calling methods on the contexts they initialize. All access to data is through a class called a *handle* which encapsulates the consistency checking and access control needed by most applications. Handles essentially act as “safe” versions of the resource objects in the low-level API.

Below is an example of a code fragment which creates and updates a globally-available resource.

```
LocalContext local;
SharedContext shared;
Handle h1 = local.Create(“Subject”, P_TRANSIENT); // Create a new resource
Handle h2 = shared.Clone(h1);                     // Publish it.
h2[“Location”] = “College of Computing”;          // Set some values.
```

Each type of context, whether local or shared, supports a common API. This API allows the creation, destruction, and cloning (publishing) of resource data throughout the global data space.

Feature-Specific Interfaces

The feature-specific programming interfaces provide an easy-to-use API for applications for which the common functionality provided by Intermezzo is sufficient. The feature API provides a set of routines for session management, policy, and awareness.

This API maps directly onto the coordination features described in this dissertation. The major goal of this API was ease of use, rather than generality. Thus, some applications may need to “drop down” to the mid-level API if their needs differ sufficiently from the facilities provided by the feature-specific API.

Below is an example of programming using the feature-specific API.

```
// Create and publish an activity all at once.
Handle myActivity = PublishActivity(local, shared, argc, argv);
// Ask the server for session management events involving this activity.
Status status = SolicitSMEvent(shared, activity, callback);
```

Note that many of the constructs of the mid-level API (handles, contexts) are also present in the feature API.

Python Programming Interfaces

The Python programming interface provided by Intermezzo supports a subset of the features present in the C++ API. The Python bindings are used not only for application development, but also for embedded scripts and trigger functions. Python provides an extremely easy-to-use object model, threads, and exceptions which makes it well-suited for an interpreted API to Intermezzo.

Like the C++ interfaces, the Python API presents a mid-level and a feature-specific programming model to clients.

Mid-Level Interfaces

The mid-level programming model is essentially a direct translation of the C++ mid-level API. Constructs such as handles and contexts are implemented using their C++ counterparts and work the same as they do in C++. C++ exceptions are “translated” into Python exceptions when using the Python bindings. The Intermezzo runtime support system also provides automatic conversion between Intermezzo-native types and Python types, so that portions of applications can be arbitrarily written in either Python or C++ with free data interchange.

Below is a sample Python program which uses the mid-level APIs.

```
# Bring in the Intermezzo language extensions, and create some contexts.
import intermezzo
l = intermezzo.localContext()
s = intermezzo.sharedContext(l, ‘localhost’, 1966)
# Create a resource, publish it, and set an attribute on it.
h1 = l.create(“Subject”, intermezzo.TRANSIENT)
h2 = s.clone(h1)
h2.Location = “College of Computing”
```

This example creates a local and a shared context, creates a resource in the local context, and then publishes and updates it.

Feature-Specific Interfaces

Like the mid-level interfaces, the feature-specific programming model is a direct translation of the C++ feature-specific APIs. The Python feature-specific bindings provide support for session management, policy, and awareness.

Below is an example.

```
activity = intermezzo.publishActivity(local, shared, argc, argv)
intermezzo.interpret(shared, “print ‘I’m downloaded code!’”)’
```

This example publishes an activity record in the shared context, and then downloads a code snippet that does a simple output).

Sample Applications

A number of test applications have been built, using both the C++ and Python programming interfaces.

Test App

The `testapp` application exercises a number of the low-level features in the Intermezzo runtime environment. This application serves as a test of the server connection procedures (authentication, authorization, and name establishment). It creates local and shared contexts and creates, copies, updates, and deletes a number of resources between these contexts. It also demonstrates the pattern matching event solicitation mechanism used by Intermezzo. It is a text-based application.

Implementation

```
int
CallbackProc(Functor *f, void *args)
{
    Callback *cb = (Callback *) f;
    Message *m = cb->GetMessage();
    cerr << “Callback received message from server: “ << *m << endl;
    return 0;
}

int
main(int argc, char *argv[], char *envp[])
{
    LocalContext localContext;
    SharedContext sharedContext(localContext, argv[1], atoi(argv[2]),
                                    argv[3]);
    Handle local1, local2, localClone, shared1, shared2, newClone;

    // At this point we’re attached to the shared context. Create some
    // resources in the local context...
    local1 = localContext.Create(“Local Resource1”);
    local2 = localContext.Create(“Local Resource2”);

    // Add some attributes to them...
    local1[“Name”] = “Fred”;
    local1[“Location”] = “CoC 260”;

    // Test local cloning of resources...
    localClone = localContext.Clone(local1->GetID());

    // Create some resources in the shared context.
    shared1 = sharedContext.Create(“Shared Resource1”);
    shared2 = sharedContext.Create(“Shared Resource2”);
```

```
        // Clone between the two contexts...
        newClone = localContext.Clone(shared1->GetID());

        // Test searching
        Handle test1 = sharedContext.Find(local1->GetID());
        Handle test2 = sharedContext.Find(shared1->GetID());

        // Test event solicitation.
        ResourceSpecifier addSpec(Event::RES_MOD);
        AttrSpecifier attrSpec1(“SomeKey”, SpecOp::EQ, “foobar”);

        addSpec.AddSpecifier(attrSpec1);
        sharedContext.Solicit(addSpec, Callback(CBProc));

        Handle eventCauser = sharedContext.Create(“EventCauser”);
        eventCauser[“SomeKey”] = “foobar”;

        sleep(5);
    }
```

Results

This section shows the results of running the above program. First, we enter the password to the system. This password is used t to decrypt the RSA private key and introduce ourselves to the system.

```
% testapp chagall 1966
Enter password for Intermezzo: ********
```

Next, several resources are created in the local context, and updated.

```
[1] Inf: Resource created in local context:
    Type = “Local Resource1”
    ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2C
[1] Inf: Resource created in local context:
    Type = “Local Resource2”
    ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2D
[1] Inf: Resource updated:
    Name = “Fred”
[1] Inf: Resource updated:
    Location = “CoC 260”
```

After this, we test cloning a resource in the local context. This operation increments the reference count for the cloned resource, and returns a new handle to it.

```
[1] Inf: Local resource cloned in local context:
    ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2C
```

Next, we create several shared resources, and make a clone of it in the local context.

```
[1] Inf: Resource created in shared context:
      Type = “Shared Resource1”
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2E
   [1] Inf: Resource created in shared context:
      Type = “Shared Resource2”
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2F
   [1] Inf: Shared resource cloned in local context:
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2E
```

We now try to search for two resources in the shared context. For the first, we use the handle of a local-only resource, and so the search fails. For the second, we use the handle of a shared resource and the search succeeds.

```
   [1] Inf: Search for resource was UNSUCCESSFUL.
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2C
   [1] Inf: Search for resource was SUCCESSFUL.
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2E
```

Finally, we tell the server that we’re interested in resource modifications, where the “SomeKey” key acquires the value “foobar.” Here we construct an event specification for this occurrence. We then create a new shared resource and set the value of “SomeKey” to “foobar.”

```
   [1] Inf: Added event specification
      [ID = 34] RES_MOD: SomeKey EQ “foobar”
   [1] Inf: Resource created in shared context:
      Type = “EventCauser Resource2”
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2G
   [1] Inf: Resource updated:
      SomeKey = “foobar”
```

Here we see that the server generates an event to us, and evaluates our callback function.

```
   [1] Inf: Event received from server.
   Callback received message from server:
      RES_MOD Resource:
      ID = Keith Edwards <keith@cc.gatech.edu>-72304E35-3A86-2G
      SomeKey = “foobar”
```

Session Management App

The `smapp` client is used as a demonstration of implicit session management with Intermezzo. The goals with the implicit session management APIs are to facilitate the easy integration of light-weight rendezvous into application code. This application performs rendezvous based on file sharing. It takes the path to a file as an argument on a command line; when another application accesses the file, an event is generated to smapp informing it of the peer user.

This example shows the minimal amount of work that must be performed to (1) publish an activity record into the environment, and (2) respond (in a minimal way) to implicit session management events. A “real” application may notify the user of the confluence, or go into a special “sharing mode.”

Implementation

The implementation of smapp uses the C++ language bindings. It is a text-based application.

```
//
// This is our event callback. It simply prints out who else is
// using the file.
//
int
CallbackProc(Functor *f, void *args)
{
    Callback *cb = (Callback *) f;
    Message *m = cb->GetMessage();

    cerr << "\nThis file is also being accessed by " <<
                *((Owner *) (*m)[3]) << "\n" << flush;

    return 0;
}

int
main(int argc, char *argv[], char *envp[])
{
    LocalContext localContext;
    SharedContext sharedContext(localContext, argv[1], atoi(argv[2]));
    Callback callback = CallbackProc;

    //
    // Generate some objects for our activity record, and publish
    // them.
    //
    Handle subject = SM::GenerateSubject(localContext);
    Handle verb = SM::GenerateVerb(localContext, argc, argv, envp);
    Handle object = SM::GenerateFileObject(localContext, argv[3]);
    Handle activity = SM::GenerateActivity(localContext,
                    subject, verb, object);

    // Make sure our session management attributes are set!
    verb["Response"] = True;
    object["Implicit SM"] = True;

    sharedContext.Clone(subject.GetID());
    sharedContext.Clone(verb.GetID());
    sharedContext.Clone(object.GetID());
    sharedContext.Clone(activity.GetID());

    // When we get an SM event on this activity, file the callback!
    SM::SolicitSMEvent(sharedContext, activity, callback);
```

```
        //...wait for something to happen...
        sleep(30);
    }
```

Results

Here we run the session management test client, specifying a file of /etc/passwd. We are asked to enter a password to authenticate ourselves to the system.

```
% smapp chagall 1966 /etc/passwd
Enter password for Intermezzo: ********
```

Next we create subject, verb, object, and activity resources. The debugging output shows the attributes of the resources.

```
[1] Inf: New subject resource created:
    Type = Subject
    RealName = Keith Edwards
    Shell = /local/bin/tcsh
    EmailAddress = Keith.Edwards@cc.gatech.edu
    Hostname = chagall
    WorkPhone = (404) 894-6266
    LoginName = keith
    Organization = Georgia Tech Multimedia Computing Group
[1] Inf: New verb resource created:
    Type = Verb
    AppName = smapp
    Started = 14:44:31
    Response = TRUE
    ProcessHost = chagall
[1] Inf: New object resource created:
    Type = Object
    Name = /etc/passwd
    Implicit SM = TRUE
    SubtypeID = [chagall /dev/dsk/c0t3d0s0 3069]
    Subtype = FileSubtype
[1] Inf: New activity resource created
```

After the resources are created, we solicit session management events from the server.

```
[1] Inf: Session management event solicited, ID = 34
```

In another window, the user runs another Intermezzo aware application that accesses /etc/passwd (in this case I have run another instance of the session management test client). The server generates an event to the application, and the callback code registered by the application is run.

```
[1] Inf: Event received from server
This file is also being accessed by Keith Edwards <keith@cc.gatech.edu>
```

System Implementation

The Intermezzo collaboration support environment is implemented primarily in C++ and runs on Sun SPARCstations under Solaris 2.4 and 2.5. The system is approximately 49,000 lines of source code and is written in C, C++, and Python.

The foundation layer provides a robust object programming model, not specific to collaboration. The data model supports consistency, persistence, and replication of data objects. Further, any data object may be transmitted “over the wire” and reconstructed on the receiving end. Access to constructs in the foundation layer is fully authenticated and access controlled.

All programming interfaces are thread safe, and many are thread “hot” (meaning that they generate and use threads internally).

The port system provides a transport-independent API for communication programming. Subclasses of the `Port` class provide particular transports. Current subclasses include an in-memory port, a TCP/IP (transmission control protocol/internet protocol) port, an SVR4 memory queue port, and a transport-independent remote procedure call (TI-RPC) port.

The foundation API also provides general OS-level programming support implemented in C++. This support takes the form of a number of classes for synchronization (mutexes, condition variables, semaphores), data classes (containers, strings, and so forth), support for dynamic loading of code segments, standardized error and debugging output, and so forth.

The higher-level coordination features provide easy access to functionality via either C++ or Python. Further, the API supports “hybrid” applications written in both languages (for example, a program may be implemented primarily in C++ with small portions expressed in interpreted Python, or a program may be largely Python, with certain parts written in C++ for efficiency). Portions of application code can be executed in other address spaces for efficiency.

Chapter Summary

Abstract models of information sharing, and runtime support based on those models, are by themselves incomplete for the task of constructing robust, practical collaborative applications. To be usable, we must provide a means for developers to access these facilities easily. This chapter has presented developer perspectives on the concepts introduced by Intermezzo.

We have investigated a number of components of the developer support, or “toolkit,” in this research, including notification, programming interfaces for accessing shared data, interfaces for accessing collaboration-specific functionality, and the use of scripting through embedded computation.

Notification is one of the most important problems to be addressed in any developer support: how do applications (and, by extension, their users) become aware of changes in their environments? This problems is especially vexing in the case of coordination, where information that may be considered interesting is plentiful, change is rapid, and the notion of “interestingness” varies greatly from user-to-user, application-to-application, and moment-to-moment.

This research supports three main facilities for notification in the development infrastructure. The first of these is events. Intermezzo uses a familiar subscription model for events, except that interesting events are described via a specification, which is pattern-matched against the state of the global data store. Pattern-matching increases the utility and flexibility of events in collaborative situations.

The second notification feature is triggers. Triggers are functions that can be associated with particular data items, and are fired whenever those data items are accessed. Triggers fit nicely within the "everything is a resource" data-oriented model of Intermezzo.

Finally, embedded computation provides an extremely general mechanism for notification. Clients can download scripts into the runtime environment that perform arbitrary action, and report back whenever some interesting condition arises.

Embedded computation also has applications where performance is crucial, and for hybrid applications constructed in both compiled and interpreted languages.

Finally, this chapter has presented a set of programming interfaces atop which developers can construct applications. This research has resulted in a set of interfaces that callable from either C++ or Python. These interfaces provide access to all layers of the runtime infrastructure: "raw" data programming, shared data programming, and coordination-specific programming. These APIs are easy to use for developers, map well onto the object-oriented model of programming, and are easily integrated into existing applications.

Chapter VII

Conclusions and Caveats

Evaluation

Intermezzo is a study of how contextual information about users, their activities, and their surroundings can be brought into a computer-mediated environment with the goal of increasing coordination between users and applications. In this research I have followed a belief that by providing contextual information applications can be made more situationally-aware and responsive to users—applications can accommodate the fluidity of normal human discourse, they can subsume some of the burden that is usually placed on users (for performing session management or role definition, for example), and they can behave intelligently in the face of user activity.

I believe that this research is best evaluated by examining how well it addresses known needs at each layer of definition: application requirements, coordination features, and foundation features. In terms of application requirements, Intermezzo specifically addresses and provides a solution for known problems cited in the literature:

- The need for awareness. Intermezzo provides a comprehensive infrastructure for the sharing of coordination information as it relates to awareness.
- Session management infrastructure. Intermezzo provides pan-application session management services to collaborative tools.
- Flexible support for roles. Support for roles, policies, and definition of access control primitives are embedded in the toolkit, so that applications don't have to recreate these constructs repeatedly.

In addition to fulfilling application needs in a number of areas, Intermezzo provides some novel capabilities for collaborative applications that have not been fully explored before. I believe that the two most important of these are the implicit session management services, and support for dynamic roles. Implicit session management provides a new paradigm for rendezvous in collaborative settings, and moves much of the burden of session creation from the user to the system. Dynamic roles support fluid interactions among participants in a collaboration, and again can relieve the users of a collaborative system from the burden of having to specify and anticipate role membership. Both of these features follow a common thread in this research: by bringing information about users in the “real world” into the infrastructure, the system can be made sufficiently intelligent to take over some tasks normally delegated to users.

I believe that much of the power represented by the coordination features provided by Intermezzo results from their tight integration. These three features together possess a utility that is greater than the features taken independently. The three features interact, share information, and inform each other.

Finally, I have shown that not only are these features usable and worthwhile, they are also implementable. By starting “from scratch,” I have been able to identify exactly the foundation components that are required to implement the coordination features, and provide justifications for each. If I had based my coordination features on some already-existing substrate (such as a distributed application toolkit, or an object database system) this would not have been the case. Further, it is entirely possible that the resulting coordination features would have been influenced by the tools available. (As the saying goes, when all you have is a hammer, everything begins to look like a nail.)

Contributions

My contributions to the field of computer-supported cooperative work lie in a number of areas. The first, and most abstract, is in the establishment of new paradigms for collaborative infrastructure. The taxonomy of the information sharing needs of application is novel and, I believe, a useful way to look at collaboration. Under this taxonomy we can see that most systems have focused nearly entirely on application information sharing, often to the exclusion of coordination support.

Next, the taxonomy of session management systems is also new in the literature. This classification provides a not only a way to think about session management, but emphasizes the trade-offs that must be addressed when selecting a session management paradigm: what metaphors are best supported by each session management form? What forms are appropriate for formal collaboration? This taxonomy has also introduced several new forms of implicit session management that, when they have been addressed in the literature, have not been identified as forms of session management at all. These forms include artifact-based and locale-based session management.

The final new paradigm is the Intermezzo model for roles and policies. The notion of dynamic roles is a new one, and has the potential to greatly enhance the flexibility of applications in access control situations.

The next broad contribution is the identification of several types of coordination information, and a comprehensive exploration of the use of the information and interaction among features. I have examined how a set of information about users and their activities (as represented by activity records and their associated resources) can be used throughout a coordination infrastructure to enhance collaboration.

This information is useful for awareness, session management, and policy, and I have investigated each of these coordination features in depth. Intermezzo provides comprehensive support for each of these features, and supports interaction among them (the application of policy to session management, or the use of awareness data as an input to policy mechanisms, for example).

My third broad contribution is in the area of infrastructure implementation. Once these coordination features have been identified as useful and important, how can they be implemented economically (in terms of code volume and time-to-implement) and efficiently. This research provides an exposition of a set of foundation features that can be used to implement an array of higher-level features for collaboration. The shared resource data model provides a substrate for information sharing. The attributes of this model—consistency, persistence, access control, and so on—are all essential for the smooth operation of the coordination features, and I have justified the existence of each. The Intermezzo foundation layer provides building blocks for the construction of coordination support in collaborative environments.

My final contribution lies in the area of programming models for coordination. This research has generated a set of programming interfaces that allow applications to take advantage of coordination services in the runtime environment. These interfaces allow developers to easily construct applications that can interact fluidly with other collaborative applications and other users; applications can respond to changes in the environment and behave intelligently in the presence of information about users and their activities. As an example, the notification services provided to application writers are designed to support rich interaction with the "outside" world; they are powerful, flexible, and easy to use.

Caveats

While this work makes important contributions to the field of computer-supported cooperative work, there are a number of weaknesses with the approaches taken here. This section outlines what I believe are the most severe limitations of my research.

The first is a potential weakness in coverage. I have examined a number of forms of coordination information (as represented by the activity-based awareness model and supporting coordination features) that have proven useful for my goals. But I believe that bringing even more information into the environment can provide powerful support for applications. My goals in the selection of information (about user tasks, location, collaborative artifacts, and so forth) were guided by a desire to build a set of common services for applications. With more experience, it is entirely possible (and even probable) that application writers could benefit from other forms of information that may be of specific use to collaborative tools. Some forms of coordination information that either are not useful for the infrastructure itself, or may have simply been overlooked, may prove useful to applications that wish to partake in rich coordination services.

A second weakness lies in the way coordination information is collected. The activity-based modeling approach requires the participation and cooperation of applications to be successful. The utility of the entire infrastructure will decrease significantly if applications do not participate in the service: information will be out of date, inaccurate, or incomplete, and users will not be able to trust the world picture presented to them by the coordination infrastructure.

This weakness can be offset somewhat through the use of controller applications that act on the behalf of non-participating tools. But the problem with information collection still represents a limitation to the approaches used by Intermezzo.

A third weakness, which is also related to the information collection infrastructure in Intermezzo, is that users can lie. That is, any unscrupulous user can forge activity information and place it in the environment. While the access control and authentication functions provided by Intermezzo make it impossible for one user to impersonate another, or to gain illegitimate access to another's data, users can still freely create data in the activity space with impunity.

One possible solution to this problem would be to only allow privileged clients the ability to generate activity information. In essence, this would absolve applications of the responsibility (and the ability) to create activity information; a special controller would be responsible for all information collection. While this solution removes the ability of users to forge activity records, it also removes the ability of applications to generate and keep up-to-date their own activity records. No controller application will be able to populate an activity record as completely and accurately as applications themselves.

Another limitation of the work revolves around the use of the policy specification language introduced by Intermezzo. The policy specification language is powerful, and provides an economical way to create new policies and roles for collaborative applications—previously developers had to “hard code” support for policies and roles into their applications. Creating policies through specification is certainly much simpler. The problem is that while the specification language is much easier than developing code, it is still not suitable for end-user use. Ideally we would like users to be able to create their own policies, and define their own roles, as the need arises. Users should be able to selective enable or disable access control without having to learn a new language, and without running the risk of inadvertently exposing their data to unwanted access.

The specification mechanisms used by Intermezzo do not provide a method for easy end-user configuration. It may be possible to build a “policy manager” tool that interacts with users and emits policy specifications according to user desires. I have not, however, investigated such a tool. The entire issue of how users *think* about policies and roles, and how to capture those concepts in an easy-to-use policy tool, would make an interesting area of future research.

A final weakness is that I have not investigated how a system designed to support coordination can be integrated with “traditional” (application-oriented) CSCW toolkits. Intermezzo is a “stand-alone” system in the sense that it provides its own set of services to applications and does not interact with any other application toolkits. In the Intermezzo model, application writers would use Intermezzo as their toolkit for performing coordination work, and a separate toolkit for application sharing needs (view consistency and the like).

I have focused on coordination to the exclusion of other forms of information sharing because I felt that coordination was an important area that had not been fully explored. Studying coordination separately from application-oriented issues allowed me to study application needs and infrastructure support for coordination without the influence of any other requirements or set of constraints.

In an ideal world, however, collaborative applications would be built using one infrastructure that subsumes all functionality required for collaboration. Certainly much benefit can be derived from a closer integration of application- and coordination-sharing facilities. Several of these benefits have been alluded to in this work. For example, the coordination policy system could be used to “inform” or control the per-application access control system that regulates access to domain-dependent constructs. Many of the approaches used by Intermezzo (dynamic roles for example) have utility beyond the domain of coordination—these techniques are equally useful in application situations.

Certainly the session management infrastructure supported by Intermezzo could be integrated with an application toolkit. Even with Intermezzo, applications must still decide what domain-specific behavior will occur when a potential collaboration arises; several collaborative toolkits have addressed the construction of these behaviors. By integrating the coordination functions related to detecting potential collaboration with the application functions of implementing specific behaviors during rendezvous, application writers will more easily be able to take advantage of robust and powerful session management facilities.

Intermezzo provides its own set of programmatic interfaces for constructing applications. Tighter integration with other programming models (whether for application-oriented sharing, distributed systems programming, or even window system programming) will allow developers to better leverage their understanding of these other programming models.

To summarize, while Intermezzo introduces many new concepts that are useful for collaboration, there is still work to be done in a number of areas. Problems related to scope, integration, and usability should be addressed by future research.

Chapter VIII

Future Directions

In this work I have explored the ways in which applications can benefit from the presence of “situational,” or real world, knowledge about users, their actions, and their environment. By bringing such situational information into the runtime infrastructure, applications can be written to be situationally-aware. That is, they can behave intelligently in the face of information about user activity. Further, this information can be “packaged” into constructs that are commonly used in collaborative applications: session management for example. The information used for coordination can be used in novel ways to enhance the fluidity of interactions among users, their applications, and the collaborative environment.

There are, however, a number of issues that are raised by this research but could not be addressed fully. This chapter investigates some of these issues and points to potentially fruitful areas of future research based on this work.

Asynchronous and Autonomous Collaboration

While many of the issues of coordination are not dependent on whether applications are built for synchronous or asynchronous collaboration, much of this research has been implicitly intended for use in synchronous collaborative environments. I feel that there are a number of areas of future research that have the potential to enhance the ability of a system like Intermezzo to support asynchronous (off-line) or autonomous (rarely-connected) collaboration.

Perhaps the most important area where this work can be extended to support off-line collaboration is through the support of additional consistency policies. The consistency policies currently supported by Intermezzo provide only synchronous updates (via the atomic synchronous or lazy synchronous policies), or no consistency at all. Asynchronous and autonomous collaboration have different consistency needs that are not fulfilled by the services Intermezzo provides. Work by Kolland has begun to investigate consistency issues for these forms of collaboration [58].

Another area where Intermezzo can be extended to support off-line collaboration is embedded computation. See the section below for more details.

Further Investigation of Embedded Computation

Intermezzo uses the ability to download code into the runtime environment for a number of purposes: server-side trigger functions, dynamic role predicate functions, and so on. The ability to securely execute downloaded code from clients can enhance the process of developing client code substantially.

In many ways, though, Intermezzo has not explored the full potential of embedded computation in a collaborative environment. Two uses for embedded computation seem obvious. The first was alluded to above: users who are rarely connected to the runtime environment can download code into an entity that is always connected and relatively long-lived (such as the server process). This code acts as an agent on the user's behalf, collecting information about "interesting" events that may have occurred while the user was disconnected. The agent can then transmit the information to the user once a connection is reestablished.

This use of embedded computation lends great power to the system in non-synchronous situations. Users who are mobile (and thus rarely connected) or in different time zones (and are thus not connected at the same time as most other users) can build elaborate systems to act on their behalf while they are off-line. A complete "asynchronous proxy" development environment could be an interesting avenue of research.

The second use for embedded computation that has not been fully explored is as a notification mechanism. This use was mentioned in the discussion of other notification schemes (events and triggers) but its use has only been briefly studied by Intermezzo. As has been stated several times in this work, one of the defining characteristics of coordination and collaboration support is that the environment must be able to cope with the fluidity, unforeseen circumstances, and potential for surprise that exist in multi-user settings. Toward that end, the facilities provided by Intermezzo have been designed for maximum expressive power and flexibility (the pattern matching scheme used by events for example).

Embedded computation represents the most general case possible for a notification system. One can imagine a "toolbox" of pre-constructed notifier agents that can be downloaded into the server to watch for interesting occurrences. These notifiers can react in arbitrarily intelligent ways to circumstances that arise in the environment; their potential power goes far beyond what is possible with any simple event notification system.

I believe that uses for embedded computation remain an area ripe with possibilities for collaboration and coordination.

Distributed Infrastructure

As stated previously, my implementation of the runtime coordination infrastructure provided by Intermezzo is not based on a distributed architecture. I felt that, for a research prototype, the benefits of scalability that could be realized in a true distributed system were outweighed by the complexities of actually constructing such a system. My approach (a centralized server with replicated data cached in clients) allowed me to explore issues related to performance in a networked setting, and with programming models that must cope with multiple copies of data, without incurring the effort of creating a true distributed infrastructure.

I do believe, however, that a distributed architecture is required for any “real life” Intermezzo-like system. The primary goal of any collaboration support environment is to allow easy access of users to collaborative settings. If the environment cannot scale well, is limited by its performance, or cannot support a wide enough array of users to provide real value, it will not be adopted.

Identification of Other Types of Coordination Information

This research has focused on three types of coordination information designed to support awareness, session management, and policy controls. I am by no means suggesting that these are the *only* forms of coordination information possible, or useful, in a collaborative setting, however. The identification and investigation of other forms of information useful to coordination is a fruitful area of future research.

There are a number of open questions that may be addressed by such an investigation. First, will other forms of coordination information be supported by the foundation infrastructure model used by Intermezzo? The Intermezzo foundation layer was designed to support the sharing of the particular forms of coordination information I have studied in depth; new forms of coordination information may require the augmentation of the foundation layer, or even entirely new and different services.

Second, what types of application conventions and runtime support will be needed to utilize new forms of coordination information effectively? Intermezzo applications use a number of conventions that are built into the toolkit (conventions about resource attributes for session management, for example). What new conventions must applications follow to adopt new methods of coordination?

Higher-Level Policy Specification

The policy mechanisms provided by Intermezzo provide a powerful and expressive tool for describing how applications will behave in the face of input from users and other applications. I believe that the Intermezzo facilities—specification languages for roles and policies, support for static and dynamic roles—provide a powerful environment for creating new policies that is far simpler to use than any alternative system (which would typically hard-code policies within applications).

I am under no illusion that the policy specification language is suitable for end-users though. Certainly end-users should not be troubled with or be expected to know about resources, attributes, and access control list parameters. Further, since policy is defined in terms of access control, there is the potential for disaster if users tinker with their policy settings and unwittingly use a too-lax access control setting for their personal data. A misstep in setting policy could open the way for the destruction (either intentional or accidental) of user information, possibly resulting in users not trusting the system with their confidential data.

I believe that some higher-level form, or even automatic generation, of policy specifications would be useful. Ideally, only application writers and perhaps “power users” would actually see the policy specification language. General purpose policy tools would have the task of generating policy specifications for controlling access to user information.

Coordination and Application Interactions

In this research I have chosen to focus almost solely on the issues related to providing runtime and toolkit support for coordination, to the exclusion of support for application information sharing. In reality, I believe that a single infrastructure for providing both coordination and application support is better in the long run (once issues related to these two forms of information sharing, and their interactions, have been sufficiently addressed).

There are a number of ways in which a toolkit for coordination could “inform” or influence an application sharing toolkit. One of the most obvious ways is in the area of access control. The coordination system can analyze situational variables to determine the roles of a set of participants in a collaboration. This information can then be processed by an application toolkit to generate a set of domain-dependent roles and access control rights. In essence, the highly-dynamic coordination system is used to set the parameters on the more static (but speedier) application toolkit constructs.

The potential for interactions between the coordination and application hemispheres exists any time an application-dependent construct can benefit profitably from knowledge about the state of users, their actions, and their environments.

Appendix A

C++ Language Interface

As described earlier in this work, the Intermezzo runtime service is constructed with exactly the same facilities as are used in the Intermezzo client-side library. Thus, the total facilities of the server are directly available in the client address space.

Because of the richness of the facilities available to clients, the complete C++ application programming interface (API) is quite large. In practice, however, most clients will only use a small subset of the features available to them.

This appendix describes the higher-level (coordination-specific programming interfaces available to developers. These APIs are built on top of the “mid-layer” interfaces: handle objects as proxies for resources, local and shared contexts as containers for resources, and so on. The handle and context interfaces, as well as the lower-level interfaces (such as threads, callbacks, and events) are not described here in detail in the interests of brevity.

Awareness

The Intermezzo awareness APIs are primarily concerned with making it easy for developers to publish activity records into the shared data space. Recall that an activity record consists of subject, verb, and object resources representing the user, application, and artifact, respectively.

The publication of activity data is a two-stage process. First, a set of “generation” routines is used to create resources of the appropriate type, fill in as many slots as possible, and return a handle to the newly-generated resource. At this point the resource is contained only in the local context of the client (that is, it is not yet shared among all clients).

At this point, clients are free to update, change, or modify their activity-related resources as needed.

Next, the collection of resources that make up the activity record are all published into the shared context. At this point, the activity record is visible to any clients connected to that shared context, with the appropriate permissions. Clients that have installed triggers or events to detect changes in the world state will be notified.

```
Handle = GenerateSubject(LocalContext& l)
```

The `GenerateSubject` procedure creates a new subject resource and returns a handle to it. The slots of the resource are filled from information available in the environment of the application, and from the operating system.

```
Handle = GenerateVerb(LocalContext& l, int argc, char *argv, char *envp)
```

`GenerateVerb` creates a representation of the application being run. The caller of the function must pass in the command line and environment to this routine. GenerateVerb will extract portions of the command line and environment that are salient to the slots of the verb resource.

```
Handle = GenerateFileObject(LocalContext& l, const String& path)
```

`GenerateFileObject` creates a new object resource representing a file, given a path to that file.

```
Handle = GenerateNullObject(LocalContext& l);
```

The `GenerateNullObject` routine creates an untyped object resource. This resource can be published "as-is" (to indicate that the current activity involves no artifact), or it can be modified in a domain-dependent manner to represent application-specific data.

```
Handle = GenerateActivity(LocalContext& l, Handle& s, Handle& v, Handle& o)
```

The `GenerateActivity` routine creates a resource representing an activity, given references to existing subject, verb, and object resources.

```
Handle = PublishActivity(LocalContext& l, SharedContext& s, Handle& act)
```

`PublishActivity` makes an existing activity resource visible in the shared context.

Session Management

As mentioned earlier in this work, session management control is accomplished by setting slots on object and verb resources. Thus, there is no direct and visible API to these features: programmers simply update resource attributes to control session management; the server does the rest.

There is one session management API that can be used for "by-hand" implicit session management or (more likely) for debugging.

```
Status = SolicitSMEvent(SharedContext& s, Handle& a, Callback& cb)
```

The SolicitSMEvent routine is used to supply a callback instance to the runtime system; this callback will be executed whenever a confluence is detected on the activity record specified by the provided handle.

Policy

Much like session management, policy is largely hidden from the C++ developer. Resource access control lists that result from policies are generated and assigned automatically, with no programmer intervention, from the policy descriptor files loaded at startup time.

Developers can control access lists "by hand" by calling directly into specific resources. In general, however, the policy layer is accessed only through the policy description language, and not through C++.

Appendix B

Python Language Interface

This appendix describes the Python language bindings for Intermezzo. Intermezzo supports Python access to the objects in the foundation layer, through the use of Python *proxies* for the native C++ objects used by Intermezzo.

This appendix is organized as a reference to the objects available through Python. The Python-to-C++ interface used by Intermezzo supports arbitrary bidirectional conversion between “native” Intermezzo types in C++ and the built-in Python types. Further, trigger functions can be specified directly in Python.

Intermezzo Module

The Intermezzo module, `intermezzo`, is a shared object module, implemented in C++, that can be imported into a running Python executable. The `intermezzo` module provides its own exception variable, and several module-scoped functions that are useful to Intermezzo clients written in Python.

`intermezzo.idFromString`

This method is used to generate an ID object from a string. It is mostly useful for debugging—generating handles to objects to bootstrap a test program, for example.

`intermezzo.mutexDebug`

This method toggles the Intermezzo mutual exclusion debugging code. When mutex debugging is enabled, the system generates diagnostic reporting about thread conditions.

`intermezzo.errorLevel`

This method allows the user to set the amount of diagnostic output that will be generated by the Intermezzo runtime library.

`intermezzo.localContext`

The `localContext` method creates a new local context (client-side resource cache). The local context is represented by a new Python proxy object (see below).

`intermezzo.sharedContext`

The `sharedContext` method creates a new connection to a shared context (server-side shared resource pool). Objects can be moved between local and shared contexts. The shared context is represented by a new Python proxy object (see below).

ID Class

The id class provides a Python type for representing resource identifiers. It has no methods, and is only used by other objects in the intermezzo module.

Handle Class

Handles are “safe” references to resources that support reference counting, exception support, and several “nice” initialization properties. Handles are used in place of direct resource references in the Python API. The `handle` class supports the ability to explicitly decrement the reference count on a handle, which is useful for hybrid Python/C++ applications to ensure proper object deletion.

`handle.initialized`

> `Initialized` returns a boolean value, indicating whether or not this handle instance has been initialized to a valid resource object.

`handle.release`

> This method is used to explicitly decrement the reference count on a handle.

`handle.id`

> The `id` method returns the ID object associated with this handle. IDs are represented by independent Python objects.

`handle.getPersistence`

> This method returns the current persistence setting for this handle.

`handle.setPerisistence`

> This method sets the persistence setting for this handle.

`handle.installTrigger`

> The `installTrigger` method is used to install either resource or attribute triggers on a handle. It takes a specification for either a resource or an attribute on which the client wishes to install a trigger, as well as the trigger code itself, specified in Python. The trigger will be evaluated whenever the resource or attribute is accessed. The method returns an identifier for the trigger.

`handle.removeTrigger`

> This method removes a trigger function from a resource or attribute, given the identifier of the trigger.

Local Context Class

Intermezzo supports the notion of a “local context” as a client-side, non-shared repository or scratch space for resource data. Contexts provide a means for resources to be created, copied, and freed. Further, resource data can be moved between contexts, via the `clone` operation (described below).

`localContext.create`

> The `create` method creates a new resource in the local cache and returns a handle to it.

`localContext.clone`

> `Clone` makes a copy of a resource in the local cache.

`localContext.find`

> `Find` searches for resources in the local cache, given a specification pattern (see below). The method returns a list of handles to the matched resources.

`localContext.free`

> `Free` explicitly destroys a resource contained in the local cache. The method takes the ID of the resource as an argument.

`localContet.release`

`Release` explicitly decrements the reference count on a handle, given an ID as an argument. This method is a short-cut for handle.release.

Shared Context Class

A "shared context" represents the complete shared data space maintained by the Intermezzo runtime server. In Python, a shared context is accessed through the `sharedContext` object. `SharedContexts` are created by specifying the host on which the context server resides, and an instance of a `localContext` (described above) to be used as a local cache by the `sharedContext`. `SharedContexts` support the same operations as `localContexts`.

`sharedContext.create`

The `create` method creates a new resource in the shared object space and returns a handle to it.

`sharedContext.clone`

`Clone` makes a copy of an existing object in the shared data space.

`sharedContext.find`

`Find` searches for resources matching a specification (see below). It returns a list of handles to all matched resources.

`sharedcontext.free`

`Free` explicitly deletes an object from the shared context, given its identifier.

`sharedContext.release`

The `release` method is used to decrement the reference count on a handle It is a short-cut for handle.release.

File ID Class

The Intermezzo session management service commonly uses representations of files for implicit session management. In Python, the file ID tuple described in this work is represented as a `FileID` object. These objects can be generated on-the-fly, given the path name of an existing file.

`fileID.host`

Returns the "host" component of a given file identifier tuple

`fileID.filesys`

Returns the "file system" component of a given file identifier tuple.

`fileID.inode`

Returns the "inode" component of a given file identifier tuple.

Resource Specifier Class

Resource specifiers are implemented in Python as lists of tuples (tuples and lists are native Python types).

Appendix C

Server Protocol

This chapter describes the protocol that is exchanged between the server and client applications. The server protocol is used to maintain the shared object store used by applications.

The protocol exchanges messages, which may be of several types:

•*Undefined*. Used for the free-form interchange of client-specific data between applications. The server protocol uses no *undefined* messages.
•*Request*. A message from a client to the server to request some particular service.
•*Reply*. A response from the server to a client, always generated in response to a *request*.
•*Event*. An asynchronous (non-requested) message from the server to a client.
•*Notify*. A message from a client to the server that does not necessitate a *reply*.

Hello

The Hello message is used by an application to introduce itself to the Intermezzo server, establish client identifiers, and exchange authentication information.

The format of the request to the server is as follows:

Item	**Type**	**Description**
Host Name	String	The network host the client program is running on.
Domain Name	String	The network domain that contains the host.
Certificate Name	String	The claimed certificate of the user running the client program.
Cypher Text	Binary	An encrypted representation of an MD5 hash of the certificate.

The format of the reply from the server is as follows:

Item	Type	Description
Success	Status	An indication of whether the client introduction succeeded.
Client ID	Owner	The unique identifier to be used by this client.

Goodbye

The Goodbye message is used to indicate to the server that a client is shutting down. The request contains no data (only the Goodbye opcode is used). The reply from the server contains the following:

Item	Type	Description
Success	Status	An indication of whether the client disconnect succeeded cleanly (disconnect may not succeed cleanly if the client still holds locks, for example).

Insert

The Insert request is used to create new resources in the server's shared data space. It is generated from the Create() and Clone() APIs. The server takes the list of resources contained in the reply, verifies them, and installs them in the shared data store. The request to the server has the following format:

Item	Type	Description
Resource List	List of Resources	A set of resources (created within the client) to be "published" in the server's shared data space.

The reply from the server has the following format:

Item	Type	Description
Success Codes	List of Status	An indication of whether the resource insertions succeeded (insertion may fail if a resource is improperly formatted, or contains bogus ownership data). There is one Status for each input Resource.

Remove

The Remove request is used to explicitly free a set of resources from the server's shared data store. The server verifies resource existence and access control before removing resources.

The request to the server has the following format:

Item	Type	Description
Resource IDs	List of ID	A list of the set of resource IDs to be freed. The server will verify existence and access rights before removing the resources.

The reply from the server has the following format:

Item	Type	Description
Success Codes	List of Status	An indication of whether the resource removals succeeded. Removal will fail if the client lacks sufficient access rights, or if the resource does not exist. There is one returned status code per input ID.

CheckOut

CheckOut is used to establish a write lock on a resource. Once a resource is checked out, no other client may update it until the lock is release via the CheckIn request. CheckOut will return the most recent copy of the resource, if the client's replica was created with lazy synchronous consistency or stronger.

The format of the request to the server is as follows:

Item	Type	Description
Resource ID	ID	The ID of the resource to be checked out. The server will verify existence and access rights before checking out the resources.

The format of the reply from the server is as follows:

Item	Type	Description
Success Code	Status	An indication of whether the resource was checked out. Check out may fail if the resource does not exist, or if the client lacks sufficient permissions.
Resource Data	Resource	A "snapshot" of the current state of the resource (only returned if the success code is TRUE and the client consistency policy requires it).

CheckIn

CheckIn releases a write lock on a resource, thereby permitting other locks to be established. CheckIn returns a new, updated copy of the resource to the server, which becomes the new "authoritative" version.

The request to the server has the following format:

Item	Type	Description
Resource Data	Resource	The new snapshot of the resource on which the lock was held.

The reply from the server has the following format:

Item	Type	Description
Success Code	Status	An indication of whether the new version of the resource was accepted as authoritative. Check in may fail if the client attempts to update resource components on which it has no permissions, or if the client attempts to change the ownership or ID of the resource.

ReadLock

The ReadLock request is used to establish a lock on a resource for purposes of reading. The server allows any number of clients to have simultaneous read locks at a given time, but only one write lock can be in effect at once.

The format of the request to the server is:

Item	Type	Description
Resource ID	ID	The ID of the resource to be write locked.

The format of the reply from the server is:

Item	Type	Description
Success Code	Status	An indication of whether the read lock was successfully acquired (the lock may fail if the resource does not exist or if the client has insufficient permissions).
Resource Data	Resource	The current snapshot of the resource (only returned if the success code is TRUE and the client consistency policy requires it).

ReadUnlock

The ReadUnlock request is used to release a read lock on a resource. The format of the request to the server is:

Item	Type	Description
Resource IDs	ID	The ID of the resource on which the lock should be released.

The format of the reply from the server is:

Item	Type	Description
Success Code	Status	An indication of whether the release of the lock was successful (the request may fail if the resource did not exist or was not locked).

Resolve

The Resolve request retrieves a resource given its identifier. This request is much like ReadLock, only a lock is not actually placed on the object—Resolve returns an instantaneous “snapshot” of the resource and does not ensure concurrent access. This call is primarily useful for debugging. The format of the request is:

Item	Type	Description
Resource ID	ID	The ID of the resource to be resolved.

The format of the reply from the server is:

Item	Type	Description
Success Code	Status	An indication of whether the resolution was successfully acquired (the resolve operation may fail if the resource does not exist or if the client has insufficient permissions).
Resource Data	Resource	The current snapshot of the resource (only returned if the success code is TRUE).

Interpret

The Interpret request is used to request remote execution of a downloaded code segment by the server. The format of the request from the client is as follows:

Item	Type	Description
Code	String	The code segment to be executed.

The format of the reply from the server is:

Item	Type	Description
Success Code	Status	An indication of whether the execution succeeded or failed.
Result	String	The results of the remote execution, returned as a string.

Solicit

The Solicit request is used to perform a “time-forward” event subscription. The server installs the event specification pattern and will issue events to the client whenever the pattern is matched.

The format of the request from the client to the server is:

Item	Type	Description
Event Specification	Spec	A description specifier of the event the client is subscribing to.

Events are generated asynchronously from the server as they occur. The format of the reply from the server is:

Item	Type	Description
Success Code	Status	An indication of whether the specification was successfully installed.
Spec ID	ID	A unique identifier that may be used to refer to this specification.

Search

The Search request is the “time-backward” cousin to Solicit. Search is used to scan over the shared object store, attempting to match resources described by the specification pattern. The server will return all matching resources that the client has permission to examine.

The format of the request is:

Item	Type	Description
Event Specification	Spec	A description specifier of the resources the client is searching for.

The format of the reply is:

Item	Type	Description
Success Code	Status	An indication of whether the search succeeded (the search may fail if the specification is mis-formatted).
Resource List	List of Resources	All resources matching the specification pattern that the client has permission to retrieve.

List

List is essentially a Search with a wildcard specification—it returns all resources in the shared data store to which the client has permission to read. The request contains no data, other than the List request opcode.

The format of the reply is:

Item	Type	Description
Resource List	List of Resources	A list of all resources in the shared data store that the client has permission to examine.

REval

The REval request is used to test resource access control rights explicitly. The client issues a descriptor of a requested action on a particular resource. The format of the request is:

Item	Type	Description
Actor	Owner	The owner who is requesting a particular access. The server will disregard the request if the actor specified here is not the owner of the client application.
Resource ID	ID	The ID of the resource to be tested.
Requested Action	Rights	A description of the requested action (read, write, delete, existence check).

The server responds with the following:

Item	Type	Description
Success Code	Status	An indication of whether the requested access would be allowed.

AEval

The AEval request is used to explicitly test access control rights for an attribute on a resource. The format of the request is

Item	Type	Description
Actor	Owner	The owner who is requesting a particular access. The server will disregard the request if the actor specified here is not the owner of the client application.
Resource ID	ID	The ID of the resource to be tested.
Attribute	Key	The name of the attribute to be tested.
Requested Action	Rights	A description of the requested action (read, write, delete, existence check).

The server responds with the following reply:

Item	**Type**	**Description**
Success Code	Status	An indication of whether the requested access would be allowed.

References

[1] Ackerman, Mark S., “Augmenting the Organizational Memory: A Field Study of Answer Garden.” *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, NC: ACM, October 22-26, 1994. pp. 243-252.

[2] *The American Heritage* Dictionary. Boston, MA: Houghton Mifflin Company.

[3] Baecker, R.M., Nastos, D., Posner, I.R., and Mawby, K.L., “The User-centered Iterative Design of Collaborative Writing Software.” In *Proceedings of the ACM/InterAct Conference on Human Factors in Computing Systems* (INTERCHI’93). Amsterdam, The Netherlands: ACM. April 24-29, 1993. pp. 399-405.

[4] Birman, K., Cooper, R., Joseph, T., Kane, K., Schmuck, F. *The ISIS System Manual*, June 19, 1989.

[5] Bly, S.A., Harrison, S.R., and Irwin, S., “Media Spaces: Bringing People Together in a Video, Audio, and Computing Environment.” *Communications of the ACM*, Vol. 36, No. 1 (January 1993), pp. 28-47.

[6] Borning, A., and Travers, M., “Two Approaches to Casual Interaction over Computer and Video Networks.” *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI’91), New Orleans, LA: ACM, 1991, pp. 13-19.

[7] Brothers, L., Sembugamoorthy, V., Muller, M., “ICICLE: Groupware for Code Inspection.” *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 1990, pp. 169-181.

[8] Clement, A., Kling, R. “Working Notes from a Workshop on Privacy Considerations in CSCW,” workshop notes from *Sharing Perspectives: Conference on Computer-Supported Cooperative Work, CSCW’92*, Toronto, Ontario.

[9] Conklin, Jeff, “Hypertext: An Introduction and Survey.” *Computer-Supported Cooperative Work: A Book of Readings*, Irene Grief, ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 423-475.

[10] Conklin, Jeff, and Begeman, Michael L. “gIBIS: A Hypertext Tool for Exploratory Policy Discussion.” *CSCW 88: Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM, 1988, pp. 140-152.

[11] Cool, C., Fish, R.S., Kraut, R.E., and Lowery, C.M. “Iterative Design of Video Communication Systems,” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work, CSCW’92*, Toronto, Ontario: ACM, 1992, 25-32.

[12] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R. “MMConf: An Infrastructure for Building Shared Multimedia Applications.” *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 1990, pp. 329-342.

[13] Curtis, Pavel, *LambdaMOO Programmer’s Manual*, available as ftp://ftp.parc.xerox.com/pub/MOO/ProgrammersManual.ps

[14] Curtis, Pavel, and Nichols, David A., “MUDs Grow Up: Social Virtual Reality in the Real World.” In the *Proceedings of the 1994 IEEE Computer Conference*, pp. 193-200, January 1994. Also available as ftp://ftp.parc.xerox.com/pub/MOO/papers/MUDsGrowUp.ps.

[15] Curtis, P., Dixon, M., Frederick, R., and Nichols, D.A., "The Jupiter Audio/Video Architecture: Secure Multimedia in Network Places." In *Proceedings of the Third ACM International Multimedia Conference*, San Francisco, CA: ACM. November, 1995, pp. 79-90.

[16] Davies, D.W., and Price, W.L., "The Application of Digital Signatures Based on Public-Key Cryptosystems," *Proceedings of the Fifth International Computer Communications Conference*, October 1980, pp. 525-530.

[17] Dewan, Prasun, "Designing and Implementing Multiuser Applications: A Case Study," *Software-Practice and Experience*, 23:1, pp. 75-94, 1993.

[18] Dewan, P., and Choudhary, R. "Flexible User Interface Coupling in a Collaborative System." *Reaching Through Technology, Proceedings of ACM CHI '91: Conference on Human Factors in Computing Systems*, New Orleans, LA: ACM, 1991, 41-48.

[19] Dewan, P., and Choudhary, R. "Primitives for Programming Multi-User Interfaces." *UIST 91: Proceedings of the ACM Symposium on User Interface Software and Technology*, Hilton Head, SC: ACM, 1991, 69-78.

[20] Dewan, P. and Choudhary, R. "A Flexible and High-Level Framework for Implementing Multiuser User Interfaces." *ACM Transactions on Information Systems* 10:4, New York, NY: ACM, pp. 345-380.

[21] Dewan, Prasun. "An Inheritance Model for Specifying Flexible Displays of Data Structures," *Software Practice and Experience*, Vol. 21, No. 7, July 1991. pp. 719-738.

[22] Dewan, Prasun. "A Tour of the Suite User Interface Software," *Proceedings of the 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology*, October 1990, pp. 57-65.

[23] Dewan, Prasun, Choudhary, Rajiv, and Shen, HongHai, "An Editing-based Characgerization of the Design Space of Collaborative Applications." *Journal of Organizational Computing*, 4:3, pp. 219-240, 1994.

[24] Dewan, Prasun, and Vasilik, Eric, "An Object Model for Conventional Operating Systems." *Computing System*s, 3:4, pp. 517-549, 1990.

[25] Dixon, G.N., Shrivastava, S.K., Parrington, G.D. "Managing Persistent Objects in Arjuna: A System for Reliable Distributed Computing." *Proceedings of Workshop on Persistent Object Systems*, Persistent Programming Research Report, No. 44, Department of Computational Science, University of St. Andrews, August 1987.

[26] Dixon, G.N., Parrington, G.D., Shrivastava, S.K., Wheater, S.M. "The Treatment of Persistent Objects in Arjuna." *Proceedings of the Third European Conference on Object-Oriented Programming, ECOOP89,* University of Nottingham, pp. 169-189, July 1989.

[27] Dourish, Paul, and Bellotti, Victoria, "Awareness and Coordination in Shared Workspaces." *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW'92, Toronto, Ontario: ACM, pp. 107-114.

[28] Dourish, Paul, and Bly, Sara, "Supporting Awareness in a Distributed Workgroup." *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI'92), Monterey, CA: ACM, pp. 541-547.

[29] Edwards, W. K., "The Multimedia Applications Kit/C++." Unpublished Sun Microsystems Laboratories Technical Report, 1991.

[30] Edwards, W. K., "A Session Management Service for Collaborative Applications." Unpublished Sun Microsystems Laboratories Technical Report, 1992.

[31] Edwards, W.K., “Session Management for Collaborative Applications.” In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW’94), Chapel Hill, NC: ACM, October 22-26, 1994. pp. 323-330.

[32] Ellis, C.A., Gibbs, S.J., and Rein, G.L. Design and Use of a Group Editor. In *Engineering for Human-Computer Interaction* (G. Cockton, Editor), North-Holland, Amsterdam, 1990, 13-25.

[33] Ellmer, E., Pernul, G., Quirchmayr, G., Tjoa, A.M., “Access Control for Cooperative Environments.” Workshop on Distributed Systems, Multimedia, and Infrastructure, ACM Conference on Computer-Supported Cooperative Work (CSCW’94), Chapel Hill, NC October 22, 1994.

[34] Fish, R.S., Kraut, R.E., and Chalfonte, B.L., “The VideoWindow System in Informal Communications.” Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW’90), October 7-10, 1990. Los Angeles, CA: ACM, pp. 1-11.

[35] Fish, R.S., Kraut, R.E., Leland, M.D.P., and Cohen, M., “Quilt—A Collaborative Tool for Cooperative Writing.” In *Proceedings of the Conference on Office Information Systems* (COIS’88), Palo Alto, CA: ACM. March, 1988.

[36] Gaver, W., “The Affordances of Media Spaces for Collaboration.” In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’92), Toronto, Ontario: ACM, pp. 17-24.

[37] Gaver, W., “Sound Support for Collaboration.” In *Proceedings of the European Conference on Computer Supported Cooperative Work* (ECSCW’91).

[38] Gaver, W., Moran, T., MacLean, A., Lovstrand L, Dourish, P., Carter, K., and Buxton, W., “Realizing a Video Environment: EuroPARC’s RAVE System.” In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI’92), Monterey, CA: ACM, pp. 27-35.

[39] Gibbs, S.J. “LIZA: An Extensible Groupware Toolkit.” *Wings for the Mind, Proceedings of ACM CHI ‘89: Conference on Human Factors in Computing Systems*, Austin, TX: ACM, 1989, 29-36.

[40] Gintell, John W., and McKenney, Roland F., “CSCW Infrastructure Requirements Derived from Scrutiny Project.” Workshop on Distributed Systems, Multimedia, and Infrastructure, ACM Conference on Computer-Supported Cooperative Work (CSCW’94), Chapel Hill, NC October 22, 1994

[41] Gosling, James, “SunDew—A Distributed and Extensible Window System.” Methodology of Window Management (Proceedings of an Alvey Workshop at Cosener’s House, Abingdon, UK, April 1985), Berlin: Springer-Verlag, 1986. pp. 47-57.

[42] Gosling, James, and McGilton, Henry, “The Java Language Environment.” Sun Microsystems White Paper, May 1995.

[43] Graham, T.C.N., and Urnes, T. “Relational Views as a Model for Automatic Distributed Implementation of Multi-User Applications.” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, pp. 59-66.

[44] Greenberg, Saul, and Marwood, David, “Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface.” *Transcending Boundaries: Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, October 22-26, 1994. pp. 207-217.

[45] Grief, I., Ed. *Computer-Supported Cooperative Work: A Book of Readings*. San Mateo, CA: Morgan-Kaufmann, 1988.

[46] Grief, I., and Sarin, S. “Data Sharing in Group Work,” *Computer-Supported Cooperative Work: A Book of Readings*, Irene Grief, ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 477-508.

[47] Grudin, J., and Poltrock, S. “Computer Supported Cooperative Work and Groupware.” Tutorial presented at *ACM SIGCHI Conference on Human Factors in Computing Systems*, New Orleans, LA: ACM 1991.

[48] Grudin, Jonathan, “Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces.” *CSCW 88: Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM, pp. 85-93.

[49] Hammainen, Heikki, and Condon, Chris, “Form and Room: Metaphors for Groupware.” In *Proceedings of the ACM Conference on Organizational Computing Systems* (COOCS’91), Atlanta, GA: ACM, November 5-8, 1991, pp. 95-105.

[50] Isaacs, E.A., Morris, T., and Rodriguez, T.K., “A Forum for Supporting Interactive Presentations to Distributed Audiences.” In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, NC: ACM, October 22-26, pp. 405-416.

[51] Isaacs, E.A., Morris, T, Rodriguez, T.K., and Tang, J.C., “A Comparison of Face-to-Face and Distributed Presentations.” In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI’95), Denver, CO: ACM, May 7-11, 1995. pp. 354-361.

[52] Ishii, Hiroshi, “TeamWorkstation: Towards a Seamless Shared Workspace.” In *Proceedings of the Conference on Computer Supported Cooperative Work* (CSCW’90), Los Angeles, CA: ACM, October 7-10, 1990. pp. 13-26.

[53] Ishii, H., Kobayashi, M., and Grudin, J., “Integration of Inter-Personal Space and Shared Workspace: ClearBoard Design and Experiments.” In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW’92), Toronto, Canada: ACM, October 31-November 4, 1992. pp. 33-42.

[54] Julienne, Astrid M., and Holtz, Brian, *ToolTalk and Open Protocols: Inter-Application Communication*. Des Moines, IA: Prentice-Hall, 1994.

[55] Kaplan, S.M., Tolone, W.J., Bogia, D.P., and Bignoli, C., “Flexible, Active Support for Collaborative Work with ConversationBuilder.” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, pp. 378-385.

[56] Kling, Rob (moderator), “Controversies about Privacy and Open Information in CSCW.” Panel at ACM Conference on Computer Supported Cooperative Work (CSCW’92), October 31-November 4, 1992. Toronto, Ontario.

[57] Knister, M.J., and Prakash, A. “DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors.” *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 343-355.

[58] Koland, Markus, “Distributed System Support for Consistency of Shared Information in CSCW (Extended Abstract).” Workshop on Distributed Systems, Multimedia, and Infrastructure, ACM Conference on Computer-Supported Cooperative Work (CSCW’94), Chapel Hill, NC October 22, 1994.

[59] Lantz, Keith A., “An Experiment in Integrated Multimedia Conferencing.” *Computer-Supported Cooperative Work: A Book of Readings*, Irene Grief, ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 533-552..

[60] Leland, M.D.P., Fish, R.S., and Kraut, R.E., “Collaborative Document Production Using Quilt.” *CSCW 88: Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM, 1988, 206-215.

[61] Manandhar, Sanjay, “Activity Server: You Can Run But You Can’t Hide.” *Proceedings of the 1991 USENIX Conference*, Nashville, TN: USENIX Association, 1991, pp. 299-312.

[62] Mantei, M.M, Baecker, R.M., Sellen, A.J., Buxton, W.A.S., Milligan, T., and Wellman, B. “Experiences in the Use of a Media Space.” *Proceedings of the ACM Conference on Computer-Human Interaction* (CHI), April 28-May2, 1991. New Orleans, LA: ACM. pp. 127-138.

[63] McGuffin, Lola, and Olson, Gary, “ShrEdit: A Shared Electronic Workspace,” CSMIL Technical Report, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1992.

[64] Micallef, Josephine, “Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages.” In *Journal of Object-Oriented Programming* (JOOP), Vol. 1, No. 1, April/May, 1988, pp. 12-36.

[65] Minneman, Scott L., and Harrison, Steve R., “Where Where We: Making and Using Near-Synchronous, Pre-narrative Video.” *Proceedings of the First ACM International Conference on Multimedia*, Anaheim, CA: ACM, August 16, 1993. pp. 207-214.

[66] Mitchell, A., Posner, I., and Baecker, R., “Learning to Write Together Using Groupware.” In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI’95), Denver, CO: ACM, May 7-11, 1995, pp. 288-295.

[67] Moffett, Jonathan D., and Morris, S. Sloman, “The Representation of Policies as System Objects.” In *Proceedings of the ACM Conference on Organizational Computing Systems* (COOCS’91), Atlanta, GA: ACM, November 5-8, 1991, pp. 171-184.

[68] Moran, Thomas P., and Anderson, R.J., “The Workaday World as a Paradigm for CSCW Design.” *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 1990, pp. 381-393.

[69] Narayanaswamy, K., and Goldman, N., “Lazy Consistency: A Basis for Cooperative Software Development.” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, 257-264.

[70] NBS FIPS PUB 46, “Data Encryption Standard,” National Bureau of Standards, U.S. Department of Commerce, January 1977.

[71] *NCSA Collage for the X Window System User’s Guide*, National Center for Supercomputing Applications.

[72] Neuwirth, C. M., Kaufer, D. S., Chandhok, R., and Morris, J. “Issues in the Design of Computer Support for Co-authoring and Commenting.” *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 1990, 183-195.

[73] Neuwirth, C. M., Kaufer, D. S., Chandhok, R., and Morris, J. “Computer Support for Distributed Collaborative Writring: Defining Parameters of Interaction.” *Transcending Boundaries: Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, October 22-26, 1994. pp. 145-152.

[74] Olson, J., Olson, G. Storrøsten, M., and Carter, M., “How a Group Editor Changes the Character of A Design Meeting as Well as its Outcome.” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, pp. 91-98.

[75] Pacull, F., Sandoz, F., and Schiper, A., “Duplex: A Distributed Collaborative Editing Environment in Large Scale.” *Transcending Boundaries: Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, October 22-26, 1994. pp. 165-173.

[76] Patel, Dorab, and Kalter, Scott D., “A UNIX Toolkit for Distributed Synchronous Collaborative Applications.” *Computing Systems*, Berkeley, CA: University of California Press, 1993, pp. 105-134.

[77] Patterson, J. F., Hill, R. D., Rohall, S. L., and Meeks, W. S. “Rendezvous: An Architecture for Synchronous Multi-user Applications.” *CSCW 90: Proceedings of the Conference on Computer-Supported Cooperative Work*, Los Angeles, CA: ACM, 1990, 317-328.

[78] Poole, M.S., Holmes, M., and DeSanctis, G., “Conflict Management and Group Decision Support Systems.” In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (CSCW’88), Portland, OR: ACM. September 26-28, 1988, pp. 227-241.

[79] Prakash, Atul, and Shim, Hyong Sop, “DistView: Support for Building Efficient Collaborative Applications using Replicated Objects.” *Transcending Boundaries: Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, NC: ACM, October 22-26, 1994. pp. 153-164.

[80] Reder, Stephen, and Schwab, Robert G., “The Temporal Structure of Cooperative Activity.” In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’90), Los Angeles, CA: ACM, October 7-10, 1990, pp. 303-316.

[81] Reiss, Steven P., “Interacting with the FIELD Environment.” Software—Practice and Experience, vol. 20(S1), pp. 89-115. John Wiley & Sons, Ltd., June 1990.

[82] Reiss, Steven P., “Connecting Tools Using Message Passing in the Field Environment.” IEEE Software, pp. 57-67, July 1990.

[83] Ritter, Norbert, “An Infrastructure for Cooperative Applications Based on Conventional Database Transactions.” Workshop on Distributed Systems, Multimedia, and Infrastructure, ACM Conference on Computer-Supported Cooperative Work (CSCW’94), Chapel Hill, NC October 22, 1994.

[84] Rivest, R., “The MD5 Message DIgest Algorithm,” RFC 1321, April 1992.

[85] Rivest, R., Shamir, A., and Adleman, L., “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of the ACM*, v. 27, n. 4, April 1994, pp. 393-395.

[86] Rivest, R., Shamir, A., and Adleman, L. “On Digital Signatures and Public-Key Cryptosystems,” MIT Laboratory for Computer Science, Technical Report, MIT/LCS/TR-212, January, 1979.

[87] Root, R.W. “Design of a Multi-Media Vehicle for Social Browsing,” *CSCW 88: Proceedings of the Conference on Computer-Supported Cooperative Work*, Portland, OR: ACM, September 26-28, 1988. pp. 25-38.

[88] Roseman, M., and Greenberg, S. “GroupKit: A Groupware Toolkit.” Poster presented at *ACM SIGCHI Conference on Human Factors in Computing Systems*, Monterey, CA: ACM 1992.

[89] Roseman, M., and Greenberg, S. “GroupKit: A Groupware Toolkit for Building Real-Time Conferencing Applications,” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, 43-50.

[90] Roseman, Mark, and Greenberg, Saul, “GroupKit—A Groupware Toolkit for Building Real-Time Conferencing Applications.” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, 43-50.

[91] Rouncefield, M., Hughes, J.A., Rodden, T., and Viller, S., “Working with Constant Interruption: CSCW and the Small Office.” In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW’94).* Chapel Hill, NC: ACM, October 2-26, 1994. pp. 275-286.

[92] Saltzer, J.H., “Protection and the Control of Information Sharing in Multics.” *Communications of the ACM* 17(7), July 1974, pp. 388-402.

[93] Sarin, S., and Grief, I., “Computer-Based Real-Time Conferencing Systems.” *Computer-Supported Cooperative Work: A Book of Readings*, Irene Grief, ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 397-420.

[94] Scheifler, Robert W. and Gettys, James, “The X Window System.” *ACM Transactions on Graphics* 16:8 (August, 1983), pp. 57-69.

[95] Scheifler, Robert W., *X Window System Protocol Specification, Version 11*. Massachusetts Institute of Technology, Cambridge, Massachusetts, and Digital Equipment Corporation, Maynard, Massachusetts, 1987.

[96] Schneier, Bruce, *Applied Cryptography*. New York: John Wiley & Sons, 1994.

[97] Sellen, A., “Speech Patterns in Video-Mediated Speech.” In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI’92), Monterey, CA: ACM, May 3-7, 1992. pp. 49-59.

[98] Shen, H., and Dewan, P. “Access Control for Collaborative Environments,” *Sharing Perspectives: Proceedings of the Conference on Computer-Supported Cooperative Work*, CSCW’92, Toronto, Ontario: ACM, 51-58.

[99] Shrivastava, S.K., Dixon, G.N., Little, M.C., Parrington, G.D., Hedayati, F., and Wheater, S.M., “The Design and Implementation of Arjuna.” University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, No. 280, March 1989.

[100] Shrivastava, S.K., Dixon, G.N., Parrington, G.D., “An Overview of the Arjuna Distributed Programming System.” IEEE Software, January 1991, pp. 66-73.

[101] Smith, Ian, and Hudson, Scott, “Low Disturbance Audio For Awareness And Privacy In Media Space Applications,” *Proceedings of ACM Conference On Multimedia*, November, 1995, San Francisco, CA: ACM.

[102] Smith, Ian, and Hudson, Scott, “Applying Cryptographic Techniques to Problems in Media Space Security.” In *Proceedings of the ACM Conference on Organizational Computing Systems* (COOCS’95), Milpitas, CA: ACM, August, 1995. pp. 190-196.

[103] Sohlenkamp, Markus, and Chwelos, Greg, “Integrating Communication, Cooperation, and Awareness: The DIVA Virtual Office Environment.” *Transcending Boundaries: Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, October 22-26, 1994. pp. 331-343.

[104] Stefik, M.J., Bobrow, D.G., and Kahn, K.M. “Integrating Access-Oriented Programming into a Multiparadigm Environment.” *IEEE Software*, 3,1, IEEE Press, January, 1986, 10-18.

[105] Stefik, M., Bobrow, G., Foster, G., Lanning, S., and Tatar, D., “WYSIWIS Revised: Early Experiences with Multiuser Interfaces,” *ACM Transactions on Office Information Systems* 5:2 (April 1987), pp. 147-167.

[106] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., Suchman, L., “Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings.” *Computer-Supported Cooperative Work: A Book of Readings*, Irene Grief, ed. San Mateo, CA: Morgan Kaufmann, 1988, pp. 335-366.

[107] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., and Suchman, L. “Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings,” *Communications of the ACM* 30:1 (January 1987), pp. 32-47.

[108] Stults, Robert, *Experimental Uses of Video to Support Design Activities*, Xerox PARC Technical Report #SSL-89-19, December 1988.

[109] SunSoft, Inc., *The ToolTalk Service: An Inter-Operability Solution*. Des Moines, IA: Prentice-Hall, 1994.

[110] Tang, J.C., Isaacs, E.A., and Rua, M., “Supporting Distributed Groups with a Montage of Lightweight Interactions.” *Transcending Boundaries: Proceedings of the ACM Conference on Computer Supported Cooperative Work* (CSCW’94), Chapel Hill, October 22-26, 1994. pp. 23-34.

[111] Tolone, W.J., Kaplan, S.M., Fitzpatrick, G. “Specifying Dynamic Support for Collaborative Work within WORLDS.” *Proceedings of the ACM Conference on Organizational Computing Systems*, 1995.

[112] Van Rossum, Guido, *Python Reference Manual Release 1.3*. October 13, 1995 (available as http://www.python.org/doc/ref/ref.html).

[113] Want, R., Hopper, A., Falcao, V., and Gibbons, J., “The Active Badge Location System.” *ACM Transactions on Information Systems*, Vol. 10, No. 1, January 1992, pp. 91-102.

[114] Weber, Karon, and Poon, Alex, “Marquee: A Tool for Real-Time Video Logging.” *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI’94), Boston, MA: ACM April 24-28, 1994. pp. 58-64.

[115] Wexelblat, Alan. “Building Collaborative Interfaces.” Tutorial presented at *ACM SIGCHI Conference on Human Factors in Computing Systems*, New Orleans, LA: ACM 1991.

[116] Zimmerman, P., *PGP User’s Guide*, December 4, 1992.

Vita

Warren Keith Edwards was born on August 19, 1966 in Fredricksburg, Virginia. He attended high school at Red Bank High School in Chattanooga, Tennessee, where he grew up. He received his Bachelor of Science degree in Information and Computer Science at the Georgia Institute of Technology in 1989. He received his Masters of Science degree in Information and Computer Science, also at Georgia Tech, in 1991.

During his graduate work, Keith has served as a Research Assistant in the College of Computing, working on a number of projects related to multimedia systems, novel interface techniques, and computer-supported cooperative work. He has also worked as a teaching assistant, instructor, author, and consultant for a number of companies, including Lockheed, NeXT, BellSouth Information Systems, SecureWare, and Sales Technologies, Inc. Keith has also done internships at the Olivetti Research Center (Menlo Park, California), SunSoft, and Sun Microsystems Laboratories (both in Mountain View, California).