A PROGRAMMING FRAMEWORK

SUPPORTING THE RAPID APPLICATION DEVELOPMENT

OF HIGHLY-INTERACTIVE,

COLLABORATIVE APPLICATIONS

by

Conan Carl Albrecht

A Dissertation Submitted to the Faculty of the

COMMITTEE ON BUSINESS ADMINISTRATION

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY
WITH A MAJOR IN MANAGEMENT

In the Graduate College

THE UNIVERSITY OF ARIZONA

2000

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Conan Carl Albrecht

entitled A Programming Framework Supporting the Rapid Application Development of Highly-Interactive, Collaborative Applications

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Jay F. Nunamaker	10-13-00 Date
James D. Lee	13 Oct 2000 Date
Olivia R. Liu Sheng	10-13-00 Date
Kris Bosworth	10/13/00 Date
	Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Jay F. Nunamaker — 10-13-00 Date

STATEMENT BY THE AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under the rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: 

ACKNOWLEDGMENTS

I first thank my children, Brynn and Lyndsie. These beautiful girls have been so wonderful to come home to after long hours at school. Seeing them looking out the window as I drive up each night has lifted my spirits and increased my enthusiasm. They and children around the world are a constant reminder of why we must strive with faith for excellence in all we do.

I thank Duffy Gillman, friend and fellow researcher. The Property hierarchy was his idea and was the spark for the entire Collaborative Server. Duffy has a talent for seeing "outside the box" and contributing unique ideas to projects.

I thank my mentors and committee members, Dr. Jay F. Nunamaker, Dr. Jim Lee, Dr. Doug Dean, Dr. Olivia Sheng, Dr. Mary McCaslin, and Dr. Kris Bosworth for their example and for teaching of what research really is. They have shown me what it means to search for knowledge. I owe much of what I have learned in my doctoral studies to these people. They have been instrumental in my progress through the program.

I thank the Air Force Office of Technology and the Department of Defense Environmental Security Corporate Information Management Office for their generous support of the development of the Collaborative Server.

Finally, I thank my parents for their love and support throughout my life. They provided a stable home for me to grow up in, and they practiced what they preached. There were no double-standards in my home. Especially, my Mother deserves a special thanks because she places her children above all else, including her own interests and concerns. She is a model of love, devotion, and commitment. I owe much of what I am to my Mom and Dad. I also thank my parents-in-law for their support of Laurel throughout her life.

DEDICATION

I dedicate this dissertation to two individuals who have profoundly impacted my studies toward a Ph. D. The first is my wife, Laurel Albrecht. She has been a supporter of my doctoral program since the beginning–at significant cost to herself. She has spent many days and nights as a "single parent" with two children at home while I studied for classes, worked on research, and wrote my dissertation. She has always encouraged me to give my best to this research. She is my best friend, a wonderful mother, and an excellent example to me and all around her.

The second person to whom this dissertation is dedicated is Dr. W. Steve Albrecht, my father. He is a model of professional success in research, teaching, and service. During his career, he has made significant contributions to the knowledge base of his field. He is an outstanding teacher and has served in numerous service positions, including President of several national accounting- and fraud-related organizations such as the American Accounting Association and the Association for Certified Fraud Examiners. His example prompted my entry into doctoral school, and his constant encouragement provided needed support during the program.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

Business team collaboration in the future will increasingly take place in an environment where everything is potentially distributed. Traditional group support systems have been LAN-based and have not ported easily to distributed, Internet-based products. Rather, distributed group support systems require new architectures if they are to support the same type of real-time, high-collaboration environments that traditional systems have.

This dissertation describes the development of a server architecture and programming model to support the rapid application development of real-time, distributed, and collaborative applications. Since the efficiency, robustness, and scalability of these applications are handled by the server, complex applications can be developed in very short production cycles.

Background literature presented in the dissertation includes a taxonomy of distributed networking and data sharing methods, models of dialog independence, and messaging methods. Many different prototype and real systems are presented, analyzed, and compared with the Collaborative Server in this dissertation. The Systems Development methodology is applied to this research domain.

A prototype system–the "Collaborative Server"–is developed, including a unique data persistence and replication scheme based upon a central Property object. Properties are created in hierarchical trees to form the "scaffolding" for collaborative applications. Each Property governs a single value, or datum, and controls the access, replication, security, viewing, locking, and persistence of its value. Properties are layered with predefined or custom child indices.

A new systems development methodology, based upon the Waterfall method but tailored for the Collaborative Server, is presented. The efficiency and effectiveness of this methodology is analyzed using a simulated example and two case studies.

The research contribution of this dissertation is the Collaborative Server. The basis of this server is the Property concept, including the methodology changes it effects, the efficiencies and application stability it produces, and the specialized messaging system it allows. Required programmer knowledge is significantly reduced when applications are built upon the Collaborative Server. In addition, the presented examples and case studies show significant decreases in code size and in time-to-market for their respective applications.

CHAPTER 1: INTRODUCTION

Group support systems (GSS) research has traditionally dealt with the development of application tools that make group collaboration more efficient and effective. The research focus has often been centered on user interfaces, meeting semantics and processes, and effective facilitation methods. Much knowledge has been gained about the types of group processes, productivity requirements and expectations, group communication, and benefits. For example, Nunamaker, Briggs, and Mittleman (1990) summarized the benefits learned from GSS research as follows:

- Shorter project elapsed times of 90%.
- Reduced labor costs of 50-70%.
- Improved decision quality.
- Improved buy-in to decisions.

Further, many different and useful models of collaboration have been created. For example, see Nunamaker, et al. (1991) for process gains and losses; Ellis, Gibbs, & Rein (1991) for a GSS taxonomy and typing model; and Dennis, et. al. (1988) for a model of environment, methods, and group processes and outcomes. Research has also traditionally

dealt with the design of user interfaces for effective team collaboration. For example, Nunamaker, Briggs, and Mittleman (1990) cite significant productivity effects that even small user interface tweaks create.

The context of this dissertation is different than these traditional areas of GSS research. It considers the technology involved in making online collaboration possible. At its foundation, a GSS application must distribute shared data to client computers. This dissertation proposes a new method and technology for collaborative data sharing. It lies one layer beneath traditional GSS research. The motivation and need for this technology are addressed in the following paragraphs.

GSS applications have traditionally been based in LAN architectures, an area that has been well developed for several decades. In addition, mature integrated development environments such as Borland C++, Microsoft Visual Studio, and other platforms assisted programmers in quickly creating robust applications within LAN environments. However, the growing popularity of the Internet during the 1990s has shifted GSS research from centralized, face-to-face meetings to distributed, asynchronous meetings. In response, several products have been built to support distributed teams (see Chapter 2). Despite these applications, no single platform has emerged as a best-practices method for creating collaborative applications. Rather, Internet-based groupware has been riddled with time delays, buggy behavior, and data update anomalies.

Several reasons may be cited for the delays in porting GSS software to distributed, Internet-based applications, including differing assumptions, team dynamics, and the new

complexities of distributed, network programming. This dissertation focuses on the last reason: the complexities of developing Internetworked, multi-user software. Several of these complexities are presented in the following list:

- Network programming models (such as RMI and CORBA) present complex application programming interfaces (APIs) and new problems not found in LAN environments. For example, firewalls are ever present in the Internet business world and must be worked around for applications to function correctly.

- Traditional GSS environments are normally homogeneous in their hardware, operating system, and software, allowing programmers high control over protocols, installation procedures, and application behavior. In contrast, the Internet is by definition a heterogeneous network of smaller, disparate networks. Programmers have very little control over the environments within which their applications run.

- Data replication and consistency issues require significant programming. These issues, normally handled by relational database systems, cannot be easily solved with off-the-shelf software. Further, packet and message

ordering and timing cannot always be predicted with computers connected over public networks such as the Internet.

- Real-time applications generally require high bandwidth. While bandwidth is not an issue in local applications, distributed clients are often connected over 14,400 baud modems which do not respond well to high concurrency. GSS applications have traditionally been immediate in their shared-view updates. This functionality, while possible with distributed clients, requires complex programming to achieve robust and scalable solutions.

- GSS applications share many common behaviors, such as shared views, data replication, access control, etc. Network application programmers often "reinvent the wheel" in solving these problems, rather than building upon a standard, tested set of solutions.

Despite the complexities involved in distributed, real-time programs, product release cycles have become increasingly shorter. A product's success is often attributed to its ability to beat competitors to market rather than to its technical advances. Programmers are under increasingly difficult time constraints to develop complex, robust, and scalable solutions that can be deployed quickly.

The need for short release times is very apparent within The Center for the Management of Information (CMI), the organization within which the server was created. As a research organization, CMI designs and produces prototypes that meet needs that have not yet been addressed by commercial products. Prototype development is notorious for quick cycles and rapid changes. At the "Internet speed" that companies are now moving, commercial development projects are also becoming more and more notorious for the quick cycles and rapid design changes that traditionally existed only in prototypes.

These two issues–the complexity of distributed, collaborative software and quick product release cycles–prompt the need for a rapid application development (RAD) framework for real-time, collaborative software design and code.

1.1. Project Goals

Given these problems and complexities, several goals were outlined for a new architecture and programming model. The CMI Collaborative Server, the end product of this effort, included the following goals which were outlined as the foundation of this research:

1.1.1. Abstracted Collaborative Behavior

The server should contain common collaborative behavior found in most applications. Examples of these functions are security, replication, etc. These behaviors should be abstracted into a framework and then coded into the server directly. Applications should be able to access these functions with standard, interface-driven, remote calls.

1.1.2. Rapid Application Development

Programmers should be capable of programming complex applications very quickly within the framework. Since the server provides collaborative behavior and Java's Swing provides standard user interface components, this goal is attainable. In addition, the server API should be simple so programmers can quickly harness the power of the framework.

1.1.3. Dynamic Data Structures

The server must be based in dynamic data structures that allow it to be used in a wide range of application spaces. For example, if a relational database is used for data persistence, the schema must not define tables and fields that are specific to any product area. Rather, the schema must be flexible enough to handle graphics, text, sound and video bytes, and other data types in various forms of organization (lists, trees, and even more complex graphs).

1.1.4. Thin, Common Client

The initial cost of Java development is the need for the Java Virtual Machine (JVM), which is currently a 5 megabyte download. The JVM is the engine that allows common Java code to run on many different platforms and is packaged for applet use as a browser plug-in. Since full-featured client applications are required, there is no way around the dependence upon the JVM. However, all code beyond the JVM should be thin, light, and small. This allows downloaded applets to start very quickly, even on slower modem connections. As a rule of thumb, the client download should be less than 150K.

1.1.4.1. A Common Client

A common client must be built in Java that all applications run within. The common client should care of the initial bootstrap process, which includes connecting to the server, establishing event queuing, logging in and sessioning, and application invocation. The client should provide a standard frame that application panels are placed into.

A further advantage of the common client is increased potential for caching. Since most browsers cache files downloaded from the Internet, once a user has downloaded the common client, he or she should be able to run all framework-based applications without needing to download the common client again.

1.1.4.2. Multiple Client Environments

Since applications must be written to an interface that allows them to be placed into a common client, multiple client environments can be programmed. At least three environments should be provided:

- *Applet-based*: This environment will probably be the most used since it can be hosted by a simple web browser. The applet should perform the special functions needed to allow applets to download code and communicate via RMI from within Java's security sandbox.

- *Application/Executable-based*: This environment allows for a local installation of framework-based applications. Users should be able to run

applications and connect to remote servers around the world. Since Java applications and applets are based in the same set of components, code should run seamlessly within both of these first two environments–*with no changes to the application code*.

- *HTML/Servlet-based*: This environment uses pure HTML pages and forms via the common gateway interface (CGI) of web browsers. Clients should not need a local JVM and should be able to use any browser of their choice. Since HTTP, the web protocol, is a request/response architecture, web-based applications will not be as real-time as in the first two environments. Browsers should request updates by polling the server periodically. However, for applications that are not as real-time intensive (such as administration dialogs), servlets provide a great client interface. In addition, servlets will most likely be used for printing, saving, and uploading since browsers can access the local machine resources, while applets cannot.

1.1.4.3. User-Interface Driven

Since the server provides access to data as well as common collaborative behaviors, client programmers should be able to focus almost solely on user interfaces. It is expected that after the common client download, most other classes downloaded will be user interface classes. These classes should be downloaded upon demand by the clients; if a user does not

open certain screens or dialogs of the application, these class definitions should not be downloaded.

In addition, all client programs should be based upon Java's Swing architecture. Swing provides standard UI components such as lists, text fields, tables, and graphics. Since these classes are included in the JVM package, they are already on client machines and do not need to be downloaded again.

1.1.5. Distributed

The new server should fully support distributed applications. In addition, the data should be kept on the server or set of servers and not on client machines. The applications should be accessible from any client computer on the Internet, providing they have firewall and security access to the applications.

1.1.6. Real-Time

The applications built upon the framework should be multi-user applications. They should be real-time in their ability to replicate data very quickly to all clients connected to a server. Therefore, when one client modifies data on the server, all other clients accessing that same data should immediately see the changes on their screens. While data is real-time, users should be in control of their applications. A modern event system should provide the replication to clients; this event system should be based upon industry-standard messaging theory.

1.1.6.1. Locking

Multiple users should be able to access the same data at the same time in a robust, consistent way. Therefore, a locking system should be provided by the server and should be automatic at all possible locations.

1.1.6.2. Security

The server should explicitly manage access to data and applications. Clients should not make decisions on security–decisions must be made at the server level. Having the server manage security ensures that rogue client programs cannot undermine security.

Clients should be assigned session tokens containing their rights that are good for a period of time. All server access is carried out through these tokens, and the server may refuse access based upon token rights. Sessioning also prevents usernames and passwords from crossing the network unnecessarily.

1.1.7. Portable

The server should be programmed in Java so it can run on a wide variety of operating systems. In addition, no proprietary ties to any third-party code should be made. Custom systems should be written where standard Java interfaces do not provide. For example, the inclusion of custom security and messaging systems ensures the framework is portable to different platforms.

The server must be the only portal to the data, thereby allowing different data storage routines (relational databases, other databases, flat-file schemes, etc.) to be plugged into the

server with no change in client code. Application programmers should see the data only as a set of objects.

In addition, client-side code should be written towards the Java plug-in from Sun. The Microsoft-Netscape battle that persisted in the 1990's resulted in different virtual machine implementations. The Java plug-in runs as a regular plug-in (similar to Shockwave or RealPlayer), overriding any proprietary JVM(s) included with the browser.

1.1.8. Efficient

While the server is being developed as a prototype, high efficiency is required to ensure scalability and quick replication. These efficiencies should not affect the overall server or client interface, but should integrate seamlessly whether or not the efficiency-based code is enabled.

In addition, replication should only take place between clients viewing the same data. Data updates for parts of the application that are not currently visible (other than for caching) are unnecessary. Therefore, the server must know the location of users and what data they are interested in. These lists should be programmed in an efficient manner and should not unnecessarily spend server processor time.

All processing that can be done by client machines must be done there, with the server managing only core collaborative functions.

1.1.9. Scalable

The server should initially support small groups of clients (1-50 machines). It is hoped that within a short time the server code can be moved to a production-scale server to serve several thousand clients at a time on a single machine.

The server code should be written to allow for n-tiered server architectures. Client machines should view their server as the only worldwide server for their data. However, the server might in actuality be one in a large set of n-tiered servers, either in hierarchical or workgroup fashion.

1.1.10. Robust

At their most basic level, collaborative applications share data. In the case of this server, data should be kept on the server machine(s). Clients have little control over the integrity of their data. Therefore, it is imperative that server data are *never* lost. Data integrity includes the following issues:

- *Consistency*: Different clients should always have a current view of the data. One user should not be able to overwrite the changes of another without passing security and locking rules first. These rules should be enforced automatically by the server.

- *Storage*: The back-end data storage schemes used by the server should be robust. Regular backup and integrity-verification programs should be included to provide failover security.

1.2. Research Questions

The goals described above can be summarized into the following two research questions:

1. Can a generalized server abstract common collaborative behavior without becoming application-specific? If so, what are the required features of such a server?

   This question prompts the use of the Software Development methodology: *build a system to prove the concept.* An instantiated system provides a real-world answer to questions such as the one posed. Running systems and source code are concrete and are more detailed than English-language descriptions. In addition, prototypes are required by many engineering fields because they promote intellectual honesty.

2. Is the software development process using this framework faster than the software development process using traditional methods?

> The foundation and prompting goal of this research was to increase the speed of creating distributed, collaborative applications. This question asks whether this goal has been met: Is a software development process that uses the Collaborative Server faster/better/cheaper than one that uses traditional methods?

Following Chapter 2's literature review and Chapter 3's methodology description, research question one is addressed in Chapter 4 by describing a new server designed to meet the goals described earlier in this Introduction. Chapter 5 uses a simulated example and two case studies to address question two and evaluate the proposed server. Chapter 6 concludes the argument with discussions of contributions, limitations, and future research.

CHAPTER 2: LITERATURE REVIEW

The foundation research for this dissertation involves models and schemes that support the sharing of data with many clients. This chapter presents previous research in the area of data sharing, including data sharing methods, models of dialog independence, and messaging.

2.1. Data Sharing Methods

Several methods have been proposed in the literature to allow the sharing of data, and most of these models include reference implementations. This chapter presents a taxonomy of previous research published in Management Information Systems, Computer Science, and other journals. It also details the systems created in relation to these models. Several taxonomies and groupings for data sharing are given in the literature (Dewan, Choudhary, & Shen, 1999; Dewan & Beaudouin-Lafon, 1998; Greenberg, Roseman, & Beaudouin-Lafon, 1996; Menges & Kevin, 1994; Bentley, et. al., 1994; Stefik, et. al., 1987). These are summarized in the following taxonomy:

Method	Comments
Centralized	Uses a single data source. Centralized systems are easier to implement and keep consistent, but they prove challenging in providing different types of sharing, rapid feedback, and scalability.
Replicated	Replicates the data across many clients or servers in distributed fashion. Replicated systems excel at different levels of sharing and rapid feedback, but they are harder to implement, can be network intensive, and present consistency and data clashing challenges.
Hybrid	Uses a single data source, but replicates the presentation and interaction semantics. Hybrid systems use elements of both centralized and replicated models in order to best meet specific domain needs.

TABLE 2.1, Taxonomy of Data Sharing Methods

2.1.1. Centralized

Centralized systems are the easiest to implement and are the simplest, both conceptually and during implementation (Greenberg, Roseman, & Beaudouin-Lafon, 1996). A centralized model maintains one server that all clients retrieve data from. Since the data are kept on a single, centralized server, data replication is automatic and locking is trivial.

Centralized models are sometimes so common that many take them for granted. These models include operating systems, shared databases, polling, and many others.

2.1.1.1. Multi-User Operating Systems

Multi-user operating systems (OS), such as VMS and Unix (Ritchie & Thompson, 1978; Bell Labs, 2000), provide inherent support for shared data. In these OSs, the filesystem is by default a shared data store. The OS provides support for locking and security. Telnet and Remote X allow clients to view and even share system data from remote locations.

Single-user OSs, such as Windows and DOS, provide data sharing capabilities for programs running in parallel. For example, DOS' SHARE.EXE program manages locks on system files and data.

Many "green screen", multi-user applications have been written with multi-user operating systems as the base mechanism for data sharing. For example, many readers are familiar with 3270 Terminal access, which is a common method of accessing corporate data on mainframes.

2.1.1.2. Shared Database

The advent of database systems, particularly relational database systems (Codd, 1970), provided a more automatic method of collaborative data management. Most database management systems (DBMS) provide multi-user support, two-phase locking, security, and other mechanisms common to collaborative systems. These features are even found in many desktop DBMS's such as Microsoft Access and Xbase (Microsoft, 2000a) systems. Since all client applications connect to the same data source, changes to the data are automatically reflected throughout the system.

This method of collaboration has proved very effective for many applications. Shared databases are used in most corporations today for organization-wide data access. However, the actual update mechanism involved is dependent upon the database the application is based upon. There is no way to ensure that the system uses direct connections, polling, or other mechanisms.

Databases manage large data sets and expect most users to work in different areas of the table structures. They generally provide collaboration support at the record level. In fact, they excel at locking others out of records while one user modifies data. This dissertation deals with applications with smaller data sets and significantly increased collaboration needs than databases typically provide; users are expected to work on the same areas of data at the same time. Therefore, shared databases are not addressed further in this dissertation.

2.1.1.3. Polling

Polling shifts update responsibilities to clients. Client machines query the server at specified intervals for data updates. This method does not result in "true" real-time collaboration since data are only refreshed at periodic rates. However, with sufficiently short refresh periods, real-time collaboration can be approximated. For example, GroupSystems suite of applications (GroupSystems.com, 2000) uses a refresh period of 3 seconds. HTTP, the protocol of the World Wide Web, is a request/response protocol (W3C, 2000; Johnson, 1996), which requires a polling scheme since only clients can initiate data transfer. While methods to get around this limitation exist, HTTP remains based in polling schemes.

Polling has several disadvantages. First, since clients must initiate data refresh, applications cannot ensure the entire client network is up-to-date. This may result in data inconsistency and update anomalies. Second, clients poll at specified periods whether or not the data on the server have been updated. Therefore, this method usually results in unnecessary bandwidth use and server processing. While this may not be an issue on local area networks, distributed collaborative applications suffer significantly from these inefficiencies.

2.1.1.4. MMM

The Multi-Device Multi-User Multi-Editor (Bier & Freeman, 1991) project at Cambridge University created a multi-user editor that could be shared across time and space. Concurrent users shared a single screen, with each user having his or her own modes, style settings, insertion points, and feedback. However, this research limited itself to shared editors on a *single* workstation with multiple input devices. It was successful in this small space, but it ignored many of the potential problems that occur when multiple workstations are networked. Therefore, by default this system is a centralized system with no distribution.

2.1.1.5. CVW

The Collaborative Virtual Workspace is a prototype for a collaborative team computing environment at Mitre Corporation (MITRE, 1994a; MITRE, 1994b; MITRE, 1994c). It is a framework including whiteboards, shared documents, chat rooms, and other applications. While it includes several different technologies, it is based on a centralized

scheme with root-level objects. Extending the core database involves defining a new object on "#0", the system object, which resides in a central location.

2.1.1.6. Promondia

Promondia (Gall & Hauck, 1999) is a Java-based framework for collaborative applications. Most information is passed through a server, which includes an HTTP server that client applets are downloaded from. Promondia uses a tuple space model for data persistence, and a Model-View-Controller method of client interface. It provides support for sessions, users, and administrators.

2.1.2. Replicated

Replicated architectures stand at the opposite extreme from centralized models. In these systems, data are fully distributed to all clients and there is no central server. These systems are more complex to design because issues such as locking, security, and persistence must also be distributed (Dewan & Beaudouin-Lafon, 1998; Greenberg, Roseman, & Beaudouin-Lafon, 1996). For example, in a replicated system, a program must determine where a desired piece of datum resides before it can replicate it to the local screen.

Replicated schemes usually offer higher scalability than centralized one because no one server must control the collaborative behavior. Rather, many machines are allowed to do the work of a single, centralized server.

2.1.2.1. Gnutella

Very few systems are as extremely replicated as Gnutella. However, Gnutella is not a highly collaborative system (i.e. many clients do not need to see the same piece of datum

at the same time). Rather, clients share copies of data and no mechanism exists to update those copies when changes are made.

The Gnutella web site describes it as follows (Gnutella, 1999):

> Gnutella is a fully-distributed information-sharing technology. Loosely translated, it is what puts the power of information-sharing back into your hands . . . Gnutella client software is basically a mini search engine and file serving system in one. When you search for something on the Gnutella Network, that search is transmitted to everyone in your Gnutella Network "horizon". If anyone had anything matching your search, he'll tell you.

While Gnutella does not strictly compare with highly interactive, multi-user applications, it serves as a very good example a truly distributed system. Gnutella is also a very good example of the complexities of distributed systems; Clip2 (2000) reports that Gnutella's scalability limit has already been reached due to the number of low-bandwidth nodes on its network.

2.1.2.2. GroupKit

GroupKit (Roseman & Greenberg, 1999c; Roseman & Greenberg, 1995) is an extension to Tcl/Tk (a user interface component set) and makes these components collaborative. It uses built-in socket commands for its low-level networking. It supports the idea of users and sessions. A registrar on each client machine is used to connect users with

each other. Clients typically designate one workstation as the "well-known" registrar, and other workstations' registrars stay unused. GroupKit is distributed in that each client maintains connections to other clients in the meeting once the registrar has connected them. All further communication in the meeting is done client-to-client.

GroupKit components allow a single-user application to quickly morph into a multi-user application (Roseman & Greenberg, 1999b). However, it does not address some inherent problems of collaborative software, such as concurrency control and security.

2.1.2.3. Sync

Sync is a Java-based framework for mobile collaboration at the University of North Carolina (Munson & Dewan, 1997). Applications are divided into client-side and server-side implementations. Each implementation contains replicated data which is synchronized with all other clients. Data merging and conflicts are handled in batch, during periodic synchronization. Since no central server is used, Sync is a fully replicated architecture.

2.1.2.4. Collaboration Bus

The Collaboration Bus (Marsic, 1999a) is a research framework developed at Rutgers University. It provides a component-based, plug-and-play environment. Single-user application components can be swapped with Collaboration Bus components to easily make an application collaborative.

Under this architecture, each client runs a local copy of the program and communicates data changes to other clients in a replicated manner. A central server keeps

track of sessions (ongoing conferences) and a change history. However, all other functions are done client-to-client.

2.1.2.5. Shared Events

Shared Events provide a mechanism to quickly turn an event-based system into a *multi-user*, event-based system. In this scheme, events that are normally relayed to only one client are copied and passed to all clients. Therefore, all clients receive the same data changes and stay in sync with each other. In addition, if events are also shared, shared events allow a facilitator-type client to control the user interfaces of all other clients.

A system that uses shared events is the NCSA's Habanero environment. "The Habanero framework or API is designed to give developers the tools they need to create collaborative Java applications. The framework provides the necessary methods that make it possible to create or transition existing applications and applets into collaborative applications. Using the Habanero Wizard, developers can easily convert applets by selecting the objects and events they want to share. The Wizard then rewrites the code to take advantage of the Habanero API." (Chabert, et. al. 2000)

Shared event systems excel at speed of development since only user-interface components need to be changed. However, these systems are normally bandwidth-intensive since local event models are not designed for distributed use. Messaging systems specifically optimized for distributed applications are usually more efficient.

2.1.2.6. NetMeeting

NetMeeting (Microsoft, 2000a) is a communication system and shared whiteboard on the Internet. It uses central servers only to connect clients. Participants then communicate directly with each other for the duration of the session. NetMeeting gives very little support for many features that are required in collaborative applications, such as persistence, locking, security, etc. However, it is a good example of a replicated scheme.

2.1.2.7. System V

System V is a distributed, multi-user OS developed at Stanford University (Lantz, Nowicki, & Theimer, 1985; Lantz & Nowicki, 1984). While it is an OS rather than a single application, its testing included a shared graphics application that is similar to the design space of this dissertation. System V is fully distributed; each workstation is treated as a multifunction component of the OS.

Each node in System V controls the data and communication required for its function. Therefore, clients wishing to access the system must communicate separately with each functional node they need to use. A common communication protocol, Interkernal Protocol, allows clients and system nodes to send synchronous messages in a standard way.

2.1.3. Hybrid

Hybrid architectures include aspects of both centralized and replicated systems in an effort to gain advantages of both models. Hybrid architectures are often very different from one another because the designers of these systems focus efficiency gains and tradeoffs based upon product needs.

2.1.3.1. Socket-based proprietary

A direct connection is the most basic method of communication. It typically streams bytes socket to socket or utilizes Remote Procedure Call (RPC) to directly call methods on a target machine. Sockets are included in the Hybrid section because they normally take this form. However, they can actually be used in any type of architecture.

Direct connections (Stevens, 1997) do not use events but send more basic data structures such as strings or integers. In the instance of byte streams, the source and target machines open a socket connection and send bytes to each other. Case or switch statements on each end parse control characters from the byte streams to determine the specific data being sent. Direct connection messaging is synchronous in nature; it requires both the source and target machines to be connected in real-time.

RPC was first used at Xerox PARC in the early 1980's (Nelson, 1981; Xerox, 1981). RPC allows source machines to call procedures or methods on target machines. While this is more abstracted than byte streams, the end result is much the same: target procedures are called with basic data structures. RPC has been a very popular method of communication for several decades in many different application spaces.

2.1.3.2. Egret

Egret (Johnson, 1995a; Johnson, 1995b; Johnson, 1993; Johnson, et. al., 1993; Johnson, 1992) is a client-server framework that supports offloading of server responsibilities to separate machines. Therefore, it is a modified-centralized model. Node instances are the basic repository of information in the Egret architecture. Each node

instance holds an arbitrary number of fields, which represent the data of the application. Fields can also link to other nodes in the data store. Nodes and fields are defined by schema, as in relational schemas. However, Egret entities are allowed to diverge from the schema structure once they are created.

Egret is more than just a collaborative architecture. It is based in the defined group process of consent, exploration, and consolidation (Johnson & Moore, 1994). This can be seen as both a strength and weakness: close ties to a collaborative process equate to better support for that process but also hinder the framework's use in other domains.

2.1.3.3. Zipper

The Zipper architecture (Dewan, 1999e; Dewan & Sharma, 1999) is a good example of a hybrid architecture. It allows a program design to be as centralized or as replicated as needed for the problem space. Zipper assumes a hierarchical layout of servers with a centralized scheme at the top layer. At some point in the layering of one-to-one servers, a replicated scheme is begun and continues through all lower layers.

The Zipper architecture has notions of concurrency (called floor control) and coupling (how replicated client screens are). Applications based upon it are very comparable to those created with the framework in this dissertation.

2.1.3.4. Publish/Subscribe

This scheme is a more advanced version of shared events (TIBCO, 1999). It requires that messages are passed within event objects, which usually subclass a common super-object. The event system contains routing information that pass objects from event

queues to interested targets. The source usually knows nothing about the targets – it simply knows that it posts events to a specified event queue. Objects interested in events posted to a queue register themselves as listeners to that queue. When events are posted to the queue, copies are passed to each listening object. The listeners are then responsible for unwrapping the event object and dealing with the information as they see fit.

Message queues allow for asynchronous communication, a significant advance over synchronous RPC. Source applications simply post events and continue execution with little delay. The event queues are then responsible for the method, timing, and route of deliver to listening objects.

While the publish/subscribe methodology has existed for many years, it has recently gained increased popularity. This is in part due to the adoption of message queuing by the Object Management Group (OMG, 2000), Microsoft, the Java (Sun Microsystems, 2000a) community, and many others. Most current middleware products support some type of publish/subscribe scenario. Because of their popularity, messaging products are described in detail in a later section of this literature review.

2.1.3.5. Suite

Suite (Dewan, 1999b; Dewan 1999c) is an RPC-based framework that provides basic collaborative functions. It is based upon work done with Dost (Dewan & Solomon, 1990) at the University of Wisconsin-Madison. It offers the notion of active values, or pieces of datum that are made collaborative and require updates, locking, etc. (Dewan & Shen, 1999)

It also includes a dialogue manager, which acts as a middleman between the server data and client view.

Suite first demonstrated how a semi-replicated implementation of abstractions and views can be used for hybrid architectures (Dewan & Choudhary, 1999b).

2.1.3.6. DISCIPLE

DISCIPLE, developed at Rutgers University (Wang & Marsic, 1999; Marsic, 1999c), is a collaboration-enabling framework for Java. It supports both collaboration aware client beans (CABs) and collaboration unaware beans. CABs can take advantage of collaboration functionality such as concurrency control and coupling (Li, Wang, & Marsic, 1999).

DISCIPLE supports sessions through Place Servers, which maintain the state of the virtual workspace as well as the location of each user. Communication is divided into channels. The announcer channel is used within the nodes to communicate node-level information. Clients register as "talkers" on the poster channel and "listeners" on the merger channel. Events are posted and received accordingly (Dorohonceanu & Marsic, 1999).

Since DISCIPLE currently uses the iBus messaging middleware, it inherits its advantages and disadvantages. All communication going over a central server may limit bandwidth and scalability. However, the use of a commercial event bus adds to its robustness and maturity.

DISCIPLE uses a common communication center as the basis for applications. It includes a common tree with a Place Server as the root node. The tree progresses similar to other directories such as Novell's NDS (Novell, 2000) or Microsoft's Active Directory

(Microsoft, 2000b), giving organization, place, and user lists. Clicking on a node displays information about the user, including e-mail links, etc. Applications are created by customizing the components in this interface.

2.1.3.7. Rendezvous

Rendezvous (Patterson, 1991) is a collaborative framework similar to Suite. It provides the notion of "Shared Abstraction", or a collaborative piece of datum, and a View, or client program. Rendezvous's Abstraction-Link-View (ALV) architecture is described in detail in the next section entitled "Dialog Independence".

Rendezvous allows the programmer to determine whether it is fully centralized, hybrid, or fully replicated (Hill, et. al., 1994). The centralized model houses the entire ALV structure on a single Unix server, with clients seeing the views via multiple X servers. The replicated model creates several duplicate abstractions which replicate with each other at periodic times. The hybrid, seen as the most commonly-used form, uses a single abstraction on one machine with multiple views on multiple client machines. This hybrid architecture is similar to the Property model proposed in this dissertation.

2.1.3.8. Clock

A significant amount of work has been done on the Clock architecture, making it very robust and mature in features (Urnes & Graham, 1999; Graham, 1999; Graham, Morton, & Urnes, 1999; Graham, 1997; Graham, Urnes, & Nejabi, 1996; Nejabi, 1995; Graham 1994). Clock defines its own declarative programming language for developing multi-user

applications and provides a visual programming environment as a development aid. However, because it uses a custom language, programmers may feel constrained.

Clock is included in the Hybrid section because it allows programs to be developed with centralized to replicated architectures. Most applications are created using the Model-View-Controller method (described later in this chapter). Clock excels at event management and automatically ensures all client screens are updated.

2.1.3.9. Lotus Notes

Lotus Notes (IBM, 1999b) sees data as document databases. Programmers define creation and view templates which dictate the user interface for these databases. The back-end Notes engine seamlessly replicates changes on one database server to other database servers, with automatic conflict resolution. In this way, Notes is a hybrid model allowing changes that are made and reflected on one server to be cascaded across the entire network.

Notes is somewhat different than the collaborative application space of this dissertation because it allows conflicting changes to be made to separate servers without inter-server communication. Conflict resolution is done during periodic replication.

2.2. Dialog Independence

Dialog independence is the notion that data and user interface are two separate entities (i.e. dialogs are independent from data). Although many different models for dialog independence exist (Dewan, 1993; Dewan & Choudhary, 1999a; Bowers & Rodden, 1993; Dewan, 1991; Young, Taylor, & Troup, 1988), most follow a common pattern. Application

data are stored in a central location, whether it be a database, file server, or object store. These data pieces normally exist atomically and are replicated only for efficiency or robustness. Client's view data as if they were a single copy. Interaction objects, or views of the central data, are created on the client side when a client connects to the data. Each client owns his or her own instance of the interaction objects. These client-side objects control all communication between users and central data, including update, summarization, protocol, and user-interface.

If defined in abstract ways (meeting some defined and common interface), entities can be plugged together in many different ways to achieve enhanced functionality. In effect, client-side interaction objects can be connected to different server-side data to provide customized and different views for each user or application screen. This "pluggability" is one of the foremost advantages of dialog independence.

Figure 2.1 describes a common model of dialog independence. The diagram shows a single data source being viewed in three different ways and by four different users. The different objects depict different views of the data source. For example, User 1 and User 2 are viewing the data with the same type of interaction object. User 3 and User 4 are viewing the data with different types of interaction object. While each user's view

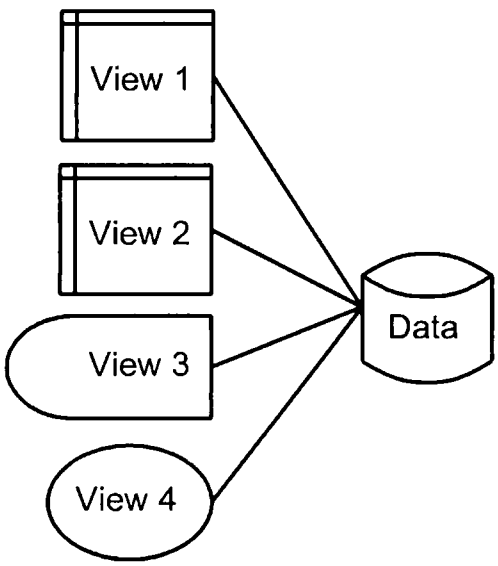

FIGURE 2.1, Dialog Independence

into the data might be different, the underlying data source is the same. Thus, each client interaction object controls how the data are viewed and interacted with.

2.2.1. Model-View-Controller

Model-View-Controller (MVC) was first conceived at Xerox PARC in the late 1970's (Goldberg, 1990; Goldberg, 1983; Goldberg & Robson, 1983) as part of the Smalltalk-80 programming language. It has become an increasingly popular metaphor for dialog independence since the middle 1990's. Part of its popularity may be due to its adoption by the Java language architects, who based the language's Swing foundation classes on MVC.

MVC distributes itself into three elements: Model, View, and Controller. *The Model* represents the data structure of the application. The model can be as simple as an integer or character, but is usually a more complex data object. In some instances, the model is even a back-end, relational database.

The *View* represent the graphical part of MVC. Views are composed of GUI elements such as list boxes, trees, buttons, text fields, and other components. Views interact directly with the user, showing the model data in different formats and on different medium.

The *Controller* interfaces between a View and a Model. It normally receives user input and contains behavior to act accordingly. Controllers track mouse movement, keyboard activity, and other user input devices. In many MVC implementations, such as Java's Swing foundation classes, the view and controller are combined into a single class. However, the distinction between the two elements remains clear.

Models often have many view/controller pairs associated with them. Messaging is used to pass update events from model to view, controller to model, and controller to view. Messaging keeps all views connected to a single model in sync with one another. For example, consider the situation where two users, A and B, are viewing a given model that represents a user name. If User A changes the name, the controller will send a change event to the model. The model will change the underlying data structure and then broadcast the change event to all connected views. The new user name is reflected on all user screens.

Different types of views can also connect to the same model. In the user name example above, one view might consist of a text box that displays the entire user name, whereas another might consist of a text field that displays only the user's initials. These differences are irrelevant to the corresponding model. It simply send change events to all registered views. Views decide how to display the changed data.

MVC has proved very powerful in representing common data for multi-user applications. It is one of the most popular metaphors for dialog independence in both theory and practice.

2.2.2. Abstraction-Link-View

Abstract-Link-View (ALV) was presented as part of the *Rendezvous* architecture (Hill, et. al., 1994; Patterson, 1994). An *Abstraction* represents application-level, common data that is shared between users. It serves much the same purpose as MVC's Model entity. A *View* represents a user-interface view which can be connected to an Abstraction, again

similar to MVC. However, ALV's View objects embody the behavior of both MVC's View and Controller.

ALV is different from MVC in its *Links*. Links are the real power of ALV. They are created by an independent third-party (i.e. not by the Abstraction or the View) and convey information among Views and Abstractions when needed. The programmer does not need to indicate when information is conveyed. Rather, this information is provided by a constraint system. By keeping Links independent, Abstractions and Views become more general and are easily reused between applications.

While MVC uses messaging as the communication medium between objects, ALV uses only constraints, which are automatically installed and used by Link objects. This effect is more general than MVC where objects explicitly communicate with one other. However, since ALV uses third-party links that define constraints, the Rendevous system must define its own language to use these links. This may be seen as a drawback in situations where programmers prefer to use more common languages such as C++ or Java.

The framework presented in Chapter 4 bears a closer resemblance to ALV than to the other models presented in this chapter in that common Links are created between client-side GUI components and server-side data objects.

2.2.3. Presentation-Abstraction-Control

Presentation-Abstraction-Control (PAC) "recursively structures an interactive system as a hierarchy of agents." (Coutaz, 1990) An agent defines some level of abstraction, which includes a Presentation, a perceivable behavior; an Abstraction, a functional core which

implements some expertise; and a Control, which manages the connection between a Presentation and Abstraction, and contains links to other Abstractions. PAC focuses on intelligent agent-based systems, such as self-configuring networks.

PAC defines independence between Abstractions and Presentations, which normally do not know about each other's existence. The Controller serves the purpose of interfacing the two objects. While PAC does not explicitly define messaging as the communications medium (as MVC does), it specifies an abstract, common communication method. Links may be static or dynamic, single or multiple, and local or distributed. Multiple linkage supports 1:n correspondences between a common Abstraction and multiple Presentations. In this way, PAC is similar to MVC.

In contrast to MVC, PAC Abstractions are not always representations of base application data. Rather, Abstractions are often expert systems or otherwise "intelligent" systems. Presentations provide a user interface window into this specialized behavior, much like an object interface.

2.3. Messaging

Messaging refers to a system of communication between processes on the application level. It allows two different programs (usually on distributed computer systems) to coordinate activities and communicate with each other. For example, one program might specialize in parallel, complex, statistical calculations; it runs on hardware and an operating system specific to high-speed floating-point calculations. Another program might specialize in user interface, residing on a Macintosh computer. The user interface-program uses

messaging mechanisms to open a channel of communication to the statistical computer for its complex calculations.

Messaging is a foundation piece of any collaborative application, since multiple clients need to share data and screen views. Messaging techniques have existed from the early days of computers, and have evolved throughout their history.

2.3.1. Messaging Theories

Several different messaging methods have been proposed in the literature. It is helpful to view these theories as either synchronous or asynchronous. While both methods are often employed in different parts of applications, each has its strengths and weaknesses.

2.3.1.1. Synchronous Messaging

In synchronous messaging, the sending computer blocks execution until the receiving computer responds. This is useful in situations where program execution depends upon the answer received from the remote computer, such as the statistics example proposed earlier in this chapter. Synchronous messaging is used with techniques such as Remote Procedure Call and simple socket connections (Stevens, 1997; Nelson, 1981; Xerox, 1981). These techniques are presented in Section 2.1.3.1.

2.3.1.2. Asynchronous Messaging

In asynchronous messaging (TIBCO, 1999; OMG, 2000; Sun Microsystems, 2000a), the sending computer continues execution immediately after making the remote call. The response is returned in a separate thread of execution to the sending computer. Asynchronous

messaging is inherently multi-threaded and requires advanced platforms that support multi-threading and shared-memory spaces.

Since program execution continues without the need for a response, asynchronous systems are more tolerant of slow or fragile network situations. However, these systems are complex to program because of their multi-threaded nature.

The most common method of asynchronous messaging is the publish/subscribe methodology (it is used in almost all of the products listed in the next section). Publish/subscribe defines a third-party, independent message queue that handles abstract message objects. It serves as a central point of contact for message producers (publishers) and message consumers (subscribers). Consumers are objects interested in being notified when certain types of events are posted to the event queue. Consumers register as listeners to event queues by providing the message types they are interested in. When producers publish events to central queues, the messaging service enumerates over all consumers that have registered for the given event types and sends copies to each.

Since a third-party message queue handles the communication with each listening consumer, the publishing producer object continues with its thread of execution immediately after publishing the event. This separation allows asynchronous messaging architectures to provide high scalability to applications.

2.3.2. Current Products

Because Information Systems research is intricately tied to real-world use, it is helpful to review the current messaging products on the market, including the theories they

use. Nearly all of these products employ asynchronous, publish/subscribe methods, showing this method's strength in production use. Many of these products are listed at MessageQ (1999), and readers are referred to this site for further information. The products are presented in the following sections.

2.3.2.1. ACI: NET24

NET24 (ACI, 1999) is a set of mission-critical middleware products. NET24 solutions provide message integrity and high-performance message delivery .

2.3.2.2. ATB: Transaction Coordinator Technology

Transaction Coordinator Technology (MessageQ, 1999) targets distributed commit coordination and recovery in transaction-based applications. It is intended to facilitate coordination by enabling a distributed transaction monitor using many different protocols. If a transaction aborts, control is left to terminate, roll back, and recover.

2.3.2.3. Alier: Network Data Model

Alier's Network Data Model (NDM) (MessageQ, 1999) provides a common, normalized language for financial data integration. The result is that disparate systems can be integrated with each other immediately. Applications connect to the NDM via Alier's Application Adapters (ready made connections to and from applications), or through conversion routines rapidly created using the Alier DataBridge. Any system connected to the NDM will automatically exchange data with all other systems on the NDM network. The NDM/Adapter architecture enables integration of new systems into the network. Alier

provides live-link interfaces to messaging technologies such as Tibco ETX, Tibco Rendezvous, and IBM MQ Series.

2.3.2.4. IBM: MQSeries

MQSeries (IBM, 1999a) messaging software enables business applications to exchange information across over twenty-five different operating system platforms. Programs communicate using the MQSeries API, a high-level program interface which shields programmers from the complexities of different operating systems and underlying networks. Programmers focus on the business logic, while MQSeries manages connections to the computer systems.

2.3.2.5. IONA: OrbixTalk

OrbixTalk (IONA, 1999) brings messaging technology to the CORBA environment. OrbixTalk is designed to support asynchronous, decoupled messaging, enabling developers to build event driven applications used in real world mission critical environments. OrbixTalk extends the distributed object paradigm into the large scale messaging arena.

2.3.2.6. Information Builders: EDA/Messaging

EDA/Messaging (MessageQ, 1999) enables the disconnected model of client/server computing, which uncouples the actions of clients and servers, and guarantees the delivery of all messages (requests and responses), whether the client or server is up, down, busy, or disconnected. EDA delivers both processing models - synchronous and asynchronous - for heterogeneous databases, applications, transactions, and operating systems in a client/server or Web environment.

2.3.2.7. Inprise: VisiBroker

VisiBroker (Inprise, 1999) for Java and VisiBroker for C++ CORBA object request brokers are designed to facilitate the development and deployment of distributed enterprise applications that are scalable and flexible. They are built to be easily maintained.

2.3.2.8. Sun Microsystems: Java Shared Data Toolkit

The JSDT (Burridge, 1999) has been defined to support highly interactive, collaborative applications. It provides the basic abstraction of a session, and supports full-duplex multipoint communication among an arbitrary number of connected application entities–all over a variety of different types of networks. In addition, this toolkit provides efficient support of multicast message communications, the ability to ensure uniformly sequenced message delivery and a token-based distributed synchronization mechanism. It also provides the ability to share byte arrays among the members of the session.

2.3.2.9. Level 8 Systems: XIPC

XIPC (MessageQ, 1999) provides fault-tolerant management of guaranteed delivery and real-time message queuing, synchronization semaphores and shared memory, all of which are network-transparent.

2.3.2.10. M/Ware: InterPlay

InterPlay (M/Ware, 1999) is a message-oriented middleware product for Windows application developers. InterPlay provides a messaging capability between applications and enables network communications without writing code using a drop-in ActiveX control and

an intelligent data router. InterPlay manages all network connections, data routing, lookup of processes on the network, and other low-level network coding.

2.3.2.11. MicroScript Corporation: MicroScript Server

MicroScript Server (MicroScript, 1999) provides an easy way to read and write to any application. Network sockets, serial ports, files, databases, and even screen sessions are accessed using a library of MicroScript DataStreams.

2.3.2.10. Microsoft: MSMQ

Microsoft's Message Queue Server, MSMQ, (Microsoft, 1999) is a feature of Microsoft Windows NT operating system that provides loosely-coupled and reliable network communications services based on a message queuing model.

2.3.2.11. Oracle: Oracle8i Advanced Queuing (AQ)

Oracle8i Advanced Queuing (Oracle, 1999) is a high-performance, robust, and scalable queuing system that is integrated with the Oracle8i database and can be used to develop large-scale, message-oriented distributed applications.

2.3.2.12. PIDAS: HIT

HIT - Heterogeneous applications Integration Technology - (MessageQ, 1999) is a message oriented middleware product for linking heterogeneous systems and applications. HIT seamlessly integrates applications and provides inter- (and intra-) application connectivity across multiple platforms. HIT synchronizes redundant data repositories (that may reside at geographically distant sites) and ensures instantaneous access to information. Both asynchronous and synchronous communication are supported.

2.3.2.13. Real-Time Innovations: NDDS

NDDS (MessageQ, 1999) provides connectivity between real-time network nodes and Unix and NT workstations. NDDS allows the sharing of data and event information among distributed systems, including multicast capability, client/server functionality, and reliable communications.

2.3.2.14. SAGA: EntireX Message Broker

EntireX DCOM (MessageQ, 1999) is a component-based middleware solution and EntireX Message Broker is a message-oriented middleware solution geared to speed the integration of legacy, custom, packaged and database applications across mainframes, Unix systems and the Windows desktop.

2.3.2.15. Sun Microsystems: Java Message Queue

Sun's Java Message Queue (Sun Microsystems, 1999) software is a message-oriented middleware product designed to support the development and operation of distributed applications on a corporate network. It allows various applications to share data with each other. Java Message Queue handles all the inter-process communication aspects of an n-tier architecture. This enables developers to focus more on the development of the business logic itself and allows for more flexible deployment options.

2.3.2.16. TIBCO: The TIB

The Information Bus (TIBCO, 1999) middleware underlies TIBCO's entire product line, and is based in publish-subscribe communications middleware. It supports both real-time messaging and message queuing for guaranteed delivery.

2.3.2.17. Talarian: SmartSockets

SmartSockets (Talarian, 1999) consists of multiple processes working together over a heterogeneous network (LAN, WAN, or the internet). SmartSockets includes automatic data translation, message routing, publish-subscribe services, peer-to-peer communication, prioritized message queues, flow control, reusable and extensible message type, message logging, asynchronous message transfer, synchronous message transfer.

2.3.2.18. Verimation: V-Com

V-Com (MessageQ, 1999) supports program-to-program communication in a heterogeneous computing environment - client/server communication as well as reliable message queuing.

2.3.2.19. Vertex: NetWeave

NetWeave (MessageQ, 1999) enables queuing, interactive broadcast messaging, updates to foreign SQL databases, and connections to legacy file systems.

2.3.2.20. Xing: OpenMOM

OpenMOM, (Xing, 1999) is a message broker that enables deployment of heterogeneous multi-tier applications over networks (Internet, intranet, extranet, etc.). It supports asynchronous or synchronous two-way object exchange between Java, Visual Basic, C/C++, Python, and Perl distributed components. OpenMOM supports asynchronous messaging, publish/subscribe, load balancing, self describing objects, multi-casting, guaranteed delivery, and fault tolerance.

CHAPTER 3: METHODOLOGY

This section describes the methodology used to conduct this research. The method is firmly based in systems development and engineering methods.

3.1. Systems Development

The *systems development* methodology is described in detail in "Systems Development in Information Systems Research" (Nunamaker, 1992; Nunamaker, Chen, & Purdin, 1990). This research method is similar to other research methods (such as hypothesis testing or theoretical modeling) in that it starts with one or more research questions. Systems Development research methods attempt to answer questions through the development and engineering of systems. Nunamaker, et al., (1990) state:

> Concepts alone do not ensure a system's survival. Systems must be developed in order to test and measure the underlying concepts. Systems development is therefore a key element of IS research. Research methodologies, such as laboratory experiments, surveys, and mathematical modeling, are very

useful, but not sufficient by themselves to form a well-grounded IS research program.

Systems that are developed as part of academic research serve as proofs-of-concept that theoretical foundations are practically possible and feasible. They are critical in answering business-related, real-world questions that Information Systems purports to solve. System development is directly tied to other methods of knowledge discovery. The following figure shows the relationship between the different research methods:

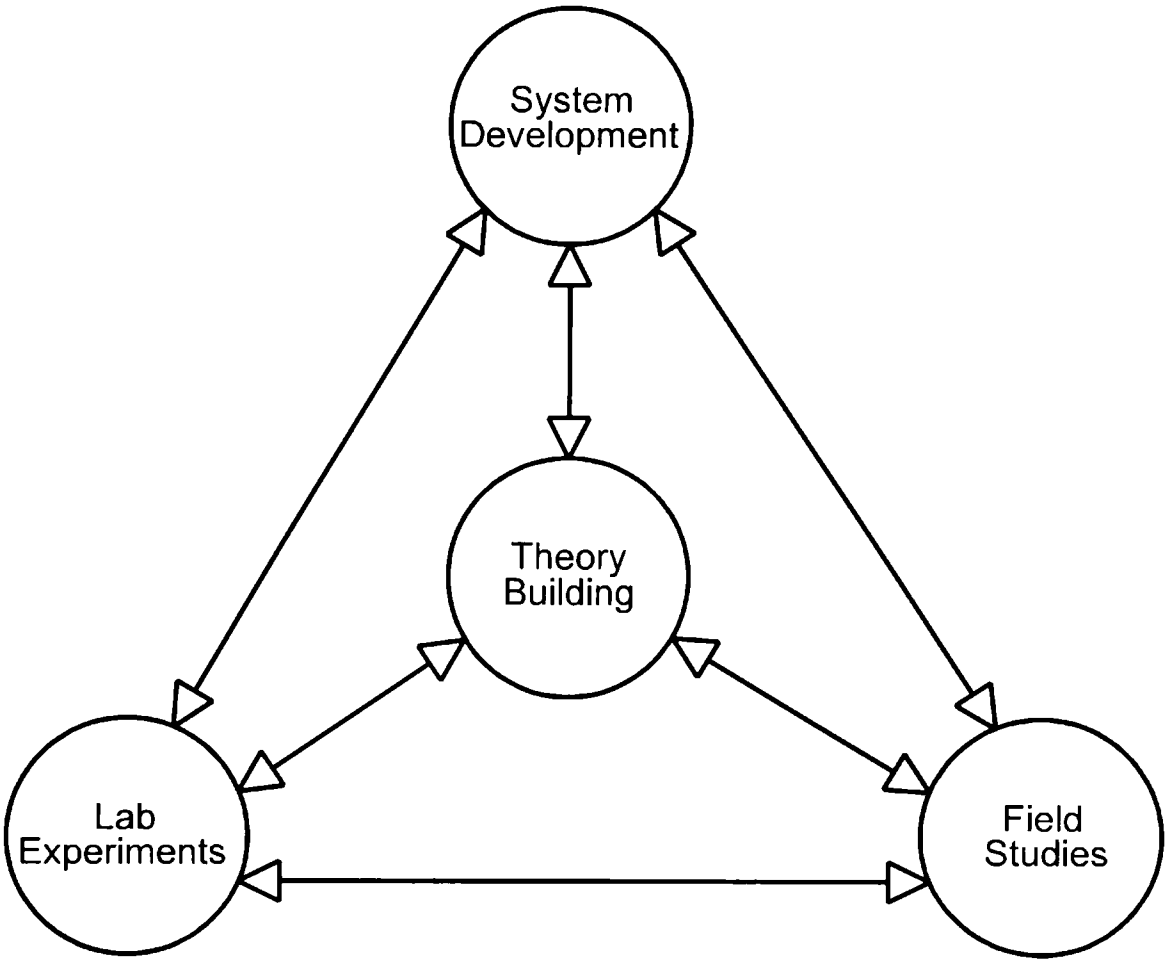

FIGURE 3.1, Systems Development Triangle

The authors continue by describing the following example:

> Basic research into the principle of the airfoil and the internal combustion engine allowed the Wright brothers to bring these new technologies together in the first airplane. Aerodynamics and aerostatics, of course, were not recognized research domains at the time. The existence of an airplane, as both

proof-of-concept and proof that there were still problems to be solved, served to promote these as important branches of engineering.

Today's theoretical tools and models have derived from the principles of flight that were discovered in the original airplane prototypes. Similarly, new Information Systems knowledge is often gained through the building and testing of real systems.

The authors propose a systems development methodology that contains a hierarchy of sub-methodologies. These methodologies are presented in the following figure:

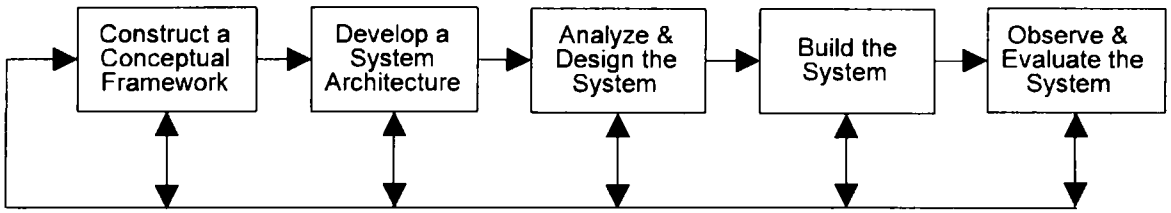

FIGURE 3.2, Systems Development Research Process

The steps of the process are described in the following sections:

3.1.1. Construct a Conceptual Framework

Every new research project begins with a concept of what one wants to accomplish. This concept is often designed as a framework within which the system will be built.

A conceptual framework starts with a declaration of "truth" or something very close to it. The research of this dissertation began with the often-said statement, "there must be a better way!" As stated in earlier chapters, the development of distributed applications is very hard and potentially buggy.

The second step in constructing a conceptual framework is the formulation of a research concept. For this research, it was determined that the abstraction of common distributed functions into an object framework would dramatically simplify the development of these systems. This concept was the seed for the "Property" hierarchy–the foundation of the Collaborative Server.

The third step is the construction of a method. This research progressed by following standard object-oriented design principles and methods. These methods were adapted as the project continued and as the Property structure modified the object-oriented paradigm.

The last step in the conceptual framework is a development of a theory. This research is based in a combination of the theories presented in Chapter 2. These include data sharing methods, dialog independence, and asynchronous messaging.

3.1.2. Develop a System Architecture

A system architecture provides the road map for systems-building research. The architecture of this system was very much determined by its goals: it had to be written within the Java environment, support the development of distributed applications, and abstract the common needs of collaborative applications. The Enterprise JavaBeans architecture was

chosen as the container the server application would run within. The server application would exist as a set of Director and Property objects.

3.1.3. Analyze & Design the System

Analysis and design, firmly rooted in engineering methods, are at the heart of the systems development methodology. Significant time was spent in analyzing current systems and methods for distributed applications. The lessons learned from these surveys, as well as researcher experience, were applied to the initial design of the Collaborative Server. Much of this analysis took place in the years preceding this research project.

By comparison to the systems previously developed by researchers, the design of the Collaborative Server was a significant departure from traditional distributed application frameworks. The principles that had been discovered and presented for several previous years were debated, and many were included in the server design.

3.1.4. Build the System

The Collaborative Server is not just a prototype, but a fully-implemented application that is designed for semi-production use. This is evident in the focus on efficiency and stability that is documented in Chapter 4. Because of these goals, the two main architects of the system programmed the framework together–side by side. The programmers developed the heart of the system at one computer, taking turns at the keyboard. While this programming method took more time, it enabled the system to be better implemented, both in terms of speed and stability. It is the only project that either developer has worked on that used this method.

The design and implementation of the system occurred over a period of one year. The resulting system is described in Chapter 4.

3.1.5. Observe & Evaluate the System

Once the system was finished, distributed applications were developed upon the Collaborative Server. While these applications not part of this research, they provided a test bed for observation and evaluation of the server. The knowledge gained from analysis of these systems is presented in Chapter 5.

3.2. Measurement

Several measurement methods are used in this dissertation because of the differing nature of the two research questions. The first question, *Can a system be developed that meets the outlined goals?* is measured by the description of the system itself (Chapter 4). Much like the first air flight of the Wright brothers, the running, abstracted collaborative Framework provides the answer to the this research question.

The second question, *Is development under this framework faster than traditional methods?* is answered in Chapter 5 using observation and testing of the system with several applications. These applications were developed twice: once using traditional methods and once using the Collaborative Server. Metrics and evaluations such as lines of code and time-to-market are presented.

3.3. Constraints

This system was developed in response to application needs within the business world–specifically because of application needs within the CMI research center. While the

system was not designed using specific research funds from a supporting organization, the applications that drove its development were funded and met a business need for a government entity, military organization, or private business. Therefore, while the system was developed to be abstract and applicable to most programming circumstances, its development was constrained by the goals it was developed within. These include the following:

- The server had to abstract behavior into a framework in its effort to support rapid application development (RAD). Since RAD was the main goal of the server itself, this constraint served as an enabling principle that guided all development.

- The environment dictated a thin-client structure, which in turn drove a full-featured server. The drivers of this constraint were the applications' use over the Internet, within web page applets, etc. Client applications had to be less that 150Kb.

- The framework had to work over slow connections, such as 28,800 modems. This constraint limited the amount of messaging that could be sent between client and server.

- The portability goal drove the development to the Java environment. Java is currently more pervasive and mature than any other full-featured, portable language.

CHAPTER 4: THE COLLABORATIVE SERVER

The CMI Collaborative Server (also called the "Framework") is a proof-of-concept that a highly abstracted, generalized server can be created. This chapter describes its design in detail. Following its presentation, several "lessons learned" from its development are described.

4.1. Server Description

The CMI Collaborative Server started with the goals outlined in Chapter 1. These goals drove the initial development of server models and diagrams. The server plan is shown in Figure 4.1.

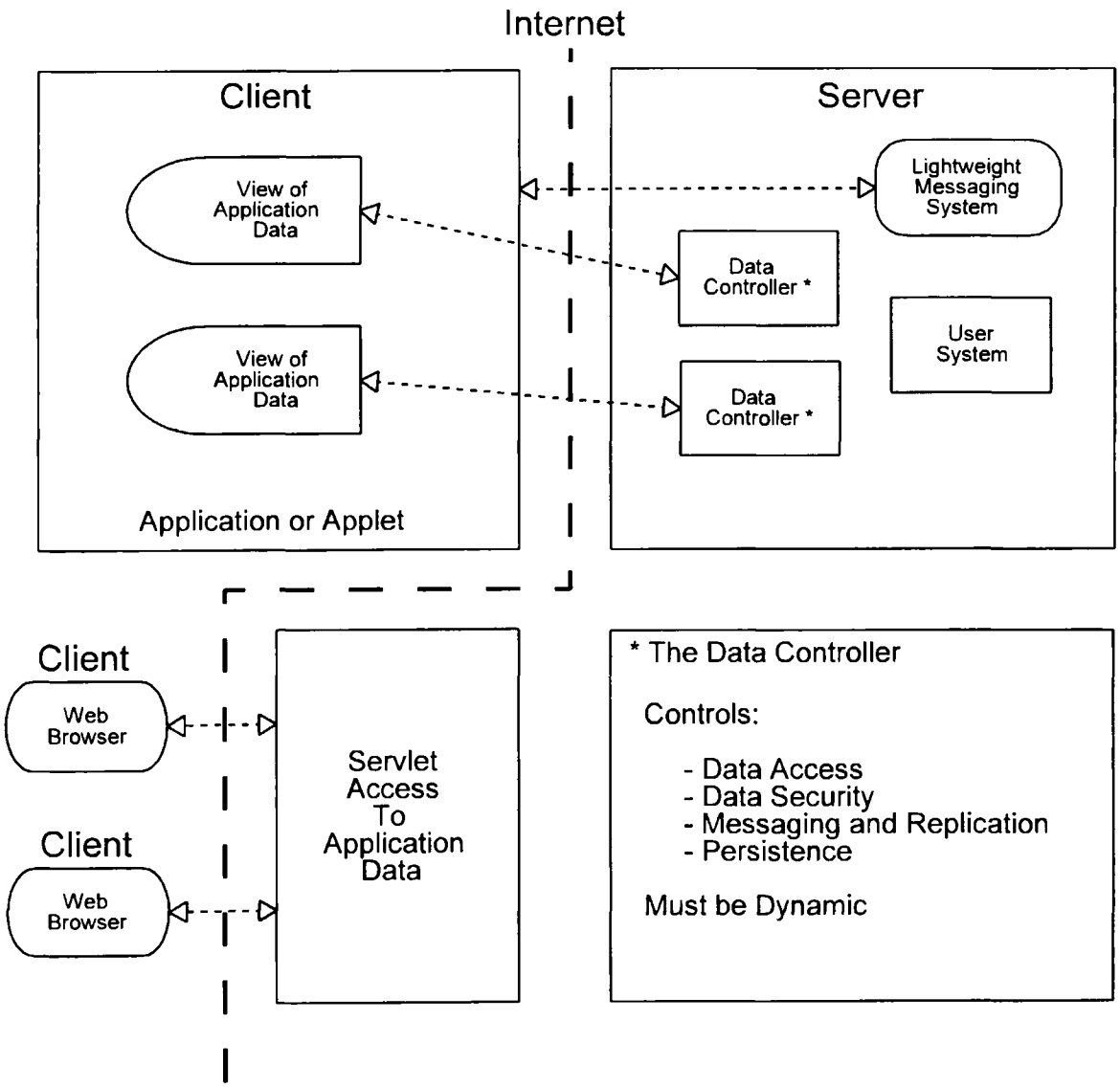

FIGURE 4.1, Initial Server Model

The diagram is separated into four areas: the Application/Applet square, the Servlet access area, the Server square, and the Data Controller square. The Application/Applet square represents the main client view into application data. Following the traditions of MVC and ALV, client applications are views of server-side objects. These views are

connected at runtime to provide dynamic and real-time distributed functionality. Since full-featured clients are not always possible or desirable, the Server must support more basic access to applications. The Servlet area fills this need with an HTML-based, web browser entry into applications. The Framework should handle most of the differences between Applet-based and Servlet-based applications.

The Server square shows the heart of the Collaborative Server: the data controllers. These controllers (which are termed Properties) are described in the Data Controller square. They control access to, security of, messaging and replication for, and persistenceof application data. Controllers are dynamic to allow them to control any type of data. In addition, the Server square shows the need for a lightweight messaging system as well as a full-featured user system.

4.1.1. Introduction

The CMI Collaborative Server is better viewed as a programming framework than as a server. It makes use of several other servers, including an operating system, a database, a Java Virtual Machine, and an Enterprise Javabeans server to achieve its functionality. It automates most of the work involved in distributed programming, leaving only those things that are explicitly application-specific (such as user interfaces) to the programmer. As shown in the following figure, the framework is only part of a larger set of servers and applications.

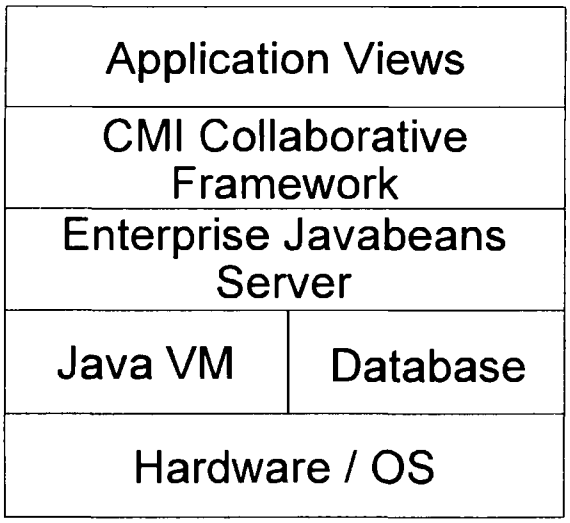

FIGURE 4.2, High Level Design

The *system hardware and operating system* (OS) form the foundation layer of the architecture. They are responsible for managing file structures, TCP/IP and network connectivity, and multitasking.

The *Java Virtual Machine* (JVM) is run from the OS command line. The JVM translates Java bytecodes from the layers above it to the native OS and hardware calls of the platform beneath it. This translation allows the Collaborative Server to be run on any Java-enabled platform, including Windows, Solaris, Linux, Macintosh, and many others.

A *database* provides a means for data persistence. Since Property values are serialized in binary form to the database, several database types are supported, including relational formats, object-oriented databases, and even simple file-based systems. The data persistence scheme is hidden from applications, so it can be swapped at any time without affecting existing code.

The *Enterprise Javabeans Server* (EJB) controls remote access to server-side Property objects. Properties are a special-type of EJB object, which allows them to be managed by any EJB server (The framework currently utilizes an open-source EJB solution.) The EJB server controls the creation and activation of Property objects, including managing

which Properties are in memory and which are saved to the database at any given moment. It also interacts with the database system to persist Properties and their values.

The *CMI Collaborative Framework* is built upon the EJB foundation. It deals with many issues left open by the layers beneath it, including messaging, locking, security, and data scaffolding. To support the original RAD goals of the framework, it includes everything that can possibly be abstracted out of the application layer above it, leaving only those things that must be application-specific to the programmer.

The *Application* layer includes application-specific user interfaces that give clients custom views into the Property structure. It also includes business logic that cannot be managed automatically by framework.

4.1.2. Property Hierarchy

The Property hierarchy is the heart of the Collaborative Server. All activity revolves around Property objects, which provide the foundation for data persistence and control. Properties are organized in hierarchical fashion, similar to object hierarchies. This structure serves as a sort of "scaffolding" for all shared data in applications.

The Property hierarchy is best explained with a well-known example: a User class. Most collaborative applications have a class that contains user information, such as username, password, real name, phone number, e-mail address, post office address, and title. The following figure shows a simplified but typical object-oriented (OO) approach for representing a user:

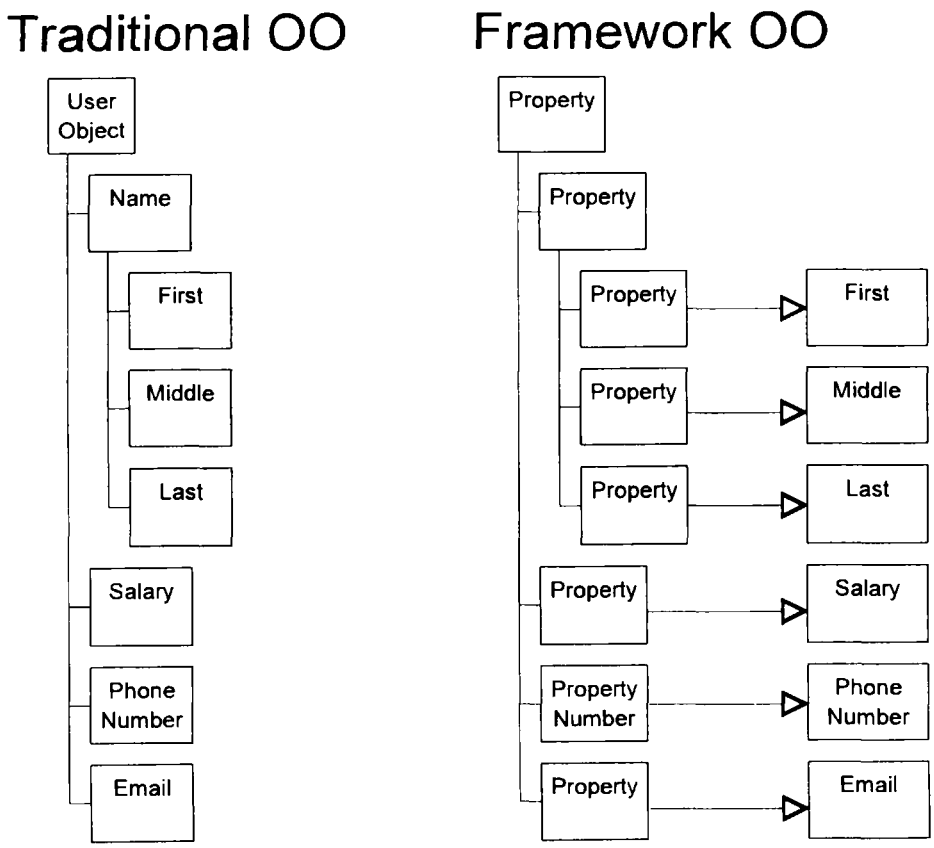

FIGURE 4.3, Traditional OO vs. Framework OO

Traditional OO: The main structure–the user object–contains a sub-object, name (holding string variables for first, middle, and last name), a number variable for salary, a string variable for phone number, and a string variable for e-mail.

The user object holds references to each of the sub-objects it contains, and it manages their existence in memory and on disk. Object-oriented programming typically includes all code required to manage these objects within the class.

The "Framework OO" diagram in the figure is a converted version of the traditional one. It shows the modifications the architecture makes on traditional OO programming. The base data of name, salary, phone number, and e-mail still exist. However, no governing user object references them. Rather, each value has a Property object managing it. Thus, in Framework program design, all management, references, and pointers are moved to the Property structure and away from the actual data objects.

The top-level Property in the diagram above still represents the "User" object. However, no actual data are associated at this level, so it does not point explicitly to data (although it could in other instances). It is assumed that this top-level Property is managed by other, parent Properties above it. The entire structure finally ends in a top-level, Root Property that manages the data of an entire application.

As stated earlier, the Property object abstracts many collaborative and other functions into a single class. Therefore, the eight Property objects shown above are different instantiations of the *same* class. Experience shows that the Property class can be applied to most application domains.

Note also that the values are atomic: string, number, and date. While this is the normal case, these values could also be more complex structures, such as arrays, lists, or even large graphs. The granularity of Property values is governed by application context and purpose.

Each Property governs a single datum, called the Property's *value* (even though that "datum" might be a complex object graph). This value is accessed through the Property's

`getValue()` and `setValue(Object)` methods. A Property manages the following about its value:

- *Security access*: All access to the data behind a Property is governed by an access control list in the Property structure. No other access is given, even through low-level database calls. Security is managed through the following methods:

```
Property.checkPermissions(Principal, Permission);
Property.addACLEntry(AclEntry);
```

- *Messaging*: Property objects are natively collaborative objects. Changes to their data are automatically routed to all interested clients. Messaging is managed by the following methods:

```
ClientDirector.link(Viewer, Property);
Property.fireEvent(PropertyEvent);
```

The `edu.arizona.cmi.collab.events.*` package methods.

- *Viewing*: Property values are viewed by clients using a modified Model-View-Controller architecture. Viewers are created by programmers per application, and are the main focus of application design work .

- *Locking*: Properties can be locked at nine levels of access, giving a client exclusive, shared, or replicated access to a Property's value. Locking is managed by the following methods:

```
Property.setLocked(boolean, Permission);
Property.isLocked(Permission);
```

- *Persistence*: Programmers save data through a Property's `setValue(Object)` method, after which the Framework and EJB server persist the change. No other action is required by the programmer.

Each Property contains a globally unique identifier (GUID), which uniquely identifies the Property in space and time. A GUID is a concatenation of the creation time (in milliseconds) and the server IP the Property was created on. It should never duplicate. The Property hierarchy is maintained by parent and child references kept within each Property. Child Properties can be referenced with any number of indices, which are explained later in this chapter. The Property values do not need to keep explicit references to their parent values or children's values because this information is kept at the Property level.

4.1.2.1. PropertyHandles

PropertyHandles are GUID-based handles to Property objects. They increase efficiency when clients connect to server-side Properties. Since Properties are remote

objects, socket connections are opened for each reference that is achieved. While Java automates and encapsulates this process through the use of stubs and skeletons, each reference opens a real port on both client and server machines.

The tree structure of the Property hierarchy is maintained via multiple references to parent and child Properties. When accessed from remote JVM's, these parent and child references potentially bind a significant number of sockets across the network. If allowed to perpetuate throughout the entire tree and across applications, the number of available server sockets would soon be exhausted. In addition, the processing time required to negotiate newly opened sockets is detrimental to system performance.

PropertyHandles increase efficiency by decreasing the number of required sockets. A PropertyHandle has only one persistent value: the GUID of the Property it references. Therefore, when a PropertyHandle is serialized, its connection cache and references do not serialize with it. The transient connections are made anew in the new process.

Making connections from PropertyHandles to actual server Properties is handled by the `PropertyHandle.getInstance()` method. When `getInstance()` is called, the PropertyHandle first looks in its cache to see if it has already made a remote connection to the Property. If not, it asks its parent PropertyHandle for a reference to the child it points to. Finally, if the parent does not have a connection, the PropertyHandle asks the server to find the object in the database, reinstantiate it back into memory, and return a remote reference. The PropertyHandle caches this connection for more efficient future use.

4.1.2.2. Child Indices

The Property structure is an n-tiered tree structure. The tree structure was chosen because it provides a generalizable, standard way to represent data. Client views are also tree-like structures: the root frame holds references to sub-viewers such as the menu, the toolbar, and the main application window. Each part of the view connects to its related Property on the server. This process of references continues recursively down to basic components such as text fields and buttons. The following figure shows how these references might work in a viewer of the User object example above.

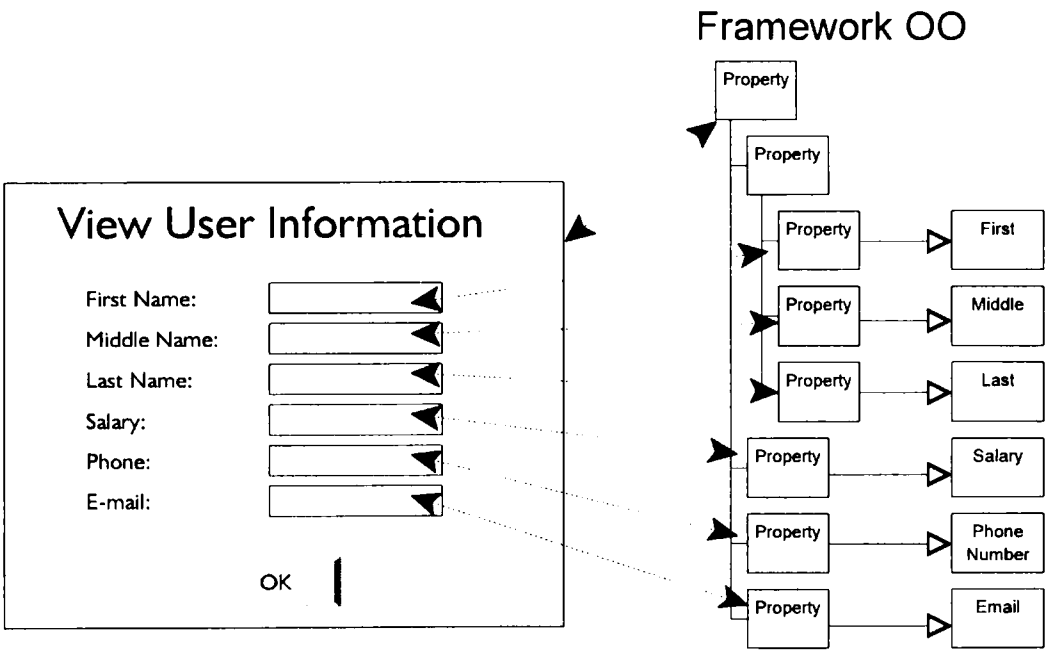

FIGURE 4.4, Viewer References to Server Propertie

When the "View User Information" dialog box on the left is created on the client side, it connects to its specified Property on the server. It then creates its framework-enabled components–text fields in this case–and connects them to their respective Properties to show the data values.

The tree structure does not artificially place constraints on program design because each Property value contains one object. These objects can be as complex as the programmer needs (hash tables or other graphs). Therefore, while the Property structure is tree-based, the Property values can be any structure supported by OO.

Child Indices hold the backbone references of the server's tree structure. Properties hold any number of index types, and each index contains any number of child Properties. Indices are plug-in's which allow different types of references to child objects. The ChildIndexer interface defines the methods of an indexer, and the IndexKey interface defines a key type to be used with a given ChildIndexer. While programmers can create application-specific indices upon these interfaces, three default indices are included with the framework. These classes are as follows:

- *GUIDIndexer*: This index is included automatically in all Properties; it is the default mechanism for referencing children of tree nodes. Children are referenced by their GUID and are retrieved by the same.

- *SequentialIndexer*: This index can be added at runtime to Properties and keeps track of children in sequential order, similar to a linked list structure. It is useful for client-side tree viewers, ordered comments, etc.

- *NameIndexer*: This index, also added at runtime to Properties, keys children by name (NameIndexKey). Children are assigned a unique name when they are added to their parent Properties and are retrieved by the same.

4.1.2.3. Client Proxy Objects

The server adapts standard RMI techniques for increased efficiency and to support RAD programming. Rather than modify RMI stubs (which was the first approach taken), a proxy object has been inserted between the client-side viewer and the client-side stub. These layers are shown in the following figure:

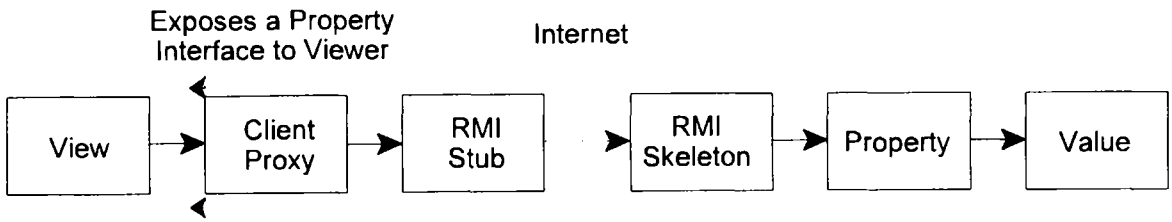

FIGURE 4.5, Client Proxy Objects

RMI stubs normally expose the remote object's interface so local objects can act on the stub as if it were the real object. The client proxy object implements the Property interface so the viewer object can act upon the proxy as if it were the remote Property. Therefore, the RMI stub thinks it is talking to the view, and the view thinks it is talking to the Property.

Efficiency is increased by short-circuiting many calls to the server. For example, since a `getChild(GUID)` call returns a PropertyHandle with the given GUID, the Proxy object simply creates a client-side PropertyHandle with the GUID and returns it immediately back to the viewer. No cross-network call is necessary, and the server is relieved some work, allowing it to process other client needs.

Application coding is made simpler (and thus faster) because Proxies handle security concerns such as sessioning and logins. Sessioning is described in detail in the *User Directory* section below.

For session tokens to be used with all Property interactions, they must be sent across the network with each call to the server. The Proxy object attaches this token automatically to all server calls, freeing the programmer from coding sessions explicitly. In fact, programmers can be somewhat oblivious to the use of session tokens and still program very secure, framework-based applications.

When sessions expire (for example, the user goes to an hour lunch and returns to the application that is still running), the server throws a `SessionExpiredException`. This exception is caught automatically within the Proxy object, which shows a login dialog to

allow the user to reenter their username and password. If successful, a new session token is assigned, and application code continues seamlessly as if no exception had occurred. Without this automation, program code would require while loops *every time* a remote method was invoked, as seen in the following example:

```
// pre processing code here

while (true) {
   try {
      Property.setValue("Hello World", sessionID);
      break; // if no exception thrown, break out of the loop
   }catch (SessionExpiredException ex) {
      // show a login dialog and
      // assign the user a new session token
   }
}

// post processing code
```

The above example shows how application code becomes complex and potentially buggy. Use of the proxy object transforms the example to the following:

```
// pre processing code here
Property.setValue("Hello World");
// post processing code
```

In the second example, the complexity of sessioning is moved to the Proxy object and away from the application coder. The code is much more readable and focused on application-specific logic.

4.1.3. Viewers

The Viewer interface comprises the foundation of client applications programmed within the framework. Most client user interfaces that application programmers design implement a form of the Viewer interface.

4.1.3.1. Bootstrapping The Client

All framework applications use the same common client. This client reduces download time between framework applications because it only needs to be downloaded once. This scheme is similar to the browser paradigm: a common web browser can view many different types of web pages from many locations in the world. The common client also bootstraps distributed connections, freeing application programmers from coding login and authentication scripts and classes.

The common client is responsible for starting the user interface, managing the session securely, establishing a connection with the server event queues, and connecting with an initial, Root Property. All Properties have a default Viewer type, which is the primary user-interface class used to view each Property. The common client uses the `getViewerType()` method to instantiate a default Viewer for the Root Property of the session. It places this Viewer object as the sole component in its main application frame. Alternatively, a Viewer type can be specified by the client to override the default behavior.

The client invokes `ClientDirector.link()` to connect the Root Viewer to the Root Property. This method also registers the Viewer as a listener to the Property's events. The client then calls `Viewer.activate(true)`,which allows the Viewer to query its Property and display data. The `activate()` method recurses this *instantiate* + *link* + *activate* process for all subviewers it contains. This recursive behavior is evident in the creation of the dialog box in Figure 4.3.

4.1.3.2. Interfaces

The framework's implementation of distributed views is defined by a set of interfaces. These interfaces include Viewer, ComponentViewer, and ServletViewer.

4.1.3.2.1. Viewer

The Viewer interface is the foundation of all user interfaces in the Framework. It allows an object to be assigned a reference to a Property. Since it extends Identifiable, a Viewer can be assigned a GUID. This GUID is important for event registration and routing. The ClientDirector adds the Viewer's ID to the end of the route when it registers with a Property's event queue. It also registers the ID with the local EventQueue object.

4.1.3.2.2. ComponentViewer

The ComponentViewer interface defines the methods for Viewers used by the applet- or application-based client. Classes implementing this interface must also extend `JComponent` as they are added to client windows.

The most important method in the ComponentViewer interface is activate(boolean). As described in *Bootstrapping the Client* above, a Viewer's activate method has two responsibilities:

- Loading the initial data from the Property. The method invokes the Property's `getValue()` method to show shared data to the client.

- Initializing subviewers: If the Viewer contains additional Viewers (which are normally linked to children of the Viewer's Property), it should initialize those Viewers by:

    - Creating instances of the Viewer, either by querying the child Property's `getViewerType()` method or by instantiating some other known Viewer.

    - Linking the child Viewer to its child Property by calling `ClientDirector.link(Viewer, Property)`.

    - Activating the child Viewer so it can set itself up and recursively repeat this process with any subviewers it contains.

The interface also defines the standard event methods of `addChild`, `removeChild`, and `valueChange`, which are automatically called by the event system when children are added or removed or the Property's value is changed. These methods ensure user interfaces are kept in sync with current Property data.

4.1.3.2.3. ServletViewer

ServletViewer defines the methods required for Servlet user interfaces to Properties. This interface has been separated from the ComponentViewer interface because of the difference in web-based and application-based user interfaces. HTTP is a stateless, request/response protocol (W3C, 2000). Therefore, server Properties cannot push events to servlet-based Viewers.

A servlet-based Framework application utilizes the common client WebFrontEnd. This common client serves the same purpose as the applet/application common client, including sessioning and logging in. The ServletViewer interface defines one method, `service(HttpServletRequest, HttpServletResponse)`, which is called by the common client after the user's session token is verified.

4.1.3.3. Classes

The framework provides a set of abstract classes which implement much of the behavior required by the Viewer interface set. While these classes are not required for applications built upon the framework, they speed development and decrease potential bugs since they already implement many required methods.

4.1.3.3.1. AbstractViewer

AbstractViewer provides implementations of all methods except `activate(boolean)` and `setEditable(boolean)`. It provides an ID variable for the Viewer and empty method bodies for event methods, similar to Java's Adapter classes. This allows Viewers to ignore events they do not want to listen to. Most applet/application-based user interfaces extend AbstractViewer.

4.1.3.3.2. ActiveViewer

ActiveViewer is an extension of AbstractViewer that allows a viewer to participate in the client-side framework event system. This system is *entirely different* than the Property/Viewer event system described later in this chapter. The client-side event system mirrors Java's event system, interacts only within each client JVM, and allows Viewers to pass events to each other.

ActiveViewers participate in the Collaboratus application. They allow Collaboratus to get default menus and toolbars and provide a means for ActiveViewers to request activation from Collaboratus.

4.1.3.3.3. AbstractServletViewer

The AbstractServletViewer class implements the common behavior for servlet-based framework applications. Its primary responsibilities include:

- User login through usernames and passwords.

- Session management once users have logged in. If the session times out, the user is automatically permitted to log back in without interruptions in application code.

4.1.4. Directors

Directors bootstrap Framework clients and servers. One Director lives per JVM and controls all framework activities within that JVM. Directors are responsible for the following:

1. Reading settings files for information such as naming service host and port, Root Property and Root Viewer, and event factory class type.

2. Setting up the VM-wide message queue and related thread pools, and connecting the queue to a specified server/client.

3. Holding references to the server's Root Property and user directory.

4. Directors live outside of the EJB server. They hold references to EJB objects (such as the user directory and the Root Property), but they are created each time the JVM starts and export themselves via standard RMI.

4.1.4.1. The Server Bootstrap Process

The following description shows the process of framework initialization within a new JVM:

1. If this is a new installation of the server, an Admin object is created; otherwise, the Admin reference is retrieved from naming service. Admin is an EJB object, so this happens via the AdminHome EJB interface.

2. `Admin.start()` is called. This method does the following:

   A. Retrieves a reference to the EJB "Directory" object, which holds all users in the system.

   B. Starts a *ServerDirector* by calling `ServerDirector.init()`. The user directory reference is passed to the Director. Init does the following:

      (1) Loads the specified settings file. This file determines the Root Property, Root Viewer, naming service host & port, etc. It sets default values for all values not explicitly specified in the settings file.

(2) Exports itself to the naming service so it can be accessed from other VMs.

(3) Instantiates the object factory. This factory is used to create new Properties.

(4) Instantiates the event queue that will be used to pass message events to connected clients. It instantiates an event factory to create new events.

(5) Sets the default route for the server. This route is used in creating messages sent from the server to clients. Since the server location doesn't change while the server is up and is always included in the route, this is only calculated once and then cached for the duration of the server's life.

3. `Admin.start()` then finds the Root Property through its home interface. During installation, the Root Property's GUID is stored in the Admin object. Therefore, this value is used to find it in the database. Once found, the Root Property is reinstantiated into memory.

4. The server is now ready for use.

4.1.4.2. The Client Bootstrap Process

The following describes a client startup from a normal command-line interface. The applet startup is nearly identical.

1. ClientApp reads initialization parameters from a settings file that is specified on the command line. This file determines the Root Property, Root Viewer, naming service host & port, etc. It sets default values for all values not explicitly specified in the settings file.

2. `ClientApp.showFrame()` is invoked, which performs the following:

   A. Creates a ClientApp object, which extends JPanel, and places ClientApp as the sole component in a top-level JFrame window.

   B. Instantiates a ClientDirector within the client JVM. ClientDirector looks up the ServerDirector in the naming service and keeps a remote reference to the server for the duration of the application run. ClientDirector sets the remote reference to the Root Property, and

then creates an object factory which is used to create new Property objects from the client side.

C. Creates a new event queue and registers the ClientDirector as a delegate of the server's event queue. All routing information is set up for messages to be passed from the server to this client.

D. Instantiates the Root Viewer and links it to the Root Property on the server. ClientApp calls the viewer's `activate()` method, which recursively creates child viewers and connects them to server Property objects.

E. ClientApp calls the Login class which allows the user to enter their username and password. If successful, the user is assigned a session ID which is added to the active sessions on the server. This session ID is also stored on the client and is injected into all calls to the server. It serves as the client's permissions token for all activity on the system.

3. The ClientApp frame is finally made visible. The application is running for the client.

4.1.4.2.1. Changes for a Servlet Bootstrap

A servlet client acts as proxy for clients coming from web browsers. The servlet, while a client, actually resides within the web server container. It acts in all ways as a regular client to the system. Since it resides in its own JVM (within the web server container), it is remote to the ServerDirector.

The main difference in ServletDirector when compared with ClientDirector is that many client browsers share the same servlet. There may be 10-100 clients accessing the servlet, which acts as a single client to the server. This means that the ServletDirector must be stateless–i.e. it cannot hold default values such as Root Property, Root Viewer, etc., since these values change across users.

Default values are specified in each request to the system through the "Root" and "View" parameters. In addition, the client browser's session ID is be passed with each request. The values are not stored in an HttpSession object because implementations of sessions change from server to server and are dependent upon whether the client accepts cookies. To overcome these inconsistencies, the values are explicitly passed with each call.

The ServletDirector does not set up an event queue and does not add itself as an event delegate to the Server. Therefore, servlet-based Viewers are never registered with Properties as event listeners. Rather, they connect to Properties, read values and perform updates, and then disconnect with each request. The reason for this is that the messaging system uses "push" collaboration. Since data cannot be pushed to browsers (HTTP is a "pull"

request/response protocol), there is no way for the ServletDirector to inform browsers of changes. Browsers must explicitly refresh their screens to show Property data updates.

Because of the lack of push support in browsers, Servlet-based applications normally perform low-to-non-collaborative functions. Regular collaborative tools use application-based architectures. The main use of the Servlet-based architecture has been browser-based administration panels or browser-based meeting starter pages (which start up a local applet-based program for each meeting started).

4.1.5. Event System

The goals of the CMI Collaborative Server can be mapped onto the framework's event system. First, the messaging system should be efficient. Since the framework uses thin clients that often connect from within low-bandwidth environments, the system sends events only when necessary. Efficiency is the first reason the Framework includes a custom messaging service which is designed specifically for the CMI Collaborative Server. This system achieves many efficiency gains due to its specificity.

Second, the event system must be robust. As long as clients are connected, they should be guaranteed to get the events they have subscribed for. In addition, measures should be taken to ensure clients get events in order, no events are skipped, and events are not modified or corrupted in route. The server includes many features to increase robustness. These features are described in the next section.

Third, the event system should be portable across servers. A custom messaging service achieves this portability because no third-party libraries or classes are used.

Finally, the event system should be as transparent as possible. Application programmers should need to know very little about the messaging taking place. They should not need to subscribe or unsubscribe from event queues and should not need to generate events manually. Encapsulating the messaging system within the Framework supports the RAD goals of the overall framework. The server encapsulates almost all messaging behavior, which is described in the next section.

4.1.5.1. Event System Description

The CMI messaging system is based upon a modified publish/subscribe scheme. When a Director initializes, it creates an EventQueue object and publishes this object in the shared naming service. The Director also starts a thread pool to manage the items in its event queue. One thread is responsible for event forwarding and routing; the remaining threads are assigned to event objects at their destination queues.

The framework allows for n-tiered event queues. The source object creates an array of GUIDs describing the route the event should go follow. The event queues push the event along this route, popping a route ID off the array at each stop. When only one ID remains, the event queue assumes it is the ID of the destination object. It then forwards the event to this object within the same JVM as the last event queue, where the event is processed on the thread pool.

4.1.5.1.1. Event Queue

Each event queue is responsible for forwarding events received in a first in, first out manner to the next step in their designated routes. When an event queue initializes, it creates

a thread pool with default size of 25 items. These threads start immediately and go dormant until an event arrives in the queue. A thread checks the size of the queue every few seconds to ensure it is efficiently managed and events are handled in timely fashion. Since only one queue exists per JVM, this short time delay does not adversely affect system performance. In addition, when events are placed into the queue, the manager is explicitly interrupted; no wait is required for event processing.

4.1.5.1.2. Thread Pool

One static thread pool exists in each JVM. It is assumed that if the thread pool size is inadequate, the size of the thread pool can simply be increased rather than adding the additional overhead of new pools. A managing thread is responsible for assigning free threads to objects. In addition, a balancer thread is responsible for balancing the load placed on the pool. If most of the threads are assigned at any given time, the balancer increases the size of the pool up to a maximum number of threads. The reverse occurs if threads go unassigned for several minutes down to a minimum number of threads.

Any object that contains a run method can be processed on the pool. If free threads are available, the manager starts the object immediately on one of the threads. If all threads are busy and no more can be created, the manager retries to assign the object every second.

4.1.5.1.3. Routing

The Framework uses an abstracted routing system. A queue simply sees routes as an array of stops. Since Framework events always traverse from Property to Viewer, each route starts with a Property GUID and ends with a Viewer GUID. While the Framework supports

the passing of events across n-tiers, events normally traverse from Property to Viewer as shown in the following figure:

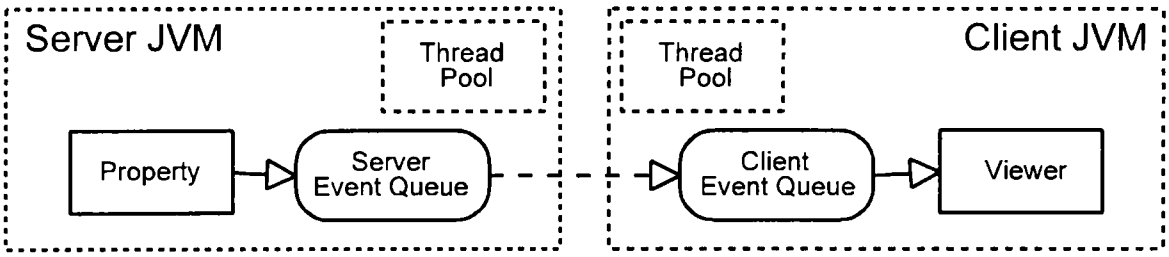

FIGURE 4.6, Event Routing

4.1.5.1.3.1. Failover and Rollback

Two potential errors occur when events are propagated through their routes:

1. A null pointer exception is thrown because a) the Viewer no longer exists, or b) the client disconnected during an earlier server session and the server has since restarted, resulting in a null remote reference (since the local stub no longer exists).

2. A connection exception is thrown because the client disconnected during this server session. The local stub still exists but has lost its socket connection.

Either of these error signals an invalid route: the publishing Property needs to be notified to remove this route from its list of event listeners. Since the remote connections link JVM to JVM, the event must be propagated backwards through its route to its origin. This is accomplished by setting a rollback flag on the event and posting the event to the JVM's event queue, whereupon it starts is backward propagation. When it arrives at the server, the Property is notified to remove the route. Using this mechanism, listener lists stay clean and current. Viewers also unsubscribe automatically during a clean logoff–described in the next section–which prevents rollbacks from occurring.

4.1.5.1.4. Viewer Subscription

The Framework is described as a *modified* publish/subscribe scheme because Viewers automatically subscribe themselves as listeners to Properties. Since Viewers are always associated with Properties, they are subscribed in the `ClientDirector.link()` method. Viewers usually call `link()` and then `activate()` on all sub-viewers, causing recursive subscription of child Viewers and Properties.

During `activate(false)`, Viewers remove their sub-viewer links to server Properties and unsubscribe these Viewers from Property event lists. If Viewers fail to do this, the subscription is automatically removed through the rollback process the next time the Property publishes an event.

4.1.5.1.5. Publishing Events

Properties are virtual event queues (see Socket Management below) because they hold lists of listening Viewers. The term "virtual" is used because Properties do not actually

publish events, even though they seem to. Publishing behavior is moved to JVM-wide event queues for efficiency purposes. Likewise, Properties do not know what is contained in their listening routes, but simply fire events to each route in their list.

The destination references of events are not set directly by Properties. Rather, Properties set abstract routes. When events reach their final queues, the Framework reads Viewer destinations and sets actual references. This is done because Viewers are not remote objects and cannot publish distributed references. Handles are used in routes to overcome this limitation.

4.1.5.1.6. Socket Management

RMI establishes socket connections between local and remote objects. While a maximum of 65,536 sockets are expected to exist in operating systems, most operating systems do not have this many actual sockets. Rather, operating systems provide a *virtual* number of sockets and an unknown *actual* number of sockets. Since many Viewers within the system need to connect to many Properties, all available sockets would soon be used up–especially on the server where many thousands of clients might be connected at the same time.

Socket limitations are one of the foremost reasons for the custom event system. To better manage socket resources, only one JVM-wide event queue publishes itself to the naming service. Viewers and Properties actually register their references with their local EventQueues, preventing the need for socket connections. As stated earlier, Property listener lists store routes rather than actual remote references. These routes are passed through the

messaging system until they reach their destination JVM. The destination queue maintains a list of GUIDs and actual Viewer references, which it uses to set the real reference of the final destination.

4.1.5.1.7. Events

The framework defines a standard event superclass which provides route information, source, destination, and a rollback flag. Traditional event theory is modified in that events *are somewhat aware* of their destination. That is, they know they are always headed toward Viewers. Traditional event theory requires that destination objects implement a standard listener interface; publishers and events are oblivious to their destinations. The drawback to traditional event theory is the necessity of additional interface class definitions. These definitions must be downloaded for each type of event, causing additional network usage.

Since destination objects in the framework are always Viewers, events can include behavior as well as data. When an event reaches its destination JVM, the event queue invokes its `doEvent()` method, which all framework events implement. Thus, the event acts upon the destination object (this behavior is opposite of traditional event theory, where the destination acts upon an event). The change significantly decreases event system bandwidth requirements.

Since a Property represents an atomic piece of datum in hierarchical format, very few actions need to be handled. These include locking, updating the value, adding children, and removing children. Almost all required system functions can be factored down to these actions.

4.1.6. User Directory

The framework includes a persistent user directory. A user in the real world is represented as a common user object in the system. Users are the basis for all security and ownership within the framework.

4.1.6.1. A Custom User Directory

The framework includes a custom user directory for two reasons: Java has no user specifications for data access, and the proprietary nature of third-party user libraries limits portability if they are used.

4.1.6.1.1. Java's User Specification

The Java specification takes security very seriously. It is one of the foundation principles of the language. However, Java's notion of security revolves around the safety of remote code being run on local machines. Therefore, it is very code-centric and not data-centric. Java defines the following two security standards:

- *Java Security Packages* (Sun Microsystems, 2000a): Defines classes for both security and access control and provides a means to enforce access controls based on where code came from and who signed it. For example, a local machine might give full filesystem rights to code downloaded from the local network and read-only rights to code downloaded remotely. More specifically, the policy might specify that code downloaded from the *sun.com*

address space and signed with a certain certificate be given printer access but not filesystem access.

- *Java Authentication and Authorization Service (JAAS)* (Sun Microsystems, 2000c): The JAAS specification defines one additional dimension to the standard Java security packages: *who* is running the code. JAAS is based in standard Pluggable Authentication Modules, which might include username/passwords, smartcards, fingerprinting, or other means. Therefore, the JAAS framework gives control over three factors: who is running the code, who signed the code, and where the code is located.

4.1.6.1.2. Limitations of Java's Security Models

Since JAAS is more powerful and fully inclusive of the standard Java security packages, it will be used for comparison. JAAS is limited in two ways: the local policy file and the lack of data-centric security.

4.1.6.1.2.1. Policy File

JAAS retrieves its security definition from a text-based policy file on the local system. Therefore, changes to access rights must be made to this local file. This poses a severe limitation for collaborative systems, where an administrator or facilitator needs to modify user rights system- or meeting-wide. JAAS requires the facilitator to change individual files on all machine meeting participants use.

4.1.6.1.2.2. Data Security

The Collaborative Server is more concerned with securing data than it is with securing code. In collaborative applications, it is assumed that all users have access to run the application, but not all users have access to all data. JAAS gives no support for access over data. Facilitators need fine-grained control over the access of different users to the viewing, changing, adding, and deleting of meeting data.

4.1.6.1.3. Application Server Support

Different application servers that support Java, such as IBM Websphere, BEA Weblogic, Gemstone, Sun Forte, and others, fill the gap in data security by providing user and security classes. Users can be assigned different rights throughout these systems and databases.

However, application servers implement user classes in proprietary ways. Therefore, once an application uses one of these proprietary classes, it is tied to the given server. This lack of portability violates the goals of the Collaborative Server.

Because of these two limitations, the framework includes its own user directory. This directory moves with the server into any application server and provides framework-specific functionality.

4.1.6.2. User Classes

The user directory is made up of two related elements: the user class and the directory interface. The user class provides basic user information: name, e-mail, password, etc. It also stores a unique id which is used as the user's token throughout Framework access control

lists. This id never changes and always references back to the user it represents, allowing users to change usernames and passwords without causing system-wide updates.

The Directory interface extends the Property interface and provides the same collaborative functionalities as Property does. Additional behavior supported by this class includes logging in and out, retrieving users, and sessioning. Users are added as child Properties to the singular directory object in the system.

4.1.6.3. Sessioning

A best practices model of user verification includes sessioning. When users successfully log in with valid usernames and passwords, they are assigned unique session ids (GUIDs) that are stored on the server. Clients include their assigned session ids in every call to the server as their authentication tokens. This prevents usernames and passwords from crossing the network unnecessarily. In addition, session ids are invalidated when users log out, ensuring security even if tokens are snooped on the wire. Further, session ids time out if they go unused for more than 30 minutes. Users must login once again to be assigned new session ids. Relogin is automatically handled by client proxy objects

4.1.6.4. Access Control Lists

The framework uses access control lists (ACL) to enforce security in the framework. ACLs are used in most modern operating systems and databases. Each Property contains an ACL, which holds user rights to that Property. All activities on an object are mapped into ACL entries, resulting in nine levels of security access. These levels include get value, set

value, get child, add child, remove child, modify ACL, set parent, get parent, and lock. Users can be given rights or locked out of any or all of these nine levels of security.

Users who create Properties are automatically added with all rights to Property ACLs. If this were not done, no person would be able to access a Property after its creation! This owning user can then add other users to the ACL. ACL rights are hierarchical in nature. If a user accesses a Property and is not explicitly included in its ACL, the parent Property's ACL is checked. This behavior continues recursively to the Root Property in the system. A user who is given superuser access to the Root Property automatically inherits rights to the entire system.

4.1.6.5. Locking

Property objects can be locked by users. Users automatically have 30 minute leases on locks they successfully obtain. After 30 minutes the object is unlocked automatically, preventing persistent locks from tying up object indefinitely. For example, if a user were to lock a Property and then crash his or her client, the Property might remain locked forever. The use of leasing prevents this persistent behavior.

Locks are permission-specific and support partial-locking of Properties. For example, a user requesting a write lock would create a lock object with a set value permission. This lock allows other users to gain get value and get parent access to the Property, but not write access.

4.1.7. Firewalls

Firewalls are a fact of any enterprise system on the Internet. Firewalls are both complex and limiting from the programmer's perspective. The vast number of different firewalls on the Internet creates significant challenges for distributed applications. Firewall issues arise in the following areas of distributed applications:

- Naming Service: The RMI registry or other naming service must run on an open port, usually 1099. The settings files allow Directors to use other ports if the firewall requires.

- Connections to the Server Director: This Director exports itself via the naming service. Each new remote connection to this object opens an unspecified port between client and server.

- Connections to Properties: When clients make calls to Property objects, new connections are opened on unspecified ports between client and server. These open connections are also cached as long as system resources allow.

- Client event queues export themselves to server event queues. This behavior poses the most serious firewall issue since potentially outside-firewall clients often try to export to inside-firewall servers.

Because of the complexity involved in firewalls, behavior has been included in the Framework to automatically work around network limitations. Encapsulating these issues into the Collaborative Server allows programmers to develop applications quickly and efficiently. The Framework uses a multi-layered approach at achieving connectivity in firewall situations. These are described as follows:

1. The client and server first try to connect using regular RMI. This assumes that open ports are available through the firewall.

2. If the client cannot connect to the naming service, the system fails. At least the naming service must be exported through the firewall.

3. If connections to the server fail, RMI automatically tries two additional approaches. These approaches are coded into the standard RMI distribution. (Sun Microsystems, 2000b).

    A. The call is forwarded directly to the naming service port, which is guaranteed to be open or the system would have failed earlier in the process. The naming service unpackages the call and forwards it to the appropriate port once inside the firewall. This approach only fails if a firewall prevents inside port forwarding.

B. If inside port forwarding fails, the call is forwarded to port 80 as an HTTP POST request. A servlet unpacks the POST request and then forwards to the appropriate port. Since almost all firewalls allow traffic on port 80, this final attempt usually succeeds.

The forwarding approaches have serious drawbacks, though. Since HTTP is a request/response protocol, the server can only respond to incoming requests. It cannot push data in the same sense as it does in non-firewall situations. This is solved with changes to the messaging layer.

Client-side event queues attempt to export themselves to the naming service. If this fails, the server creates an event queue proxy on the server side (within the firewall). This proxy acts as a buffer and collects events that are pushed to it. The client connects to this proxy in a polling fashion (every few seconds) and pulls events via regular RMI mechanisms. This type of polling gives the approximation and appearance of real-time collaboration, but works through firewalls much better than direct connections. However, the disadvantage is the increased network traffic and server processing requirements that polling creates.

4.1.8. Summary

The CMI Collaborative Server comprises many different systems, including the Property hierarchy, client-side Viewers, Directors, a full-featured event system, a security system, and a user directory. The interrelationships among these systems is shown in Figure 4.6.

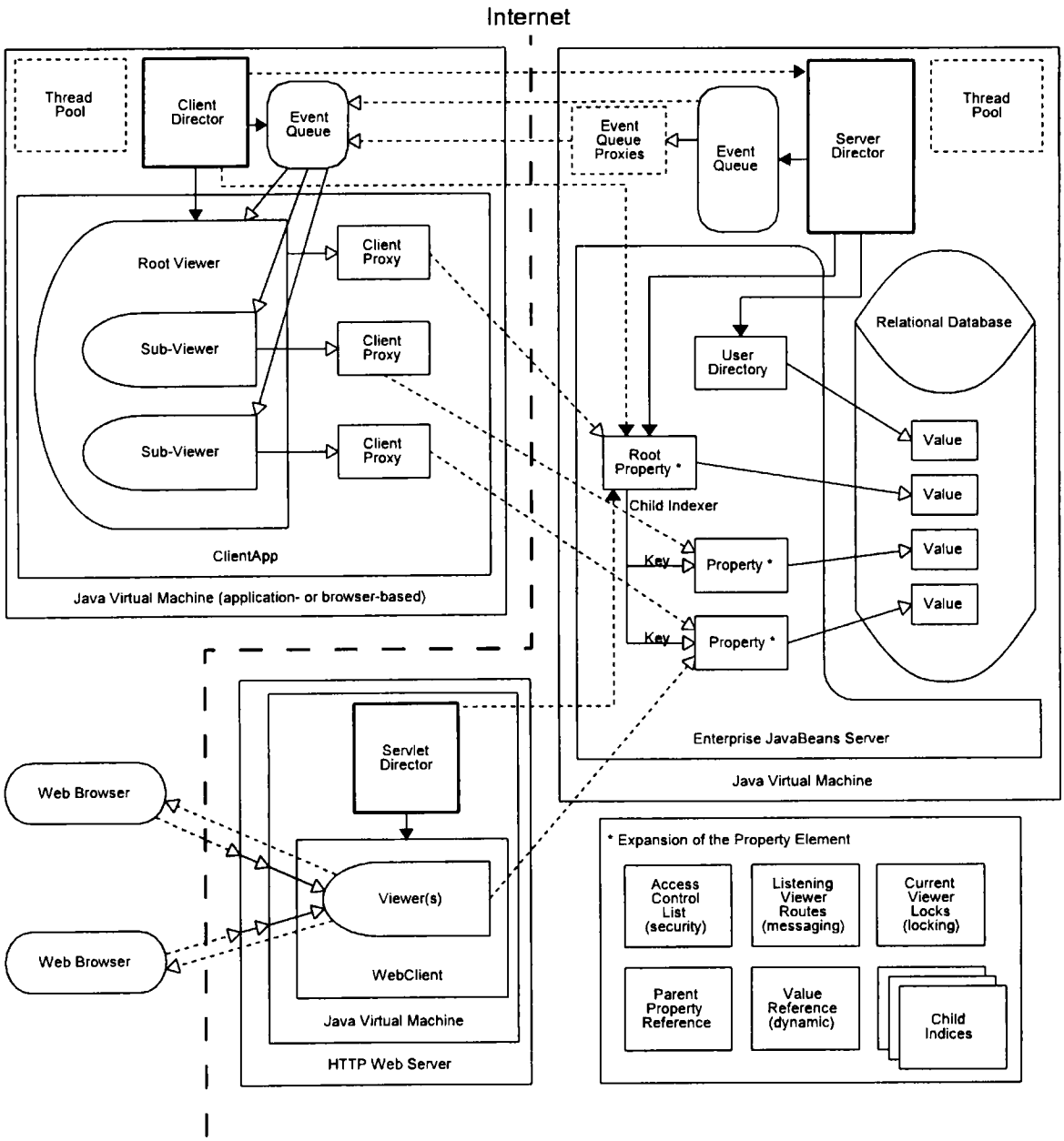

FIGURE 4.7, The CMI Collaborative Server

Figure 4.7 should be compared with Figure 4.1, the initial Server model. In effect, Figure 4.7 builds upon the earlier model and implements its systems in detail. The figure is

again divided into four sections: an Application-based client, a Servlet-based client, a Server square, and an expansion of the Property concept.

The model also introduces a number of improvements to the original model. The three Directors control behavior in their respective JVM. They are shown with bold lines, indicating their importance to the entire system. Directors control initial system bootstrap (for both clients and server), initialize messaging and routing, and hold references to the Root Property. Director references always end with bold arrows.

The hierarchical nature of Properties and Viewers is also shown in the diagram. The initial server design did not define how server-side data would be related and referenced. The inclusion of ChildIndexers and IndexKeys in the model provide the backbone and scaffolding for application data.

The model shows the location and placement of client Proxy objects. These Proxies increase system efficiency and streamline application code. Their RMI connections across the Internet to Properties are depicted with dotted lines, which are inclusive of stubs and skeletons that handle wire protocols and socket connections.

The lightweight messaging system has been implemented in this model. It includes JVM-wide event queues and thread pools. Event queues significantly decrease the potential number of actual socket connections because routes are given as IDs rather than as real connections. Client-side viewers are registered with client-side event queues, where destination references are assigned to events that traverse the system. Note the flow of data

from Viewers directly to Properties, and from Properties through the messaging system to Viewers. The Servlet square has no event queue because of the stateless nature of HTTP.

4.2. Comparison to MVC and ALV

The Framework's viewer system is built upon the same principles as MVC and ALV–described in Chapter 2. Despite the similarities between the three models, some differences exist. A major difference from traditional MVC is that the Framework distributes the objects across JVMs. Most MVC applications target local (within process) applications and, consequently, utilize heavy-bandwidth event systems. The framework's implementation is targeted at efficient network usage.

Another difference is that Properties (MVC's Models) normally drive which Viewers (MVC's Views and Controllers) are used. This is opposite of MVC where different Views are windows into non-active data. However, since each Property controls a piece of data, it also dictates the default viewer for that data. This allows applications to deal abstractly with all kinds of Properties without being tied to certain Viewers. Resulting applications are highly dynamic in nature.

Additionally, this model is more defined than MVC. Viewers are intricately tied to Property objects, client proxy objects, and the messaging system. The Property structure defines the standard "model" in the framework, where MVC applications usually define models specific to each domain.

The Framework is more similar to ALV than to any other architecture. While the *nature* of ALV's links and constraints are very different than the Collaborative Server's

messaging system, the *use* of these elements is very similar. Both systems feature automatic connections, connect every client-side viewer to server-side object, structure views and data in hierarchical fashion, and recurse behavior across the hierarchy. In contrast to ALV, the Framework includes many more features, such as locking, firewall ability, and standardized Property objects.

4.3. Lessons Learned

Design and implementation of the Framework yielded many lessons learned. These are presented as follows:

4.3.1. Impedance mismatch

The *impedance mismatch* refers to inequalities between the object-oriented and relational database paradigms. For example, an object contains inherent behavior; relational databases do not. In fact, relational theory explicitly divorces behavior from data, moving behavior to scripts, procedures, and functions. Further, object graphs are most often hierarchies, with one object holding references to several other objects. Searching capabilities within this object hierarchy is limited to the methods exposed by each object. In contrast, relational databases can be searched in ad hoc fashion in any number of ways. Relational tables are always two-dimensional in nature, while objects are inherently complex structures. Objects can contain user-defined data types (i.e. other objects and structures); most relational database applications provide only a few standard data types. Anything beyond the standard types must be stored in binary large object (BLOB) fields. Once stored, a BLOB field gives no indication as to its data type, behavior, or access. One of the foremost

advantages of relational databases is that the data is "self-describing"; applications are not needed to access data (Codd, 1970). The use of BLOB fields contradicts this advantage. Since each Property contains an untyped value, the Property table within relational storage scheme hold BLOB data fields. This data cannot be queried, typed, or accessed without client applications.

4.3.2. Object Orientation

Object-oriented programming provides a powerful way of modeling the Property structure. Without this paradigm, a generalized server of this type could not have been built. Specifically, *encapsulation*, or the ability to hide data and functionality within an object, proved invaluable in creating several parts of the framework. Black-box objects, such as the client-side proxy and even the Property object itself, provided collaborative functionality without programmers needing to understand the many problems associated with networking, RMI semantics, Enterprise Javabeans architecture, and database access languages such as SQL.

Polymorphism, the ability to treat Property values as simple objects (even though they represent a large number of different types of objects), provides the means for the framework's dynamic data structure. It allows a single Property definition to control vastly different and complex objects. Further, polymorphism allows the use of different child indices to be plugged into Properties. This capability gives a normally static tree structure very dynamic parent-child abilities.

4.3.3. Just-In-Time Linking

The Framework only connects client Views to server Properties when needed (although applications can override this behavior). Since a complex client user interface is often connected to a large number of Properties, many socket connections are potentially used. Just-In-Time (JIT) linking eases this need by making connections only when needed. Practically, this occurs when a user first visits a new part of the application (such as clicking on a new tree node, opening a new form or dialog box, etc.). While there is a quarter second delay in visiting the new section of the program, the savings in system (especially server-side) resources is significant. When the user leaves a section of the program, PropertyHandles cache connections with "soft references". Soft references are kept as long and system resources as available. When system resources become limited, the references are freed up and reconnection is required.

4.3.4. Firewalls

The first version of the Collaborative Server made no special attempts to work through firewalls. When the first applications were taken to production environments, they almost always ran into firewalls and were unusable. Further, every firewall environment was unique from others, including client-side and server-side security. The one commonality found in firewalls is that port 80 is almost always open for traffic. As long as an application uses this porthole, almost any type of application data can be tunneled without upsetting security personnel. While RMI contains algorithms to work around firewall limitations, special code has been written to adapt specific systems to work through port 80.

4.3.5. Enterprise Javabeans

Enterprise Javabeans (EJB) provides a standard procedure for accessing databases from Java-based applications. It has received significant attention in the media and from companies eager to implement it. EJB is an interface specification for remote access to server-side data, usually using RMI's JRMP and/or CORBA's IIOP protocols. EJB gives vendors a common platform and process for application servers on the market, where CORBA alone gives very little process and requires expert design. It is for these reasons that the Framework is based upon EJB persistence.

Experience with this project shows that EJB is a good solution to a specific problem: object-based access to enterprise-wide databases. EJB allows programmers to export object interfaces to legacy relational, hierarchical, and other databases. However, EJB is not the solution for all distributed applications. Since the framework is geared towards highly-collaborative, high-speed interaction, EJB brings unneeded complexity and layers. The Framework really only needs EJB for a persistence scheme since Properties handle security, data updates, locking, and transactions. Therefore, EJB is not well suited for real-time, groupware applications. It is a good solution to corporate database access, but it is not a skeleton key that unlocks all distributed problems.

4.3.6. N-Tiered Systems

The server is built upon an n-tiered model. While the event system can traverse over several layers for message delivery, the model is inherently hierarchical. A root server must hold all data, even if second or third level servers cache that data for faster client access.

Further, the server is built upon several layers, as displayed in Figure 4-1. If any layer experiences problems, the entire system is affected. This problem exists in traditional n-tiered systems: layers are intricately and statically dependent upon one another. *It is a house of cards that must be in perfect order for the system to function.* While this perfect order was relatively easy to ensure in local environments, it is rarely the case in distributed, Internet-based applications. A more dynamic architecture better meets the needs of distributed groupware. Future versions of the server will most likely be based on Jini (Sun Microsystems, 2000d) or some other dynamic system.

CHAPTER 5: THE DEVELOPMENT PROCESS

The second research question presented in Chapter 1 concerns whether the process of developing applications within the Framework is faster than developing applications using traditional methods. Increased speed of developing robust, scalable, and efficient real-time applications is the primary goal of this research. This section presents and then compares a traditional process with the Collaborative Server process using a simulated example application and two case studies.

5.1. A Traditional Process

A definition of the traditional process must be specified for a complete comparison. A subset of the Waterfall (Hoffer, 1999) method will be analyzed as an example of the traditional approach. The Waterfall method is used because most other methods are enhancements or changes to this foundation paradigm. For example, the Rapid Application Development method utilizes many "mini-waterfall" iterations. Specifically, the design and code phases are addressed in this chapter, as shown in Figure 5.1. The other phases, Analyze, Test, Deploy, and Maintain, are similar between traditional and Collaborative Server programming methods.

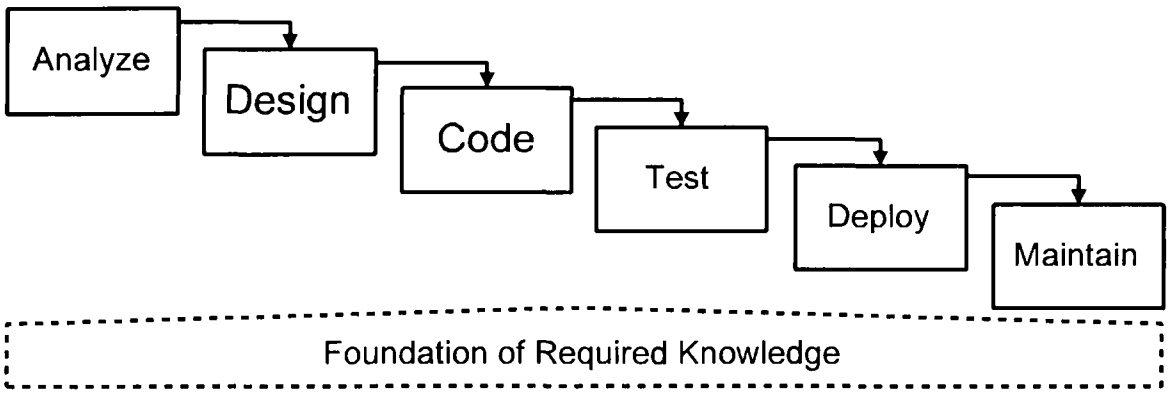

FIGURE 5.1, Waterfall Development Model

It is assumed that the project has already been adopted and the scope is the same in both methods. Since early CMI distributed applications were developed on a Windows-based, thin server, application development within this server will be used as the benchmark, traditional process. This provides a comparable benchmark since the same types of collaborative applications have been built on both servers.

5.1.1. Foundation of Required Knowledge

As stated in the Chapter 1, distributed, real-time programming requires significant knowledge and expertise. The development process is built upon this foundation of knowledge. The following sections present the knowledge required to program Java-based, collaborative programs under a thin server using traditional methods.

5.1.1.1. Object-Oriented Theory

Java is highly object-oriented (OO). Developers must understand OO and its principles if they are to program in this language.

5.1.1.2. Network Design Principles

Programmers must understand the basic principles involved in networked applications. This includes client/server architectures, n-tiered models, and distributed paradigms.

5.1.1.3. Messaging API

The programming interface of a third-party or custom messaging system must be understood to be used effectively. This includes subscription as well as publish and response methods. Timing delays and sequence errors involved in distributed, asynchronous messaging must also be included in this foundation.

5.1.1.4. Network Protocols

Since low-level socket connections are normally used to connect to a central server, the basic protocols of TCP/IP must be understood, including related protocols such as FTP, HTTP, and SMTP. The advantages and disadvantages of each will need to be used at design time for a robust system architecture. For example, HTTP will most likely need to be used to tunnel through firewalls. Since HTTP is a request/response protocol (W3C, 2000) that only clients can initiate, workarounds and standard solutions should be understood by programmers so they can be used effectively.

5.1.1.5. SQL

The common data access mechanism today for persistence (and in the Windows-based, thin server used here) is SQL. The SQL language as well as how object-oriented data structures map to/from relational schemes must be understood.

5.1.1.6. Java Language

Since Java is used as the primary application language, programmers must understand its semantics and paradigms. For example, programmers should be very comfortable with Java's GUI component sets, collections classes, and networking packages.

5.1.1.7. Security

Best-practices security paradigms will need to be employed explicitly within client application code. Therefore, programmers must understand principles such as sessioning and authentication.

5.1.1.8. Summary

Programmers who develop distributed systems using traditional approaches must have a mature understanding of the listed principles. These principles are included because they directly affect the speed at which distributed programs can be created. It should be evident from the above list that significant and expert knowledge in many areas of computer programming must be understood to develop distributed applications in Java.

5.1.2. Design

The traditional and Collaborative Server development methods differ significantly in their design phase activities. The steps required by the traditional approach are presented in the following sections.

5.1.2.1. Data Structures

One of the first tasks of design is to determine the object classes that represent data. What data will be recorded? What are the required fields in each class? What are the field types? What are the relationships between different sets of data?

Once data classes are defined, their relationships must be determined, including reference types between each class. Are data objects to be arranged in trees, linked-lists, hash tables, or some other data structure? How will data items be referenced and searched for?

5.1.2.2. Database Schema

After data objects and their relationships are defined, their persistence schema in the database must be determined. What tables are required? What normal form is needed for the specific application? What keys and foreign keys should be used? Data flow and other diagrams usually help specify the format and layout of application schema.

5.1.2.3. Business Logic

An application's business logic is the heart of its functionality. Business logic determines the course of actions when certain types of data are entered, when buttons are pressed, and when users otherwise interact with the application.

5.1.2.4. User Interface Design

The client-side user interfaces must also be designed, including screen shots and/or prototypes for main frames, toolbars, and menus. This step also includes connections between different windows, and the rules for their use.

5.1.2.5. Users and Security

Collaborative applications nearly always have concepts of users, roles, and security. Without Framework support, user data objects and different application roles (such as facilitator, editor, reader, etc.) must be defined with each different application. Further, the security rights and permissions for each role need to be designed and applied to *each* data class and user interface window. Questions to be answered in this step include: What data should be protected? How will locking be stored, and at what level will it be kept? How will user rights be stored in the database (i.e. in separate tables or only in separate fields of each tuple)?

5.1.2.6. Client/Server Protocol

While the third-party messaging system normally relays events passed throughout the system, application designers must define what data the events will contain and how interfaces will respond to these events. Normally, a standard set of constants or different event objects (both serving the purpose of differentiating between message types) must be determined. Further, the protocol for client communication with the server should be standardized across application nodes.

5.1.3. Code

The coding phase of the two development models also differs significantly. The following sections present the steps required when programmers use a traditional model.

5.1.3.1. Implement the Database

The database schema that was developed in the design phase should be implemented in an efficient manner. Normally, projects will select a third-party relational database and implement the schema into tables and relationships.

5.1.3.2. Data Access Layer

Data access routines must be developed. For example INSERT, DELETE, and SELECT statements must be written to access data from within the application. Programmers normally separate these routines into a separate layer so SQL statements are not embedded directly with application logic. Code must be created for every class, user interface, and logic routine within the application.

5.1.3.3. Users and Security

The user and security schemes from the design phase must be implemented into classes and objects. While user objects are normally straight-forward, security issues pose significant programming challenges. A very tight and secure data access system must be implemented in the data access layer as well as the code. The locking system must also be enforced at every level within application classes.

5.1.3.4. Data Objects

Object-oriented systems normally develop object equivalents for database entities. The data access layer uses SELECT statements to query the database for required data. The layer then creates in-memory objects to encapsulate this data within application classes, allowing application code to work directly with objects rather than with SQL result sets. To

make this possible, object representations of data must be created during this phase of the development.

5.1.3.5. User Interface

Programmers must develop user interfaces, generally through the use of 4th-generation, visual development environments. Once user interface skeletons are created, programmers code links between data objects and classes to display application data on the screen.

5.1.3.6. Startup

Distributed applications require bootstrapping routines to connect clients to a running server on another machine. These routines generally consult a lookup service, such as the RMI Registry, to locate server objects. Startup routines must also connect to the database and initiate the user interface. While many of these functions are done automatically by the Java Virtual Machine for local applications, most distributed applications are complex enough that they cannot use default routines and require custom bootstrapping code.

5.1.3.7. Replication

The distributed applications addressed in this dissertation are real-time, multi-user applications that share data. While each user controls his or her own screen and program location, data changes made to any screen must be replicated to all connected clients. Replication is normally done through one of the methods presented in Chapter 2, such as polling, shared database, or messaging. All of these methods require routines within each user interface class to inform and update client screens when data change.

5.1.3.8. Transactions

Distributed applications use transactions to ensure that client changes do not clash. While locking is an integral piece of distributed software, transactions deal with many more issues. For example, most databases support two-phase locking and rollback, allowing clients to roll back changes if any data updates fail. Further, when data changes fail, applications must be able to notify all clients of change rollbacks so all user screens remain consistent. This functionality must be coded manually within the data access layer to protect the integrity of application data.

5.1.3.9. Firewalls

The presence of firewalls in most corporate and government Intranets requires that distributed applications develop a mechanism to work within firewall limitations. The vast number and different types of firewalls that applications must run through multiply this problem significantly. All communication layers, including messaging, persistence, and replication, must be able to work within client-side and server-side firewalls.

5.1.3.10. Administration

Since distributed applications are shared among many users, facilitators must have access to administrative functions to perform actions such as adding and deleting users, creating meeting documents, and setting replication options. Administration screens are normally separate from user-oriented screens, but are an integral part of the overall architecture of distributed applications.

5.1.3.11. Presence

Today's distributed applications often work over large distances. Therefore, applications must provide tools that create a sense of presence among users. Functions such as telecursors, chat windows, and e-mail capabilities should be programmed into any distributed application.

While some differences do exist in the remaining phases of development, they are not relevant to this analysis. For example, while the testing phase takes less time when fewer lines of code are present, the procedure is the same for both methods. Additionally, the Maintenance phase certainly differs between the two programming models because resulting code is very different. However, the second research question of this dissertation concerns whether the Collaborative Server process speeds time to deployment, and the maintenance phase occurs after deployment.

5.2. The Collaborative Server Process

The development process (specifically the design and code phases) used with the Collaborative Server is presented in the following paragraphs. It should be compared to the traditional process given in previous sections.

5.2.1. Foundation of Required Programmer Knowledge

The following sections describe the knowledge needed to program within the Collaborative Server Framework.

5.2.1.1. Framework API

Since this type of development is built upon the Collaborative Server, programmers must understand its API and class hierarchy. However, only a subset of the classes included in the Framework need to be understood. These include the following:

- *Property*: The Property class is the heart of the Collaborative Server and represents the gateway for data access, replication, locking, and security. Also required are the child indexing classes such as SequentialIndexKey and NameIndexKey.

- *AbstractViewer*: Since all user-interface classes extend AbstractViewer, its methods must be understood. Specifically, programmers should be familiar with the four event-related methods discussed in Chapter 4.

- *Event Classes*: Programmers must understand the three classes of the event system: AddChildEvent, RemoveChildEvent, and ValueChangeEvent.

- Minimal knowledge of other classes, such as GUID, Utility, ClientDirector, and various exceptions are required. However, this knowledge is limited to one or two methods. Therefore, these classes are not listed here.

5.2.1.2. Object-Oriented Theory

As with the traditional process, the Framework is highly object-oriented. In fact, it requires a deeper understanding of the principles of polymorphism and encapsulation than the traditional process does.

5.2.1.3. Java

Since the Collaborative Server and applications built upon it are written in Java, understanding of language semantics is required. Further, programmers should be familiar with Java's AWT and/or Swing component sets and collections classes.

Note that knowledge of network design principles, messaging APIs, network protocols, and security are *not* required for development within the Framework. However, in real-world situations, understanding of these principles will aid in development because developers will understand the capabilities and limitations of the Framework better.

5.2.2. Design

Distributed application design within the Framework requires significantly less steps than the traditional method does. These steps include data object definition, business logic specification, and user interface design.

5.2.2.1. Data Object Definition

Object-oriented representations of data need to be created for each application. Note that these data structures do *not* include structure classes such as hash tables or linked lists (the Framework provides this support). These data objects are base classes such as Node, Comment, etc., that represent basic data elements the application must capture.

5.2.2.2. Business Logic Specification

As with the traditional process, business logic encapsulates the base functionality of any application. Since it is application-specific, it must be designed at this phase.

5.2.2.3. User Interface Design

Similar to the traditional process, client-side user interfaces must be designed per application.

Note that these areas of design–data objects, business logic, and user interface–are integral, application-specific areas that developers should focus on. All collaborative functionality is encapsulated within the Framework.

5.2.3. Code

The following sections describe the required coding phase steps when building upon the Framework architecture.

5.2.3.1. Create Data Objects

The data objects that were defined during the design phase must be implemented into object classes. Data object classes usually include sets of application data and accessor/modifier methods. They are usually relatively simple classes containing very little program logic.

5.2.3.2. User Interfaces

As shown in Chapter 4, Framework user interfaces extend the AbstractViewer abstract class. Programmers typically use a 4th-generation, visual development environment to create class skeletons that include all of the buttons, lists, fields, and other components

required for each interface. Programmers then modify these generated classes to include the Framework-specific methods of `activate()`, `setEditable()`, and each event destination method. Business logic might be embedded into user interfaces at this step or it might be programmed into a different layer, depending upon application needs.

Note again that the coding steps within the Framework architecture include only application-specific steps. Letting programmers spend time and resources on collaborative needs (as the traditional process does) distracts them from the specific needs of the application being created.

5.3. A Simulated Example

The differences between the traditional and Framework development processes is best illustrated with a simulated example. A chat application was developed using both processes for this purpose. This application contains relatively simple collaboration in that it replicates comments among all users connected to a common server. Replication in this application is kept to line-by-line collaboration. User directories, security, and locking are not issues in this simple example, so even the traditional process is relatively simple. More real-world examples are presented later in this chapter.

The following figure illustrates the user interface of the chat application.

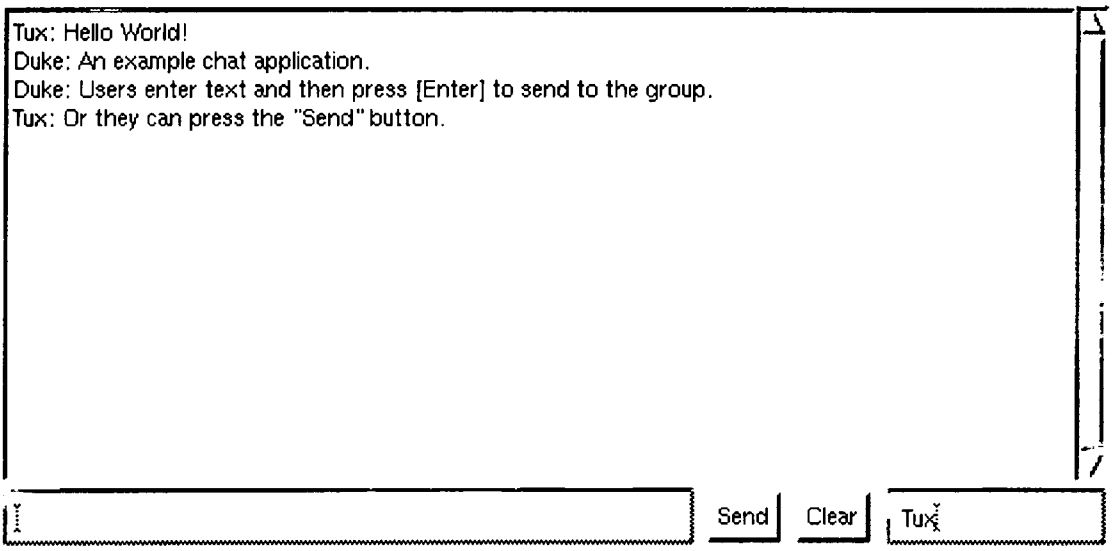

FIGURE 5.2, Example Chat Application

The following sections walk through the development process of this application using both the traditional Waterfall method and the Framework method.

5.3.1. The Traditional Approach

In order for the traditional approach and the Collaborative Server approach to be comparable, a very robust traditional application has been developed. Since the design and code phases differ between the two approaches, these phases are described.

5.3.1.1. Design

The chat application requires the following components: a server daemon, a chat client interface, and a messaging layer. The server daemon waits for incoming connections and maintains a list of current clients. When clients post messages, the server broadcasts these strings to all connected clients. For simplicity, clients connect to the server using

standard sockets and stream XML for messaging. When clients receive broadcasted XML messages, they parse the sender's name and message and display this information on their screens.

Clients initially connect to a waiting server daemon by submitting a connect packet to the server's port. The client and server then set up input and output streams for writing and reading across the network. Clients begin pinging the server with the following XML string every 20 minutes:

```
<ping/>
```

Periodic pinging allows the server to maintain a clean list of current clients, even if clients do not log off cleanly (i.e. a client crashes). If a client fails to ping at least twice in 60 minutes, the server removes the client from its current list.

When the server receives a client connection, it spawns a new thread and opens a port to handle communication with that client. This thread waits infinitely for client XML which brings chat and/or logoff messages. This thread is stopped when the server removes the client from its current list.

Clients submit chat messages when users click the "Send" button or press the [Enter] key. The XML string sent to the server is as follows:

```
<msg>
   <name>Username</name>
   <msg>Chat message here</msg>
</msg>
```

The server replicates this string to all connected clients, who in turn parse the XML and display the message on their screens. The submitting client is no different than other clients; it receives the message from the server, parses it, and shows the text on its screen.

When a client application is closed cleanly, the server thread notices the socket has been closed, and the client is removed from the server's current list. The closed client receives no more broadcasts from the server.

5.3.1.2. Code

During the coding phase of development, the server daemon and client application are implemented. The finished server code contains the following classes:

- `ChatServer.class` (117 LOC): The server daemon that listens for client connections.

- `Listener.class` (147 LOC): The thread class that is created for each incoming client.

- `TimedHashtable.class`, `TimedHashtableListener.class`, and `TimedHashtableObject.class` (184 LOC): Hold client connections for 60 minutes beyond their last ping. Clients are automatically removed and cleaned up if valid pings are not received.

The server code took 5 hours to develop and required significant testing because of networking and thread differences among operating systems and platforms. These differences made keeping the server's current client list clean very difficult and bug-prone. Development of the TimedHashtable classes and the Listener class involved significant threading. Thread programming is well supported by Java, but it still requires testing and tuning to ensure efficient and robust performance.

The client user interface was first developed without regard to networking issues. Text fields and buttons were simply placed on the form and positioned accordingly using Java's layout manager classes. This process took less than 1 hour because a visual development environment was used. Once the user interface met design requirements, bootstrapping, networking, and disconnecting code was added. The finished client code contains the following classes:

- `ChatClient.class`, `ChatClient$Pinger.class`, and `ChatClient$Waiter.class` (580 LOC): Contains user interface, networking,

and pinging code. ChatClient is the main class that communicates with the server.

- `XMLElement.class` and `XMLParseException.class` (1071 LOC): XML parsing code for messages relayed from the server. Since these classes were created before this exercise, they should not be considered in the final analysis. The only time required was understanding the class' API.

The client code took 7 hours to implement and test. 2 additional hours were required to make unforeseen changes to allow for multiple clients to connect to the server. Again, networking issues and the use of threading in the Pinger and Waiter classes complicated development and took most of the time required.

The finished chat application developed using traditional methods took 14 total hours and comprises 1028 lines of code (not including the XML code). The process required significant testing and modification.

5.3.2. Collaborative Server Chat Application

The following sections describe the design and code phases of development within the Collaborative Server architecture.

5.3.2.1. Design

The Framework-based chat application begins with the development of data objects to represent application data. For comparability reasons between the two versions in this

example, XML will be used to pass chat messages among clients in this version as well. The XML data type definitions are the same as previously discussed in the first version. These XML strings will be kept as the values of Property objects in the hierarchy, as shown in the following diagram:

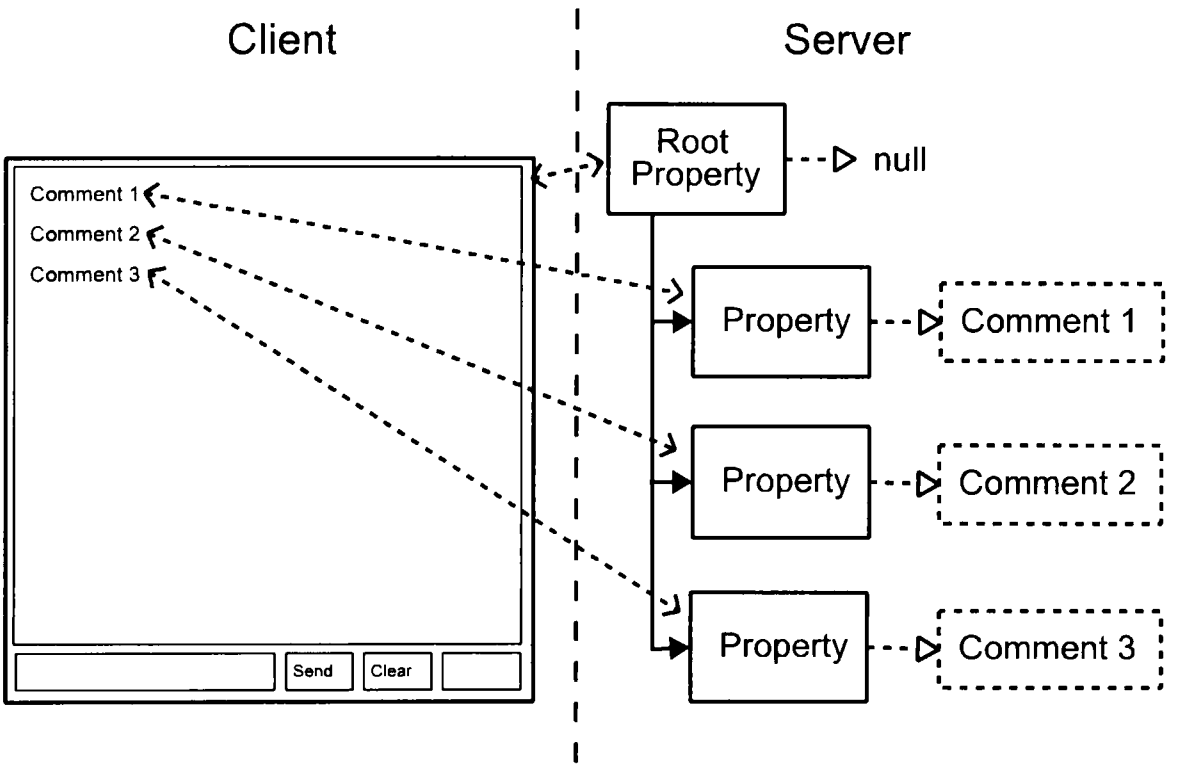

FIGURE 5.3, Framework-based Chat Application Design

As illustrated on the "Server" side of the diagram, the Property hierarchy is relatively simple: each comment is a Property child of the Root Property, and the child's value is the XML-formatted comment. These values can be seen in the boxes labeled "Comment 1",

"Comment 2", and "Comment 3". The Root Property does not require a value for this application.

The Framework automatically bootstraps each new client and makes necessary connections to the server. After setup is complete, the Framework calls the client's `activate()` method, which queries the Root Property (this connection was made automatically during bootstrapping) for its children. It parses the XML stored in each child's value and displays all previous comments to the screen.

When a user types a comment and presses the "Send" button, the client creates a new Property on the server and sets the Property's value to the message XML. The new Property is added as a child of the Root Property. The Framework automatically sends an `AddChildEvent` to each connected client, triggering each client to display the new comment.

5.3.2.2. Code

The Framework-based chat application requires only one class, `ChatViewer.class`. The first step in implementing this class is the development of the user interface, including text fields, buttons, and the output window. This step yields the exact same initial code skeleton as the traditional method, and again took less than 1 hour. The same visual development environment was used for component creation.

Several methods are then added to the client that encapsulate the business logic of the chat application. These include the following:

- `activate()`: This method uses a for loop to enumerate across the existing children of the Root Property and display all previous comments in the output window. If no previous comments were desired (as was the case in the traditional version), this method would contain no code.

- `setEditable()`: This method allows the Framework to set whether client screens are editable. Since this behavior is not used in this application, the method contains no code.

- `addChild()`: This method is called by the Framework's messaging layer when new children are added by any client. This method is coded to access the new child Property and display its comment value in the output window.

The finished chat application developed using the Collaborative Server took 1 hour and comprises 340 lines of code (note that most of this code is user interface code that was automatically generated by the IDE). Collaboration-specific code, such as setting Property values and responding to AddChildEvents, comprises less than 30 total lines of code. Very little testing was required because complex issues such as networking and threading are handled by the Framework. This version of the chat client gains additional benefits not realized in the traditional version. These benefits include the following:

- The application is able to run in applet or application mode, with the application and applet containers being provided automatically by the Framework.

- The application works through most client-side and server-side firewalls.

- The application contains optimized Framework downloads, enabling the chat session to begin more quickly with applet use.

- Although not explicitly viewed in the user interface, the application contains inherent security. If editing were enabled, users could only edit those comments they submitted–not others' comments.

- The Framework version saves comments automatically in the Property hierarchy, allowing chat history viewing and printing. For example, when a client connects to the Framework version, all previous comments can be shown in his or her output window. The traditional version keeps no such history.

A final note in this example is that because chat windows are basic to most collaborative applications, the Framework comes with a ready-made, extensible chat viewer

(which is more feature-rich than this simple example). Other components in the basic Framework download include a standard session/meeting/document manager, a user manager, a collaborative tree view, a debugger for the Property hierarchy, and a collaborative text viewer.

5.4. Case Studies

The following sections present two case studies of actual prototype development using a traditional server and using the Collaborative Server. While these applications were created as part of other research projects, every effort has been made to make the two applications comparable between the methodologies. The applications show the Framework's use in more complex, realistic applications.

CoReview and ColD SPA were originally created in Java using a Windows-based server for persistence and replication. The Windows server was developed in Delphi and was programmed to be thin and light. It provided minimal capabilities to programmers; most collaborative logic resided in the client. Therefore, the Windows server provides an example of distributed applications developed using traditional methods. The architecture of this system is as follows:

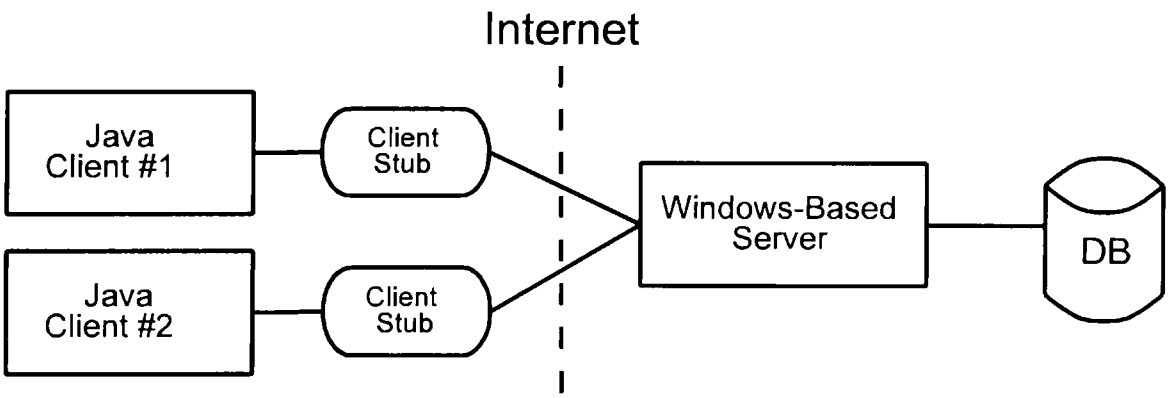

FIGURE 5.4, Windows-Based Server Architecture

The Windows server has three basic purposes. First, it manages a database connection to a relational database. Clients submit SQL calls to the server, which are then passed to the database engine. The server returns the result sets to the client stub, which package results and return them to Java clients. Second, the server broadcasts client messages to other connected clients. For example, when a client changes data in the database, it sends a broadcast message to the server, which notifies all connected clients of the change. Third, the server maintains a list of locks so clients can check whether data are accessible or not.

5.4.1. CoReview

CoReview is a collaborative document inspection system (Rodgers, 1999). It started as a code review tool and was enhanced to support inspection of any text or HTML document. Since today's word processors export to HTML, most documents can be reviewed in the tool. Code reviews are a formalized process of inspecting application code for errors

(Fagan, 1976). They are normally performed manually–with paper and pen in hand. Reviewers hold a kickoff meeting where code is introduced and the process is described. They are given standard inspection checklists that hold defect categories. Each defect found must be assigned a category, such as major, pointer defect or minor, commenting violation.

Reviewers inspect the code individually and then meet once again several days later. They step through the code, section by section, and discuss defects found. The process is time and labor intensive; duplicate defects are often found and lengthy discussions are held by proponents of the code. However, the learning process is valuable and the resulting code is much more stable.

The goals of CoReview are to increase the efficiency of inspection meetings while keeping the learning and code stability factors constant. It does this by placing the code (or document) within a collaborative application. Code is reviewed online and errors are recorded by clicking the mouse. Posted defects immediately show up on all users' screens, decreasing the amount of duplicate work. Each defect (which is shown as a flag) contains its own chat window that allows reviewers to discuss defect validity. The chat window allows individual proponents to discuss at length without taking the time of others in the group.

The following figure shows two potential screens of the CoReview application: a code document and a graphic. The colored flags represent defects that have been found in the code.

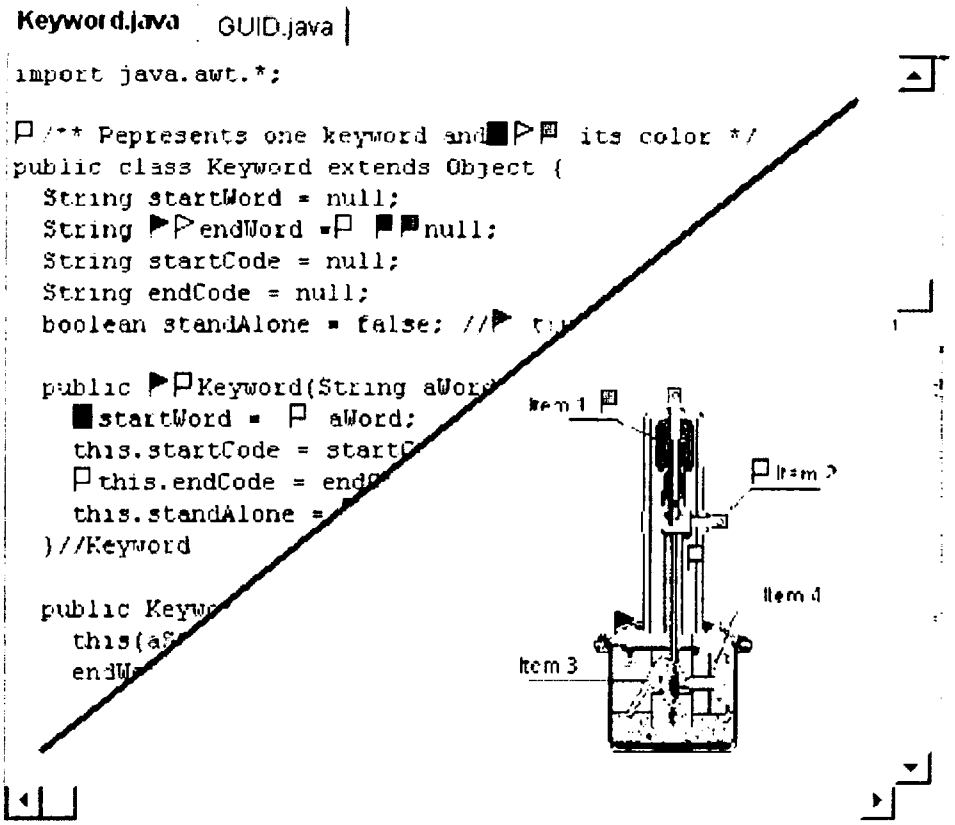

FIGURE 5.5, CoReview Code Inspection Application

5.4.1.1. Analysis

For clarity in this discussion, the version of CoReview that is built upon the Windows-server is termed CoReview 1 and the new version built upon the Framework is termed CoReview 2. Many similarities exist between the two versions. For example, the same person programmed both applications, making the coding style comparable between the two versions.

Despite their similarities, some differences between the two applications exist. CoReview 1 is solely focused on inspecting read-only documents and CoReview 2 is integrated with a collaborative editor. Further, the previous version is based in Java's AWT architecture and the current version is based in Java's new Swing architecture. These differences in user interface and in functionality have been offset by analyzing only that code which deals with collaboration-specific issues, such as persistence, replication, security, and locking. User-interface code comprises a significant amount of the code base for each application, and it is *not* included in this analysis.

Table 5.1 shows the lines of code in each version of CoReview. Note that CoReview 1 had only one security checkpoint at login. If a user successfully passes this checkpoint, he or she has full access to the system. CoReview 2 inherits the Framework's comprehensive security system without any explicit code being written. Neither version of CoReview needed locking capabilities since all work is additive (flags cannot be modified once they are posted for reasons specified by inspection theory).

	CoReview 1	**CoReview 2**	**Decrease**
Replication	1153	931	19 percent
Persistence	1208	603	51 percent

TABLE 5.1, Comparison of CoReview Lines of Code

Although differences exist between the two versions, readers might find the total lines of code informative. The major difference between CoReview 1 and CoReview 2 is

the enhanced functionality of Version 2. It has a fully-enabled, collaborative editor, including a spell checker, undo/redo, graphics capability, and viewing of historical changes made to the document. Despite these enhancements, CoReview 2 contains significantly less code. CoReview 1 comprises 24,489 lines of code and CoReview 2 comprises 8,570 lines of code, a decrease of 65 percent.

The two versions have differences in development time. CoReview 1 started in September, 1997 and finished in May, 1999, for a total development time of one year, nine months. CoReview 2 started in September, 1999 and finished in June, 2000, for a total development time of ten months. Certainly, the knowledge gained from the first version increased the efficiency of the development of the second version. However, CoReview 2 is a full rewrite and borrows almost no code from CoReview 1. In fact, due to the differences in development methodologies, very few algorithms are common between the two applications.

5.4.2. ColD SPA

Collaborative, Distributed Scenario and Process Analyzer (ColD SPA) is a collaborative scenario and process modeling tool. A traditional approach to business process modeling is for outside consultants to analyze a business (through interviews, documentation, and other methods) and create business models (Hoffer, 1999). ColD SPA takes a different approach; it allows on-the-line employees to collaboratively document the processes they follow each day. Subject matter experts (usually the employees and other people involved) meet in a group support systems room, which contains a computer for each person.

ColD SPA allows each person to contribute to the process tree and to document each node on that tree. Information such as node descriptions; involved actors; graphical time lines, junctions, and decision points; and cost metrics are recorded. The tool documents and displays each comment and addition to all connected users.

The following diagram shows a typical ColD SPA window. The application depicts the process of receiving a new customer; the window's current focus is the graphical process flow panel.

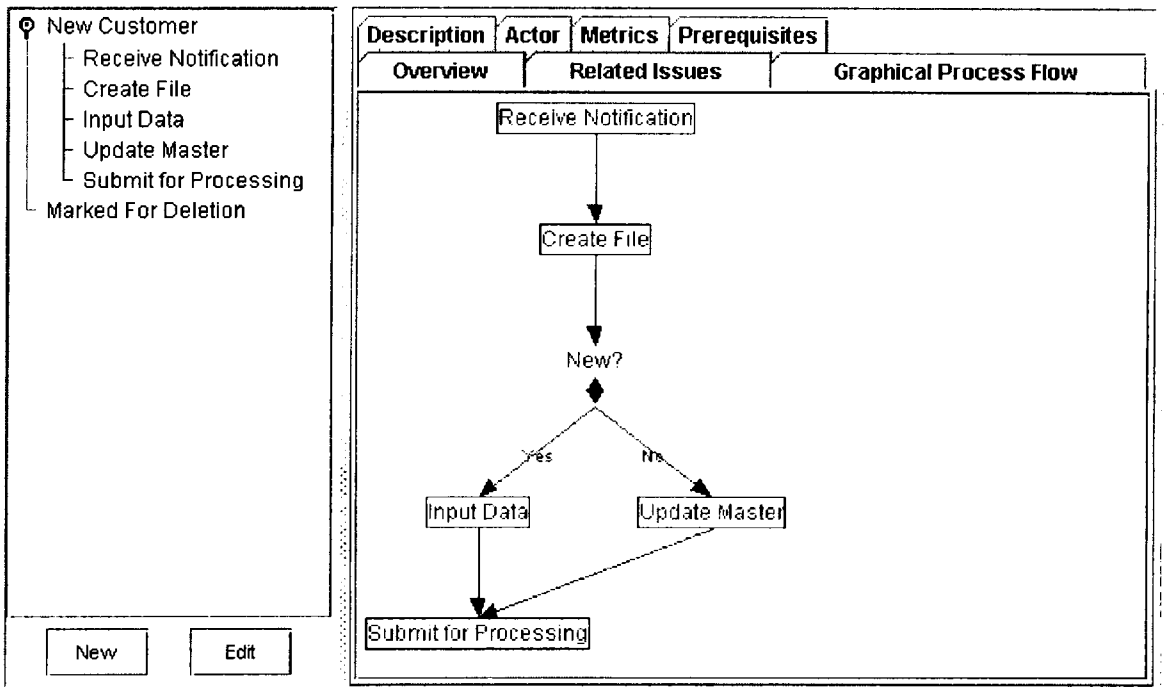

FIGURE 5.6, Collaborative, Distributed Scenario and Process Analyzer

5.4.2.1. Analysis

The ColD SPA version built upon the Windows server is termed ColD SPA 1 for clarity in analysis. The version built upon the Collaborative Server is termed ColD SPA 2. The two versions are very similar to each other in functionality. Each has a collaborative tree on the left side of the window, and a set of panels on the right. The facilitator chooses which panels to display from over 20 available panels. When the facilitator submits his or her interface changes, all user screens change to show only the selected panels. Both version are able to print and export application data to a browser window.

Security and locking are integral to both versions of ColD SPA. However, ColD SPA 1's security model is client-based. Client code checks user rights and does not allow access to the server unless the correct rights are presented. The Windows server does no such checking. Therefore, rouge programs can easily communicate with the server and access all data. Locking is also achieved through client-side code. In contrast, ColD SPA 2 has no explicit security code, but inherits the Framework's server-side security and locking systems.

Both applications use Java's Swing architecture for their user interface components. However, ColD SPA 1 uses Swing's standard tree component, and ColD SPA 2 uses a custom, collaborative tree component. The custom tree component adds 1,349 LOC to ColD SPA 2's code base that are not present in the original version.

The following table shows an analysis of the lines of code dedicated to collaboration in each version of ColD SPA.

	ColD SPA 1	ColD SPA 2	Decrease
Replication	2229	891	60 percent
Persistence	4350	579	87 percent
Locking	250	0	100 percent

TABLE 5.2, Comparison of ColD SPA Lines of Code

ColD SPA 1 comprises a total of 39,010 lines of code and ColD SPA 2 comprises a total of 8,857 lines of code, a decrease of 77 percent between the two versions. The design and coding phases of ColD SPA 1 and ColD SPA 2 took 21 and 8 months, respectively, a decrease of 61 percent. Certainly, one cause of these decreases is differences in the programming styles used in each version. However, the significant decrease is largely due to ColD SPA 2 being based upon the Framework architecture.

5.4.3. Case Analysis

Application versions built upon the Framework are significantly smaller than their traditional siblings, by 65 and 77 percent. A more detailed analysis of replication, persistence, and locking provides more insight into these differences. *Replication* code is that code devoted to keeping client screens in sync with one another. This code showed the smallest decrease in LOC at 19 percent and 60 percent. While still a decrease, the Windows server "broadcast" command takes only one line of code to send and only a few lines of code to receive. By comparison, the Framework automatically broadcasts all changes, but clients must still receive change events and update their screens.

Persistence code is that code devoted to saving data changes to the server. This type of code showed a decrease of 51 and 87 percent. This difference exists because updating the Windows server normally requires several SQL INSERT or UPDATE statements (complex objects update several relations). Further, security must be checked by client code in the Windows server architecture. In contrast, the Collaborative Server requires only one `setValue()` call. Security is automatically checked by the server.

Locking refers to locking data on the server while a client edits that data. Locking must be explicitly managed on the client side in the Windows server. This code becomes complex because cleanup must be done for clients that lock data and then fail to unlock that data. Since this functionality is provided by the Collaborative Server, no code is needed for explicit locking.

Why did ColD SPA show more improvement on the Collaborative Server than CoReview? One possible reason is style difference in programmers between the two applications. Another reason is the inherent differences between the two programs: ColD SPA is a more complex application than CoReview is. ColD SPA requires significant locking and security, and it contains more data than CoReview. Therefore, the differences between the Windows server and the Collaborative Server are more pronounced in ColD SPA.

A final note is a comparative difference between the application versions. The *nature* of replication, persistence, and locking code between the Windows server and Collaborative Server are significantly different. Code based on the Windows server contains many SQL

statements. Programmers must know both the object-oriented and the relational models to program in this environment. Further, programmers must convert data between object representation and relational representation any time they communicate with the server. This conversion is often complex and potentially buggy. The Collaborative Server is fully object-oriented and requires no embedding of SQL calls. Further, the Windows server requires programmers to be fluent in network complexities, threading issues, synchronous/asynchronous behavior, and many other behaviors that are automatically embedded within the Collaborative Server.

CHAPTER 6: CONCLUSION

The research of this dissertation has followed the Systems Development research methodology to create a new type of collaborative programming framework. This framework, based upon a Property-driven server, significantly decreases the time required to create real-time, highly-collaborative applications.

6.1. Response to Research Questions

The first research question contains two parts. First, *"can a generalized server abstract common collaborative behavior without becoming application specific?"* The CMI Collaborative Server stands as the response to this question. Its successful implementation show that a server of this type can be created. The second part of the question is: *"what are the required features of such a server?"* This question is answered in the development goals of the server (presented in Chapter 1) and in the server description (presented in Chapter 4). In summary, these requirements include a common data representation format (the Property hierarchy), an abstract client user interface class, a custom messaging system, a security system and user directory, and a set of client- and server-side directors to manage connections.

The second research question, *"Is the software development process using this Framework faster than the software development process using traditional methods?"* is answered in Chapter 5 with a description of the process, a simulated example, and two case studies. In all cases, the Collaborative Server showed significant improvements in development speed and code size when compared to the traditional Waterfall methodology. In addition to decreased development time, the given applications automatically inherited many needed functions, such as security, firewall ability, network and connection efficiency, and simplified coding and testing.

6.2. Contributions

The major contribution of this research is the development and validation of a programming framework and related server to support the rapid development of real-time, distributed, collaborative applications. The tested applications showed decreases of at least 80 percent in collaboration-related code. They also show significant decreases in the amount of time needed to code and test these applications.

Several areas of the server architecture present new ways to model data and program within the object-oriented methodology. First, the Property hierarchy is a significant departure from traditional collaborative application design. This hierarchy is abstract to any product-space and can be reused without modification for almost all collaborative applications. It automatically provides security, locking, messaging and replication, persistence, and viewing to all application data. It is the basis for the realized gains in productivity and stability.

The messaging system modifies traditional event theory in that events are somewhat aware of their destinations. Rather than programming new interfaces for each event consumer class, programmers can create custom events that contain behavior code to be run at event destinations. The result is a decrease in the number of classes and the amount of bandwidth required for the messaging layer.

The injected client proxy stub is also a new development. Client viewers essentially have two stubs: one for the Framework and one for RMI networking. The Framework stub automatically manages sessioning, security, and user identification. It also increases efficiency by short-circuiting unneeded calls to the server. It simplifies remote calls and enables programmers to focus on application user interfaces and business logic.

6.3. Limitations

Several limitations need to be discussed so readers understand the conditions the server was developed within. A major factor that influenced development was the young age of Java and Enterprise Javabeans when compared with other languages and distributed technologies. Java was 4-5 years old during the development period of the server. While the language itself changed little during this period, several standard extensions to the language were evolving. Java's Swing architecture was not finalized, and therefore, was not permanently serializable as data objects in the Property hierarchy. The Java Messaging Service, which defines a standard publish-subscribe set of messaging interfaces for third-party companies to develop with, was also not available. In retrospect, however, the development of a custom messaging service designed specifically for the Property structure

worked better than any standard messaging service could have. Third, the Enterprise Javabeans (EJB) specification was released halfway through the development of the server, and very few, if any, commercial products met this specification. No "best practices" or common patterns for deployment within EJB were available to follow. Again in retrospect, EJB was not a good decision because it meets the needs of a different business problem. Real-time, collaborative applications require devoted servers such as the Framework alone rather than one built upon EJB.

The application space in which the server has been tested is also a limitation. These applications, such as CoReview and ColD SPA, are research-based and have not been commercially released. They have not been tested for scalability beyond 100 users. Further, student programmers and not professional, mature developers were used to develop these applications. While the server itself has never crashed and was developed to a very high standard, these applications are not as fault-tolerant and are only prototypes.

In addition, every effort was made to ensure the traditional and Collaborative Server environments used for comparison in Chapter 5 were compatible. However, these environments were not as comparable as could they could have been in a scientific laboratory. For example, the CoReview and ColD SPA were combined into one application at the end of development (although the code was separated for purposes of this dissertation) because of other research pressures.

Because of application and time pressures, detailed metrics were not kept by the student programmers of the test applications. Therefore, the analysis in Chapter 5 is limited

to simulations, examples, and derived metrics from the final code, such as lines of code and months in development.

Despite these limitations, programming within the Collaborative Server Framework has shown itself to be very efficient and much easier than traditional methods. Even novice to intermediate programmers have been able to quickly understand the Framework APIs and program effective and robust collaborative applications.

6.4. Future research

The Collaborative Server is currently in version 2.0. It is highly stable and functional. However, several development paths and research opportunities exist to bring this Framework to the next level. First, it should be moved off of EJB and onto Jini (Sun Microsystems, 2000d). The Jini architecture was just emerging when the Collaborative Server was finished, but lends itself more to collaborative applications than EJB does. Jini allows clients to find server (Framework) services in a highly robust, dynamic way. For example, if several Framework servers exists on a distributed network, Jini would allow clients to find and connect to different servers depending upon their needs.

More testing and laboratory experiments should be conducted to further verify the strengths and weaknesses of the Framework's approach to collaboration. Real-world applications also need to be developed within professional programming teams and companies. These experiments will provide insights into how mature programmers react to the alternative programming methods of the Framework.

The Framework defines a server-side set of collaborative data objects as well as messaging framework with which Properties publish information to clients. A client-side framework needs to be developed to allow local-client components to interact with one another, similar to the JavaBeans architecture and AWT event mechanism. This framework will further simplify the development of collaborative applications by automating much of the client-side user interface work. It will allow visual development environments to be created that are Framework-specific. These development environments will allow programmers to create complex, efficient, and robust applications using their mice, similar to the way today's integrated development environments allow programmers to visually create single-user applications.

Finally, while the Framework comes with several common collaborative components, such as chat windows and trees, more extensive and complex components are needed, such as shared whiteboards, telecursors, multimedia, and videoconferencing windows. These components can be arranged in the toolbar of the collaborative, integrated development environment, further supporting the visual development of applications.

APPENDIX A: OBJECT HIERARCHY

Figure A.1 shows the hierarchy of major Framework classes and interfaces. The top-level interface, Identifiable, enables all implementing classes to be assigned unique id's and be referenced by the same. These id's provide the basis for child references, PropertyHandles, and event system routes.

The Property interface, which is implemented by most of the classes on the diagram, provides a basic collaborative gateway to application data. The Viewer interface is the superclass for all client-side interaction objects that connect to server-side Properties.

Runtime RMI connections and local references are also shown with gray arrows. These connections are established by the Framework as part of its bootstrapping process.

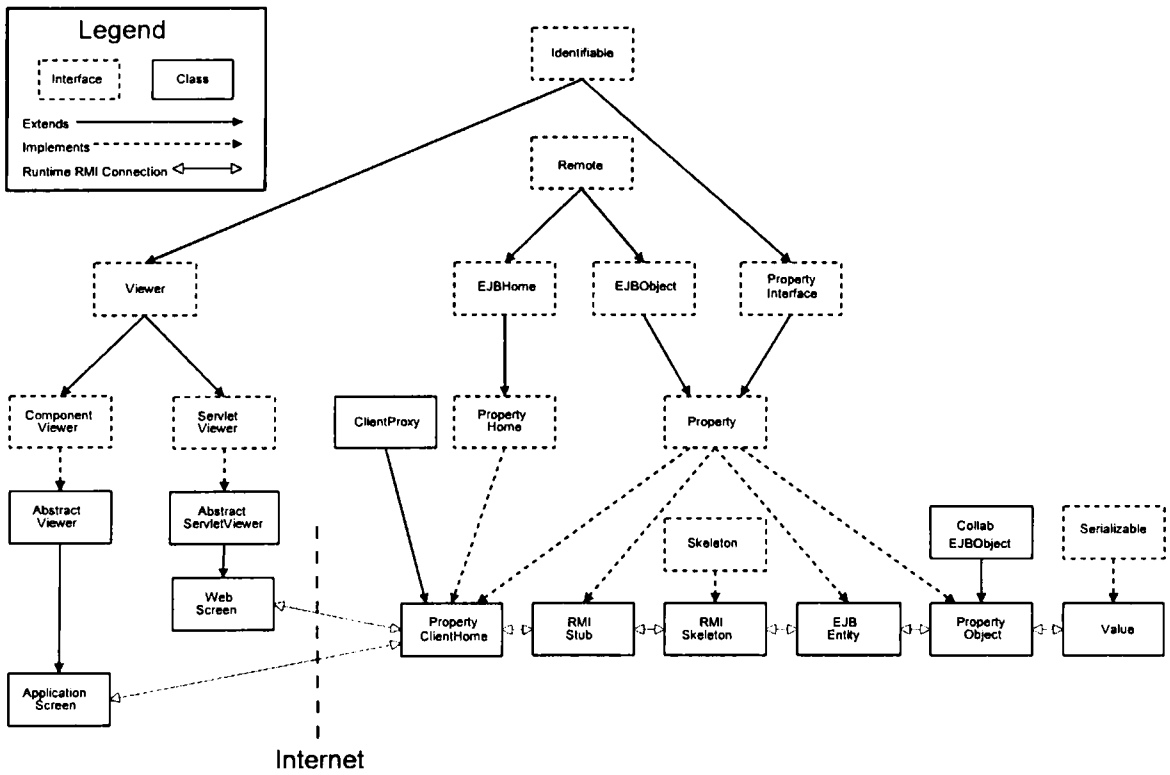

FIGURE A.1, Framework Object Hierarchy

APPENDIX B: SELECTED PROXY CODE

Client proxy objects simplify application code by handling sessioning and security (among other responsibilities). As explained in Section 4.1.2.3, the omission of client proxy objects forces programmers to use `while` loops for *every* server call. These loops increase the potential for bugs and make code more complex.

The following code shows an excerpt from the client proxy class and illustrates how the proxy encapsulates this behavior. Note that the `setValue(Serializable, Principal)` method is rarely called by client programmers. Rather, the `setValue(Serializable)` is called, which automatically injects the user's session id.

An excerpt from the Property implementation class (PropertyObject) is also presented, showing that `Property.setValue(Serializable)` simply throws an error. Whether clients use the proxy class or submit the session id explicitly, the `Property.setValue(Serializable, Principal)` class must be eventually called.

```
/** A client-side proxy stub */
public class PropertyClientHome
   implements edu.arizona.cmi.collab.Property
{
   // reference to the remote property (via an RMI stub)
   protected Property prop = null;


   /**
      Sets the value of this object using the user's login sessionID.
      The value can any type of serializable object.  Note that the size
      should be kept small because this object gets transferred across
      the wires each time it is accessed.
    */
   public void setValue (Serializable val)
      throws RemoteException, AccessControlException, LockedException
   {
      setValue(val, edu.arizona.cmi.collab.ClientDirector.getSessionID());
   }


   /**
      Sets the value of this object using an explicit sessionID.
      The value can any type of Serializable object.
      Note that the size should be kept small because this object gets
      transferred across the wires each time it is accessed.
   */
   public void setValue (Serializable val, Principal sessionID)
      throws  RemoteException, AccessControlException,
              LockedException, SessionExpiredException
   {
      boolean result = true;

      while (result) {
         try {
            prop.setValue(val, sessionID); // make the call to the server
            return; // return if successful

         } catch (edu.arizona.cmi.collab.security.SessionExpiredException ex) {
            if (CATCH_SESSION_EXPIRED_EXCEPTION) {

               javax.swing.JOptionPane.showMessageDialog(
                  edu.arizona.cmi.collab.client.ClientApp.getMainFrame(),
                  "Your session has expired.  Please login again to continue \
                  the current action.",
                  "Session Expired",
                  javax.swing.JOptionPane.INFORMATION_MESSAGE);

               result = edu.arizona.cmi.swing.dialog.Login.login(
                  edu.arizona.cmi.collab.client.ClientApp.getMainFrame());

               sessionID = edu.arizona.cmi.collab.ClientDirector.getSessionID();
            } else {
               throw  ex;
            }
         }
      }

      throw  new edu.arizona.cmi.collab.security.SessionExpiredException();
   }
}
```

```
/** The implementation of the Property interface. */
public class PropertyObject extends UnicastRemoteObject
   implements Property, Serializable, EventProducer
{

   /**
      Sets the value of this object using an explicit sessionID.
      The value can any type of Serializable object.
    */
   public synchronized final void setValue(Serializable val)
      throws RemoteException, AccessControlException, LockedException
   {
      throw  new AccessControlException(
         "PropertyObject.setValue(Object) was called.  Use \
          PropertyObject.setValue(Object, Principal) instead.");
   }


   /**
      Sets the value of this object using the user's login sessionID.
      The value can any type of Serializable object.
    */
   public synchronized final void setValue(
      Serializable val, Principal sessionID)
      throws  RemoteException, AccessControlException,
              LockedException, SessionExpiredException
   {

      // ensure this user has rights to set the property's value
      checkPermissions(sessionID, SET_VALUE_PERMISSION);

      // make the value change
      _value = val;

      // throw the change event to all listeners
      if (ServerDirector.getEventFactory() != null) {
         ServerDirector.getEventFactory().valueChange(this, _name, _value);
      }
   }
}
```

APPENDIX C: SELECTED FRAMEWORK JAVADOC

Framework class documentation is presented in this section, using the standard JavaDoc format. JavaDoc is a program that parses actual source code and produces technical documentation. For example, the following comment placed just before a given class method denotes that the comment is a description of the method behavior:

```
/** The following method compares ... */
```

Important classes are included in this section because they provide specific detail about the framework. The full documentation is formatted as a set of HTML documents.

Package edu.arizona.cmi.collab

public abstract class edu.arizona.cmi.collab.AbstractDirector
implements edu.arizona.cmi.collab.Director

Constructors

`public AbstractDirector ()`
Creates an AbstractDirector object and exports it for use by the RMI architecture.

Methods

`public static void setRegistryHost (String serverRef)`
Sets the URL for the java.rmi.Registry where the server objects are registered

`public static java.lang.String getRegistryHost ()`
Gets the URL for the java.rmi.Registry where the server objects are registered

`public static void setRegistryPort (String port)`
Sets the port number for the java.rmi.Registry where the server objects are registered

`public static java.lang.String getRegistryPort ()`
Gets the port number for the java.rmi.Registry where the server objects are registered

`public static void setRootPropertyName (String rootProperty)`
Sets the lookup name for the Root Property that should be used by default when a client connects to this JVM

`public static java.lang.String getRootPropertyName ()`
Gets the lookup name for the Root Property that should be used by default when a client connects to this JVM

`public edu.arizona.cmi.collab.Property getRootProperty ()`
Gets a reference to the Root Property. This method is exported in Director and allows remote objects to get the ref to this object's Root Property.

`public static edu.arizona.cmi.collab.Property getRootPropertyStatic ()`
Gets a reference to the Root Property, static method

`public static void setRootProperty (Property ph)`
Sets the reference for the Root Property

`public static void setObjectFactory (PropertyObjectFactory pof)`
Sets the PropertyObjectFactory that should be used to retrieve Property references.

`public static edu.arizona.cmi.collab.PropertyObjectFactory getObjectFactory ()`
Gets the PropertyObjectFactory that should be used to retrieve Property references.

`public static void setObjectFactoryClassname (String pofName)`
Sets the name for the PropertyObjectFactory that should be used to retrieve Property references.

`public static java.lang.String getObjectFactoryClassname ()`
Gets the name for the PropertyObjectFactory that should be used to retrieve Property references.

`public static java.lang.String getEventQueueClassname ()`
Sets the name for the EventQueue class to be used to handle asynchronous communication.

`public static void setEventQueueClassname (String eqName)`
Gets the name for the EventQueue class to be used to handle asynchronous communication.

`public edu.arizona.cmi.collab.events.EventQueue getEventQueue ()`
Gets the name for the EventQueue class to be used to handle asynchronous communication.

`public java.lang.Object getEventRoute ()`
Gets an Object that describes the routing that an event must take to reach the EventQueue local to this JVM.

`public static void setEventRoute (Object rt)`
Sets an Object that describes the routing that an event must take to reach the EventQueue local to this JVM.

`public static void setEventFactoryClassname (String classname)`
Sets the class type for the event factory

`public static java.lang.String getEventFactoryClassname ()`
Gets the class type for the event factory

`public static void setEventFactory (EventFactory factory)`
Sets the class for the event factory

`public static edu.arizona.cmi.collab.events.EventFactory getEventFactory ()`
Gets the class for the event factory

`public static edu.arizona.cmi.collab.Director getInstance ()`
Returns a reference to the single instance of the Director for this JVM.

`public static void printProperties (OutputStream os)`
Prints the properties of this Director

`public static java.lang.String[] getPropertyNames ()`
Gets all possible property names a ClientDirector knows about

`public static synchronized void setProperties (Hashtable props)`
Sets properties based upon the given hashtable

`public static synchronized void setSessionID (Principal p)`
Sets the user's Principal object (one per Director). This object is sent with all calls to the PropertyObject and is checked for security.

`public static synchronized java.security.Principal getSessionID ()`
Returns the user's Principal object

`public java.lang.Object getInstance (PropertyHandle ph)`
Finds the given object, using this object's object factory

`public class edu.arizona.cmi.collab.ClientDirector`
`extends edu.arizona.cmi.collab.AbstractDirector`
`implements edu.arizona.cmi.collab.Director`

Constructors
`protected ClientDirector ()`
Creates a new ClientDirector. This constructor is not public because it is necessary to restrict ClientDirector so that only one instance can exist at a time within a JVM. Construction of the ClientDirector object can be managed publicly by calling one of the static ClientDirector.init(...) methods.

`protected ClientDirector (Hashtable props)`
Creates a new ClientDirector and provides initial client-wide properties. This constructor is not public because it is necessary to restrict ClientDirector so that only one instance can exist at a time within a JVM. Construction of the ClientDirector object can be managed publicly by calling one of the static ClientDirector.init(...) methods.

`protected ClientDirector (boolean b)`
This next constructor is necessary for subclasses. Since the regular constructors call defaultProps() and initProps(), subclasses cannot override this behavior. Calling super(false) allows subclasses to circumvent these methods and call custom ones. This is necessary because the JVM seems to statically link static methods into the current class. Therefore, if ClientDirector calls initProps, it ALWAYS calls ClientDirector.initProps, even if a subclass has overridden the method.

Methods

`public static synchronized edu.arizona.cmi.collab.Viewer init ()`
This method is called to instantiate a ClientDirector object. The top-level UI object is returned from this method as a convenience so that a client application can connect to the collaborative framework with a call like: myClientAppPanel.add (ClientDirector.init());

`public static synchronized edu.arizona.cmi.collab.Viewer init (Hashtable props)`
This method is called to instantiate a ClientDirector object. In this version of the method client-wide properties can be set by supplying a hashtable in which the values are indexed by the property names. The top-level UI object is returned from this method as a convenience so that a client application can connect to the collaborative framework with a call like: myClientAppPanel.add (ClientDirector.init(propertyHash));

`protected static void defaultProps ()`
This method guarantees that all properties that have not been set by the client application are set to their default values.

`protected static void initProps ()`
This method initializes resources needed to be involved in the collaborative framework. In particular this method obtains references to the director on the next higher tier, links the event queues, and prepares the first Property/Viewer combination for the client app.

`public static void activate ()`
Notifies the Root Viewer that it should prepare its state for visible use.

`public static void link (Viewer view, PropertyHandle ph)`
This method creates the event routing connections that are necessary to put a Viewer and a Property in communication with each other.

`public static void link (Viewer view, Property prop)`
Links a viewer to a given Property.

`public static void unlink (Viewer view, PropertyHandle ph)`
Links a viewer to the Property pointed to by the PropertyHandle.

`public static void unlink (Viewer view, Property prop)`
Unlinks a viewer from the given Property, including removing the viewer from its message destination list.

`public static edu.arizona.cmi.collab.Viewer getRootViewer ()`
Returns a reference to the Root Viewer object.

`public static synchronized void setServerDirector (String name)`
Sets the server director

`public static java.lang.String getServerDirector ()`
Gets the server director

`public static synchronized void setRootViewerType (String type)`
Sets the Root Viewer

`public static java.lang.String getRootViewerType ()`
Gets the server director

`public static void printProperties (OutputStream os)`
Prints the properties of this ClientDirector

`public static java.lang.String[] getPropertyNames ()`
Gets all possible property names a ClientDirector knows about

`public static synchronized void setProperties (Hashtable props)`
Sets properties based upon the given hashtable

`public static void setPollDelay (String millis)`
Sets the poll delay, in milliseconds

`public static long getPollDelay ()`
Returns the poll delay, in milliseconds

`public edu.arizona.cmi.collab.Property createProperty (String type, Principal sessionID)`
Creates a new property object on the server. Programmers should normally not call this method directly, but should use the ClientPropertyFactory instead.

`public edu.arizona.cmi.collab.user.Directory getDirectory ()`
Returns a remote reference to the user Directory for the server

`public static java.lang.String getName ()`
Returns the name of this object

`public java.lang.Object generateEventQueueProxy (Principal proxyID)`
Adds a server-side event queue proxy for a client behind a firewall.

`public edu.arizona.cmi.collab.events.PropertyEvent[] popEventQueueProxy (Principal proxyID)`
Pops events off of the server event queue proxy

`public class edu.arizona.cmi.collab.ServletDirector`
`extends edu.arizona.cmi.collab.ClientDirector`
`implements edu.arizona.cmi.collab.Director`

A specialization of ClientDirector for servlet use

Constructors

`protected ServletDirector ()`
Constructor: See ClientDirector for description why this is protected.

`protected ServletDirector (Hashtable props)`
Constructor: See ClientDirector for description why this is protected.

Methods

```
public static synchronized edu.arizona.cmi.collab.Viewer init ()
```
This method is called to instantiate a ClientDirector object. The top-level UI object is returned from this method as a convenience so that a client application can connect to the collaborative framework with a call like: myClientAppPanel.add (ClientDirector.init());

```
public static synchronized edu.arizona.cmi.collab.Viewer init (Hashtable props)
```
This method is called to instantiate a ClientDirector object. In this version of the method client-wide properties can be set by supplying a hashtable in which the values are indexed by the property names. The top-level UI object is returned from this method as a convenience so that a client application can connect to the collaborative framework with a call like: myClientAppPanel.add (ClientDirector.init(propertyHash));

```
protected static void initProps ()
```
This method initializes resources needed to be involved in the collaborative framework. In particular this method obtains references to the director on the next higher tier, links the event queues, and prepares the first Property/Viewer combination for the client app.

```
public static void activate ()
```
Notifies the Root Viewer that it should prepare its state for visible use.

```
public static void link (Viewer view, Property prop)
```
Links a given property to a given viewer.

```
public static void unlink (Viewer view, Property prop)
```
Removes a messaging route from an object

```
public static java.lang.String getName ()
```
Returns the name of this object

`public class edu.arizona.cmi.collab.SequentialIndexer`
`implements java.io.Serializable, edu.arizona.cmi.collab.ChildIndexer`

A sequential index (1, 2, 3, ..., n) of the property children

Constructors

```
public SequentialIndexer ()
```
Constructor

Methods

```
public java.lang.String getValidKeyType ()
```
Returns the fully-qualified valid key type for this index

```
public edu.arizona.cmi.collab.PropertyHandle getChild (IndexKey ik)
```
Gets a child based upon the given IndexKey

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren (IndexKey[] ik)
```
Gets children based upon the given IndexKeys

```
public edu.arizona.cmi.collab.IndexKey[] getIndexKeys ()
```
Returns the index keys currently in the index

```
public void addChild (Property p, PropertyHandle ph, IndexKey ik)
```
Adds a child to this ChildIndexer. Uses the default add location if the IndexKey array is null or if this object's valid index key is not in the array

```
public void removeChild (Property p)
```
Remove all references to the given property from this index

```
public edu.arizona.cmi.collab.IndexKey getChildIndex (Property p)
```
Returns the IndexKey of the given child in this indexer

public interface edu.arizona.cmi.collab.Property
implements edu.arizona.cmi.collab.PropertyInterface, javax.ejb.EJBObject, edu.arizona.cmi.util.Identifiable

public interface edu.arizona.cmi.collab.ChildIndexer

Methods

```
public java.lang.String getValidKeyType ()
```
Returns the fully-qualified valid key type for this index

```
public edu.arizona.cmi.collab.PropertyHandle getChild (IndexKey ik)
```
Gets a child based upon the given IndexKey

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren (IndexKey[] ik)
```
Gets children based upon the given IndexKeys

```
public edu.arizona.cmi.collab.IndexKey[] getIndexKeys ()
```
Returns the index keys currently in the index

```
public void addChild (Property p, PropertyHandle ph, IndexKey ik)
```
Adds a child to this ChildIndexer. The PropertyHandle points to the Property and is included for convenience (since the parent Property must create the PropertyHandle and every addChild method use it).

```
public void removeChild (Property p)
```
Remove all references to the given property from this index

```
public edu.arizona.cmi.collab.IndexKey getChildIndex (Property p)
```
Returns the IndexKey of the given child in this indexer

public class edu.arizona.cmi.collab.SequentialIndexKey
implements edu.arizona.cmi.collab.IndexKey, java.io.Serializable

Denotes the location of a sequentially indexed child.

Constructors

```
public SequentialIndexKey ()
```
Constructor

```
public SequentialIndexKey (int index)
```
Constructor

Methods

```
public java.lang.Object getIndexObject ()
```
Returns the object denoting the index of the child

```
public java.lang.String getKeyType ()
```
Returns the fully-qualitifed name of this class. This return type should match the Indexer's getValidKeyType() method.

```
public java.lang.String getIndexerType ()
```
Returns the fully-qualified name of the related Indexer

public interface edu.arizona.cmi.collab.ComponentViewer
implements edu.arizona.cmi.collab.Viewer

Methods

```
public void activate (boolean b)
```
Sets the Viewer state for viewing

```
public java.awt.Component getViewerUI ()
```
Returns the Component that the Viewer manipulates in order to display the Property.

```
public java.awt.Component getRenderedIcon ()
```
Returns the Component that represents the Viewer.

```
public void setEditable (boolean b)
```
Prepares the Viewer's state to either permit or disallow editing the Property.

```
public void addChild (Property p)
```
Updates the UI when new child properties are added.

```
public void removeChild (GUID childID)
```
Updates the UI when new child properties are removed.

```
public void valueChange (String name, Object value)
```
Updates the UI when the value of this property changes.

public class edu.arizona.cmi.collab.PropertyHandle
implements java.io.Serializable, edu.arizona.cmi.util.Identifiable, edu.arizona.cmi.collab.events.EventProducer

Constructors

```
public PropertyHandle ()
public PropertyHandle (Identifiable i)
public PropertyHandle (GUID id)
public PropertyHandle (GUID id, Property parent)
```

Methods

```
public synchronized edu.arizona.cmi.collab.Property getInstance ()
```
Returns an actual reference to the Property that this PropertyHandle points to.

```
public void setID (GUID id)
```
Sets the id of this PropertyHandle

```
public edu.arizona.cmi.util.GUID getID ()
```
Returns the id of this PropertyHandle

```
public void setParent (Property p)
```
Sets the transient parent field. This variable is only used for efficiency and for the life of this PropertyHandle.

```
public edu.arizona.cmi.collab.Property getParent ()
```
Gets the transient parent field. This variable is only used for efficiency and for the life of this PropertyHandle.

```
public java.lang.ref.Reference getReference (Object o)

public int hashCode ()

protected edu.arizona.cmi.collab.PropertyObjectFactory getFactory ()

public java.lang.String toString ()

public boolean equals (Object o)

public void addEventRoute (Object route)
```
Notifies the producer to add this route to its list of listening consumers

```
public void removeEventRoute (Object route)
```
Notifies the producer to delete this route from its list of listening consumers

public interface edu.arizona.cmi.collab.PropertyInterface
implements edu.arizona.cmi.util.Identifiable

This is one of the most important interfaces/classes in the Framework. It defines the methods of a Property object. All access to program data goes through this interface.

Methods

```
public void setType (String type)
```
Sets the type of this object. Not used right now but included for future meta-typing.

```
public java.lang.String getType ()
```
Gets the type of this object. Not used right now but included for future meta-typing.

```
public void setName (String name)
```
Sets the name of this object. If this object's parent is a NamingPropertyObject, this child can be retrieved by name.

```
public java.lang.String getName ()
```
Gets the name of this object.

```
public void setValue (Serializable val)
```
Sets the value of this object using the user's login sessionID. The value can any type of Serializable object. Note that the size should be kept small because this object gets transferred across the wires each time it is accessed.

```
public void setValue (Serializable val, Principal sessionID)
```
Sets the value of this object using an explicit sessionID. The value can any type of Serializable object. Note that the size should be kept small because this object gets transferred across the wires each time it is accessed.

```
public java.io.Serializable getValue ()
```
Gets the value of this object using the user's login sessionID. The value can any type of Serializable object. Note that the size should be kept small because this object gets transferred across the wires each time it is accessed.

`public java.io.Serializable getValue (Principal sessionID)`
Gets the value of this object using an explicit sessionID. The value can any type of Serializable object. Note that the size should be kept small because this object gets transferred across the wires each time it is accessed.

`public void setDefault (Serializable def)`
Sets the default value of this object which is given if setValue() has not been used. The user's login sessionID is used to verify permissions.

`public void setDefault (Serializable def, Principal sessionID)`
Sets the default value of this object which is given if setValue() has not been used. The given sessionID is used to verify permissions.

`public java.io.Serializable getDefault ()`
Gets the default value of this object which is given if setValue() has not been used. The user's login sessionID is used to verify permissions.

`public java.io.Serializable getDefault (Principal sessionID)`
Gets the default value of this object which is given if setValue() has not been used. The given sessionID is used to verify permissions.

`public void changeValue (Serializable val)`
Changes the temporary value of this object, which can then be commited to be permanent using commitChange(). The user's login sessionID is used to verify permissions.

`public void changeValue (Serializable val, Principal sessionID)`
Changes the temporary value of this object, which can then be commited to be permanent using commitChange(). The given sessionID is used to verify permissions.

`public boolean changePending ()`
Returns whether a change is pending (i.e. whether changeValue() has been called without a commitChange() being called yet). The user's login sessionID is used to verify permissions.

`public boolean changePending (Principal sessionID)`
Returns whether a change is pending (i.e. whether changeValue() has been called without a commitChange() being called yet). The given sessionID is used to verify permissions.

`public void commitChange ()`
Commits any changes that have been made using changeValue(). The user's login sessionID is used to verify permissions.

`public void commitChange (Principal sessionID)`
Commits any changes that have been made using changeValue(). The given sessionID is used to verify permissions.

`public void rollbackChange ()`
Undos the request to changeValue() before a commitChange() has been called. The user's login sessionID is used to verify permissions.

`public void rollbackChange (Principal sessionID)`
Undos the request to changeValue() before a commitChange() has been called. The given sessionID is used to verify permissions.

`public boolean setLocked (boolean locked, Permission permission)`
Locks this object. The lock is good for 30 minutes, at which time (if the user has not already explicitly unlocked it) the lock is removed. If a longer lock is desired, the user must call this method again within the 30 minute time frame to aquire another 30 minute lease. During the lock period only the user's sessionID

can use the given permission, despite any rights the object's ACL gives. The user's login sessionID is used to verify permissions. The method returns whether the lock was successfully achieved. Multiple calls to setLocked will lock/unlock different permissions of the object.

`public boolean setLocked (boolean locked, Permission permission, Principal sessionID)`
Locks this object. The lock is good for 30 minutes, at which time (if the user has not already explicitly unlocked it) the lock is removed. If a longer lock is desired, the user must call this method again within the 30 minute time frame to aquire another 30 minute lease. During the lock period only the given sessionID can use the given permissions, despite any rights the object's ACL gives. The given sessionID is used to verify permissions. The method returns whether the lock was successfully achieved. Mutliple calls to setLocked will lock/unlock different permissions of the object.

`public boolean isLocked (Permission permission)`
Returns whether this object has a valid lock on it. If this user has aquired a valid lock on the object, the method still returns false since the object is not locked to this user.

`public boolean isLocked (Permission permission, Principal sessionID)`
Returns whether this object has a valid lock on it. If this user has aquired a valid lock on the object, the method still returns false since the object is not locked to this user.

`public void setViewerType (String vType)`
Sets the default viewer for this object. The viewer must be of type edu.arizona.cmi.collab.Viewer.

`public java.lang.String getViewerType ()`
Gets the default viewer for this object. The viewer must be of type edu.arizona.cmi.collab.Viewer.

`public void addEventRoute (Object route)`
Internal method to add messaging routes to this object.

`public void removeEventRoute (Object route)`
Internal method to remove messaging routes from this object.

`public void addChild (Property child)`
Adds a new child to this object. The user's login sessionID is used to verify permissions.

`public void addChild (Property child, Principal principal)`
Adds a new child to this object. The given sessionID is used to verify permissions.

`public void addChild (Property child, IndexKey ik)`
Adds a new child to this object. The user's login sessionID is used to verify permissions. The given IndexKey is used to specify the location in one child indexer.

`public void addChild (Property child, IndexKey ik, Principal sessionID)`
Adds a new child to this object. The user's login sessionID is used to verify permissions. The given IndexKey is used to specify the location in one child indexer.

`public void addChild (Property child, IndexKey[] ik)`
Adds a new child to this object. The user's login sessionID is used to verify permissions. The given IndexKey array is used to specify ChildIndexer locations

`public void addChild (Property child, IndexKey[] ik, Principal sessionID)`
Adds a new child to this object. The given sessionID is used to verify permissions. The given IndexKey array is used to specify ChildIndexer locations

`public void addChildren (Property[] children)`
Adds new children to this object. The user's login sessionID is used to verify permissions.

```
public void addChildren (Property[] children, Principal sessionID)
```
Adds new children to this object. The given sessionID is used to verify permissions.

```
public edu.arizona.cmi.collab.PropertyHandle getParent ()
```
Returns a PropertyHandle to the parent of this object. The user's login sessionID is used to verify permissions.

```
public edu.arizona.cmi.collab.PropertyHandle getParent (Principal sessionID)
```
Returns a PropertyHandle to the parent of this object. The given sessionID is used to verify permissions.

```
public void setParent (PropertyHandle ph)
```
Sets the parent of this Property. The user's login sessionID is used to verify permissions.

```
public void setParent (PropertyHandle ph, Principal sessionID)
```
Sets the parent of this Property. The given sessionID is used to verify permissions.

```
public edu.arizona.cmi.util.GUID getParentGUID ()
```
Returns the ID of this Property's parent.

```
public edu.arizona.cmi.util.GUID getParentGUID (Principal sessionID)
```
Returns the ID of this property's parent.

```
public edu.arizona.cmi.util.GUID[] getChildGUIDs ()
```
Returns the GUIDs of all of the children of this Property. The user's login sessionID is used to verify permissions.

```
public edu.arizona.cmi.util.GUID[] getChildGUIDs (Principal sessionID)
```
Returns the GUIDs of all of the children of this Property. The given sessionID is used to verify permissions.

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren ()
```
Returns PropertyHandles to all of the children of this Property. The user's sessionID is used to verify permissions.

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren (Principal sessionID)
```
Returns PropertyHandles to all of the children of this Property. The given sessionID is used to verify permissions.

```
public int getChildCount ()
```
Returns the number of children in this object. The user's login sessionID is used to verify permissions.

```
public int getChildCount (Principal sessionID)
```
Returns the number of children in this object. The given sessionID is used to verify permissions.

```
public void fireEvent (PropertyEvent pe)
```
Places a PropertyEvent on this JVM's event queue. This is an internal method used to allow for messaging.

```
public void addACLEntry (AclEntry entry)
```
Adds an ACL entry to an object. The user's login sessionID is used to verify permissions. Initially, only the user who created the object can add permissions to a Property. This user can designate other user's to have addACl rights.

```
public void addACLEntry (AclEntry entry, Principal sessionID)
```
Adds an ACL entry to an object. The given sessionID is used to verify permissions. Initially, only the user who created the object can add permissions to a Property. This user can designate other user's to have addACl rights.

```
public void removeACLEntry (AclEntry entry)
```
Removes permissions for a specific user. Steps through the permissions in the given ACLEntry and removes all priviledges in any entry of this property's ACL for the given user.

```
public void removeACLEntry (AclEntry entry, Principal sessionID)
```
Removes permissions for a specific user. Steps through the permissions in the given ACLEntry and removes all priviledges in any entry of this property's ACL for the given user.

```
public void checkPermissions (Principal sessionID, Permission permission)
```
Checks the given permission on this property, possibly throwing an AccessControlException, LockedException, or SessionExpiredException.

```
public void checkPermissionsRecurse (Principal principal, Permission permission)
```
Internal method to recurse permission checking to parents. This should not be called by clients

```
public java.security.acl.Acl getACL ()
```
Returns the entries in the ACL. The user's sessionID is used to verify permissions. Initially, only the user who created the object can add permissions to a Property. This user can designate other user's to have modifyACl rights.

```
public java.security.acl.Acl getACL (Principal sessionID)
```
Returns the entries in the ACL. The given sessionID is used to verify permissions. Initially, only the user who created the object can add permissions to a Property. This user can designate other user's to have modifyACl rights.

```
public void addChildIndexer (ChildIndexer cIndexer)
```
Adds a new ChildIndexer to index the children of this object

```
public edu.arizona.cmi.collab.IndexKey[] getIndexKeys (String indexKeyType)
```
Returns the keys of the given index type, or null if no index of this type exists in this object. The indexKeyType should be equal the return value of getValidKeyType() of the requested index.

```
public edu.arizona.cmi.collab.IndexKey[] getIndexKeys (String indexKeyType, Principal
sessionID)
```
Returns the keys of the given index type, or null if no index of this type exists in this object. The indexKeyType should be equal the return value of getValidKeyType() of the requested index.

```
public edu.arizona.cmi.collab.IndexKey getChildIndex (Property child, String indexKeyType)
```
Returns the key of the given child in the given indexer

```
public edu.arizona.cmi.collab.IndexKey getChildIndex (Property child, String indexKeyType,
Principal sessionID)
```
Returns the key of the given child in the given indexer

```
public edu.arizona.cmi.collab.PropertyHandle getChild (IndexKey ik)
```
Returns the child with a matching ChildIndex. The user's login sessionID is used to verify permissions.

```
public edu.arizona.cmi.collab.PropertyHandle getChild (IndexKey ik, Principal sessionID)
```
Returns the child with a matching ChildIndex. The given sessionID is used to verify permissions.

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren (IndexKey[] childIndices)
```
Returns select children of this Property. The user's login sessionID is used to verify permisisons.

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren (IndexKey[] childIndices,
Principal sessionID)
```
Returns select children of this Property. The given sessionID is used to verify permisisons.

`public void removeChild (IndexKey id)`
Removes children from this object. The user's login sessionID is used to verify permissions.

`public void removeChild (IndexKey id, Principal sessionID)`
Removes children from this object. The given sessionID is used to verify permissions.

`public void removeChildren (IndexKey[] childID)`
Removes children from this object. The user's login sessionID is used to verify permissions.

`public void removeChildren (IndexKey[] childID, Principal sessionID)`
Removes children from this object. The given sessionID is used to verify permissions.

`public edu.arizona.cmi.collab.Property derefChild (GUID guid)`
Dereferences a GUID to an actual child of this Property. This method is used internally to support getInstance().

`public class edu.arizona.cmi.collab.GUIDIndexer`
`extends java.util.Hashtable`
`implements edu.arizona.cmi.collab.ChildIndexer, java.io.Serializable`

Constructors

`public GUIDIndexer ()`

Methods

`public java.lang.String getValidKeyType ()`
Returns the valid key for this indexer: "GUID"

`public edu.arizona.cmi.collab.PropertyHandle getChild (IndexKey ik)`
Returns the child with the given GUID

`public edu.arizona.cmi.collab.PropertyHandle[] getChildren (IndexKey[] ik)`
Returns the children with the given GUIDs

`public edu.arizona.cmi.collab.IndexKey[] getIndexKeys ()`
Returns the index keys of the given GUIDs (the GUIDs themselves in this case)

`public void addChild (Property p, PropertyHandle ph, IndexKey ik)`
Adds a child to this ChildIndexer. Uses the default add location if the IndexKey array is null or if this object's valid index key is not in the array. The PropertyHandle points to the Property and is included for convenience (since the parent Property must create the PropertyHandle and every addChild method use it).

`public void removeChild (Property p)`
Removes the given child from this index.

`public edu.arizona.cmi.collab.IndexKey getChildIndex (Property p)`
Returns the index (GUID) of the given child.

`public abstract class edu.arizona.cmi.collab.AbstractServletViewer`
`implements edu.arizona.cmi.collab.ServletViewer,`
`javax.servlet.SingleThreadModel`

Implements much of the common behavior for ServletViewers. Implementation note: This class implements SingleThreadModel because it incorrectly assigns the requested property using setProperty(). Multiple threads using this object will clash since a class variable is used. This should be fixed in the future.

Constructors

```
public AbstractServletViewer ()
```

Methods

```
public void service (HttpServletRequest request, HttpServletResponse response)
```
Handles the service request from the browser.

```
public void setProperty (Property p)
```
Sets the Property to be viewed

```
public edu.arizona.cmi.collab.Property getProperty ()
```
Gets the Property being viewed.

```
public void setID (GUID guid)
```
Sets the ID of this viewer

```
public edu.arizona.cmi.util.GUID getID ()
```
Gets the ID of this viewer

```
protected void error (PrintWriter out, String st)
```
Reports an error

```
protected void error (PrintWriter out, String st, Exception e)
```
Reports an error, with an Exception

```
public void footer (PrintWriter out)
```
Displays a footer

```
public void login (HttpServletRequest request, HttpServletResponse response, PrintWriter
out, String username)
```
Handles user login and verification

```
public void addFormParameters (HttpServletRequest request, PrintWriter out, Principal
sessionID)
```
Prints the standard form parameters most form links should need

```
public java.lang.String addLinkParameters (HttpServletRequest request, PrintWriter out,
Principal sessionID)
```
Returns the standard link parameters most URL links should need. This method must be the first part of the link because it starts with the '?' parameter.

```
public static void javascriptConfirm (PrintWriter out)
```
Prints the JavaScript to confirm a URL. This code must work will all major browsers. It should be placed in the section of the page. The HTML code to use this in a link is as follows: {\f1\cb1\cf4 Do Something }

```
public edu.arizona.cmi.collab.user.Directory getDirectory ()
```
Returns the server's directory object

```
public abstract void service (HttpServletRequest request, HttpServletResponse response,
PrintWriter out, Principal sessionID)
```
Displays the HTML screen, starting with the tag and ending with . Subclasses introduce specific behavior with this method.

public class edu.arizona.cmi.collab.ServerDirector
extends edu.arizona.cmi.collab.AbstractDirector
implements edu.arizona.cmi.collab.Director

This class is very important because it bootstraps the server. Clients connect to the server through this class.

Constructors

public ServerDirector ()

Methods

public static synchronized void init ()
Ensures the creation of only a Singleton instance of the ServerDirector.

public static synchronized void init (Directory dir, Context context)
Ensures the creation of only a Singleton instance of the ServerDirector.

public static synchronized void init (String propFile, Directory dir, Context context)
Ensures the creation of only a Singleton instance of the ServerDirector. This version of the method accepts the name of a server property file that can be used to specify many of the properties that the ServerDirector manages.

public static synchronized void init (String propFile)
Ensures the creation of only a Singleton instance of the ServerDirector. This version of the method accepts the name of a server property file that can be used to specify many of the properties that the ServerDirector manages.
protected static void defaultProps ()
This method ensures that all properties that remain unassigned are initialized to default values

protected static void initProps ()
Initializes the resources and properties for the server tier.

public static void printProperties (OutputStream os)
Prints the properties of this ServerDirector

public static java.lang.String[] getPropertyNames ()
Gets all possible property names a ClientDirector knows about

public static synchronized void setProperties (Hashtable props)
Sets properties based upon the given hashtable

public edu.arizona.cmi.collab.Property createProperty (String type, Principal sessionID)
Creates a Property object of the given type and returns a (usually remote) reference to it.
protected edu.arizona.cmi.collab.Property createPropertyInternal (String type, PrincipalImpl p)
Internal method to help in creating properties

public edu.arizona.cmi.collab.user.Directory getDirectory ()
Returns a remote reference to the user directory

public static javax.naming.Context getContext ()
Returns the initial context

public java.lang.Object generateEventQueueProxy (Principal proxyID)
Adds a server-side event queue proxy for a client behind a firewall.

```
public edu.arizona.cmi.collab.events.PropertyEvent[] popEventQueueProxy (Principal
proxyID)
```
Pops events off of an event queue proxy

public class edu.arizona.cmi.collab.NameIndexer
implements edu.arizona.cmi.collab.ChildIndexer, java.io.Serializable

An index on the object names

Constructors

```
public NameIndexer ()
```
Constructor

Methods

```
public java.lang.String getValidKeyType ()
```
Returns the fully-qualified valid key type for this index

```
public edu.arizona.cmi.collab.PropertyHandle getChild (IndexKey ik)
```
Gets a child based upon the given IndexKey

```
public edu.arizona.cmi.collab.PropertyHandle[] getChildren (IndexKey[] ik)
```
Gets children based upon the given IndexKeys

```
public edu.arizona.cmi.collab.IndexKey[] getIndexKeys ()
```
Returns the index keys currently in the index

```
public void addChild (Property p, PropertyHandle ph, IndexKey ik)
```
Adds a child to this ChildIndexer. Uses the default add location if the IndexKey array is null or if this object's valid index key is not in the array

```
public void removeChild (Property p)
```
Remove all references to the given property from this index

```
public edu.arizona.cmi.collab.IndexKey getChildIndex (Property p)
```
Returns the IndexKey of the given child in this indexer

public interface edu.arizona.cmi.collab.IndexKey

A key used to insert and get children from properties.

Methods

```
public java.lang.Object getIndexObject ()
```
Returns the object denoting the index of the child

```
public java.lang.String getKeyType ()
```
Returns the fully-qualitifed name of this class. This return type should match the Indexer's getValidKeyType() method.

```
public java.lang.String getIndexerType ()
```
Returns the fully-qualified name of the related Indexer

public class edu.arizona.cmi.collab.NameIndexKey
implements java.io.Serializable, edu.arizona.cmi.collab.IndexKey

Denotes the location of a name indexed child.

Constructors

public NameIndexKey ()
Constructor

public NameIndexKey (String name)
Constructor

Methods

public java.lang.Object getIndexObject ()
Returns the object denoting the index of the child

public java.lang.String getKeyType ()
Returns the fully-qualitifed name of this class. This return type should match the Indexer's getValidKeyType() method.

public java.lang.String getIndexerType ()
Returns the fully-qualified name of the related Indexer

public interface edu.arizona.cmi.collab.Director
implements java.rmi.Remote
Defines the interface for AbstractDirector, ServerDirector, ClientDirector, and ServletDirector.

Methods

public edu.arizona.cmi.collab.Property getRootProperty ()
Gets a reference to the Root Property

public edu.arizona.cmi.collab.events.EventQueue getEventQueue ()
Exposes the outgoing EventQueue that exists on this tier.

public java.lang.Object getEventRoute ()
Exposes the routing information that directs events from the data-tier to this tier.

public edu.arizona.cmi.collab.Property createProperty (String type, Principal sessionID)
Creates a Property object of the given type and returns a remote reference to it

public edu.arizona.cmi.collab.user.Directory getDirectory ()
Returns a remote reference to the user Directory for the server

public java.lang.Object getInstance (PropertyHandle ph)
Finds the given object, using this object's object factory

public java.lang.Object generateEventQueueProxy (Principal proxyID)
Adds a server-side event queue proxy for a client behind a firewall.

public edu.arizona.cmi.collab.events.PropertyEvent[] popEventQueueProxy (Principal proxyID)
Pops events off of an event queue proxy

public interface edu.arizona.cmi.collab.ServletViewer
implements edu.arizona.cmi.collab.Viewer

The interface for a servlet view of a property

Methods

public void service (HttpServletRequest request, HttpServletResponse response)
Handles the service request from the browser. This method is responsible for setting the content type, creating the out stream, etc. See AbstractServletViewer for example code.

public interface edu.arizona.cmi.collab.Viewer
implements edu.arizona.cmi.util.Identifiable

This is one of the most important classes/interfaces in the Framework. It defines the required methods for a Viewer that can be attached to a Property. See also ComponentViewer for more methods.

Methods

public void setProperty (Property p)
Sets the Property to be viewed

public edu.arizona.cmi.collab.Property getProperty ()
Gets the Property being viewed.

public abstract class edu.arizona.cmi.collab.AbstractViewer
extends javax.swing.JPanel
implements edu.arizona.cmi.collab.ComponentViewer

This is a convenience class for creating ComponentViewers. It defines many of the required methods so programmers do not have to.

Constructors

public AbstractViewer ()
Initializes the Form

Methods

public edu.arizona.cmi.util.GUID getID ()
Returns a unique long value to identify this Object as part of the Identifiable interface

public void setID (GUID guid)
Sets a unique long value to identify this Object as part of the Identifiable interface

public void setProperty (Property p)
Set the Property to view.

public edu.arizona.cmi.collab.Property getProperty ()
Get the Property to view.

public abstract void activate (boolean b)
This method is called by the Viewer's controller to notify it that it should change to an active state

public java.awt.Component getViewerUI ()
This method returns a Component for displaying the property. In this abstract implementation the class itself is a subclass of javax.swing.JPanel. 'this' is returned to be used as a display. Developers can derive

their Viewer objects from this class if they would like the convenience of simply needing to add Components to the JPanel object.

```
public java.awt.Component getRenderedIcon ()
```
This is intended to allow the Viewer to be represented as a small icon. The default implementation is to return a javax.swing.JLabel with the current Property name.

```
public abstract void setEditable (boolean b)
```
This is called by the Viewer's controller to toggle between edit and view modes

```
public void addChild (Property p)
```
Updates the UI when new child properties are added.

```
public void removeChild (GUID childID)
```
Updates the UI when new child properties are removed.

```
public void valueChange (String name, Object value)
```
Updates the UI when the value of this property changes.

Package edu.arizona.cmi.collab.client

```
public class  edu.arizona.cmi.collab.client.ClientApp
 extends javax.swing.JPanel
 implements  java.lang.Runnable,  java.awt.event.WindowListener
```

This class starts both Applet and Application client programs.

Constructors

`public ClientApp ()`
Initializes for the applet ClientApplet. Should not be used for any other reason as it does not init() the ClientDirector. This constructor is important to set the ClientApp's class loader as the RMIClassLoader for applets.

`public ClientApp (Hashtable props)`
initializes the form with a set of parameters

Methods

`public static javax.swing.JFrame getMainFrame ()`
Returns the main frame. There should be only one per application. Application programmers can use this method to add menus, toolbars, etc.

`public static javax.swing.JApplet getApplet ()`
Returns the applet instance, or null if this app is being run from the command line

`public static void main (String[] args)`
Starts the ClientApp as an application

`public static synchronized void showFrame (Hashtable props)`
Shows a main frame to display the ClientApp object

`public void run ()`
Runs the showFrame method with the appletProps static variable. This method is used for the client boostrapping required for applet/rmi connections.

`public static void exit ()`
Closes the main window and exits the application (if not in applet mode)

`public static void exit (int retval)`
Closes the main window and exits the application (if not in applet mode)

```
public void windowActivated (WindowEvent e)
public void windowClosing (WindowEvent e)
public void windowDeactivated (WindowEvent e)
public void windowDeiconified (WindowEvent e)
public void windowIconified (WindowEvent e)
public void windowOpened (WindowEvent e)
public void windowClosed (WindowEvent e)
```

public class edu.arizona.cmi.collab.client.ClientApplet
extends javax.swing.JApplet
implements java.awt.event.ActionListener

The common client all applets use within the collaborative framework. The class makes heavy use of ClientApp.java, where the real magic occurs. This is simply an applet wrapper for ClientApp.java.

Constructors

public ClientApplet ()

Methods

public void init ()
The entry point into the Applet

public synchronized void actionPerformed (ActionEvent e)
Called when the button is pressed

public class edu.arizona.cmi.collab.client.WebClient
extends javax.servlet.http.HttpServlet

The servlet entry point for the collaborative framework. This class simply takes the parameters and gives them to the requested PropertyObject with the requested ServletViewer.

Constructors

public WebClient ()

Methods

public void init (ServletConfig config)
Initialize global variables

public void service (HttpServletRequest request, HttpServletResponse response)
Service the request. This method is the entry-point for every request coming from web-based HTTP clients. It finds the right property, instantiates a ServletViewer for it, and hands control over to the ServletViewer. \par \par The method defines two top-level parameter strings: \par "Root": The ID of the Root Property for this request. If this parameter is not given or the ID does not map to a valid Property, the servlet shows an error and cannot continue. \par "View": (optional) The ServletViewer to instantiate for the property. If this parameter is not given, the default viewer for the property will be used.

protected void error (HttpServletResponse response, String st)
Reports an error

protected void error (HttpServletResponse response, String st, Exception e)
Reports an error, with an Exception

public java.lang.String getServletInfo ()

Package edu.arizona.cmi.collab.events

public class edu.arizona.cmi.collab.events.EventPopper
extends java.lang.Thread

A client side thread that pops events from server-side proxy event queues. This is used by the ClientDirector when we are in firewall situations and the client cannot export its EventQueue for true real-time behavior. This approximates it with polling.

Constructors

public EventPopper (Director clientDirector, long delay)
Constructor

Methods

public edu.arizona.cmi.util.GUID getID ()
Returns the id of this popper

public void cleanStop ()
Stops the popper cleanly

public synchronized void poll ()
Polls the server

public void run ()
Main loop

public class edu.arizona.cmi.collab.events.PropertyEventQueue
extends java.util.Vector
implements edu.arizona.cmi.collab.events.EventQueue, java.rmi.Remote, edu.arizona.cmi.util.Identifiable

The main event queue that stores and forwards events to destinations.

Constructors

public PropertyEventQueue ()

public PropertyEventQueue (boolean initialSetup)
Constructor for subclasses that don't want any initial setup (i.e. false = a no-op constructor

Methods

public void init ()
Initializes a property event queue

public edu.arizona.cmi.util.GUID getID ()
Returns the id of this event queue.

public void setID (GUID id)
Sets the id of this event queue. A unique id is very important for routing purposes.

public void checkQueue ()
Checks the event queue for new events to be processed

```
public void rollbackEvent (PropertyEvent pe)
```
Rolls back an event

```
public synchronized void processEvent (PropertyEvent pe)
```
Adds an event to be processed on the queue thread. This method must be synchronized for the thread queue to work. See Object.notify for information.

```
public void addEQDelegate (EventQueue delegate)
```
Adds a delegate. Event Queues should use this method

```
public synchronized void addDelegate (Identifiable delegate)
```
Adds a delegate. All objects other than Event Queues should use this method

```
public void removeDelegate (Identifiable delegate)
```
Removes a delegate.

public interface edu.arizona.cmi.collab.events.QueueListener
implements java.rmi.Remote, edu.arizona.cmi.util.Identifiable

Defines a listener to the event queue.

Methods

```
public void processEvent (PropertyEvent o)
```
When a PropertyEvent is handled by an EventQueue the event route is checked. If the PropertyEvent has not reached its final destination the next stop on the route is cast to its QueueListener interface and is passed the PropertyEvent through this method call.

public class edu.arizona.cmi.collab.events.DefaultEventFactory
implements edu.arizona.cmi.collab.events.EventFactory

Creates and throws events for a property object. This class defines the default event behavior.

Constructors

```
public DefaultEventFactory ()
```

Methods

```
public void addChild (Property prop, Property child)
```
Create and throw an add child event

```
public void removeChild (Property prop, GUID childID)
```
Create and throw a remove child event

```
public void valueChange (Property prop, String name, Object value)
```
Create and throw a value change event

public interface edu.arizona.cmi.collab.events.EventFactory

Defines the interface for an event factory whose job it is to create and throw standard events.

Methods

```
public void addChild (Property prop, Property child)
```
Create and throw an add child event

```
public void removeChild (Property prop, GUID childGUID)
```
Create and throw a remove child event

```
public void valueChange (Property prop, String name, Object value)
```
Create and throw a value change event

public abstract class edu.arizona.cmi.collab.events.PropertyEvent
implements java.io.Serializable, java.lang.Runnable

The superclass of all Framework server-to-client events.

Constructors

```
public PropertyEvent ()
public PropertyEvent (EventProducer source)
public PropertyEvent (Object route, long seqn)
```

Methods

```
public long getSequenceNumber ()
```
The sequence number is used to ensure events reach their destination in the order they were sent.

```
public void setDestination (Identifiable o)
```
Identifies this event as rolling back (i.e. going backwards through the route structure) back to its source. This occurs if it hits a cancelled or null connection

```
public void setRollback (boolean b)
```
Identifies this event as rolling back (i.e. going backwards through the route structure) back to its source. This occurs if it hits a cancelled or null connection

```
public edu.arizona.cmi.util.Identifiable getDestination ()
```
See setDestination.

```
public java.lang.Object getRouteInfo ()
```
Returns the route information of this event. A route is composed of an array of GUIDs that denote the IDs of event queues and viewers that the event should span across.

```
public void setRouteInfo (Object route)
```
Sets the route information. See getRouteInfo.

```
public void setProducer (EventProducer source)
```
Sets the producer of this event

```
public edu.arizona.cmi.collab.events.EventProducer getProducer ()
```
Returns the producer of this event.

```
public abstract void doEvent (Identifiable dest)
```
This method is run when the event reaches its final destination.

```
public void run ()
```

```
public abstract java.lang.Object clone ()
```
This method is important because the event will be cloned for each listener. Subclasses must ensure the event data is fully cloned with a deep copy.

public class edu.arizona.cmi.collab.events.ValueChangeEvent
extends edu.arizona.cmi.collab.events.PropertyEvent

Thrown when a Property's value is changed.

Constructors

```
public ValueChangeEvent (EventProducer producer, String name, Object val, long seq)
```

Methods

```
public long getSequenceNumber ()
public java.lang.String getPropertyName ()
public java.lang.Object getPropertyValue ()
public void doEvent (Identifiable dest)
public java.lang.Object clone ()
```

public class edu.arizona.cmi.collab.events.AddChildEvent
extends edu.arizona.cmi.collab.events.PropertyEvent

Automatically thrown when a new child is added to a property.

Constructors

```
public AddChildEvent (EventProducer producer, Property prop, long seq)
```
Creates a new AddChildEvent

Methods

```
public void doEvent (Identifiable dest)
```
Performs this event's action when it hits the target viewer object

```
public java.lang.Object clone ()
```
Clones this object

public interface edu.arizona.cmi.collab.events.EventQueue
implements edu.arizona.cmi.collab.events.QueueListener,
java.rmi.Remote

The remote interface exposed by event queues via RMI.

Methods

```
public void addDelegate (Identifiable delegate)
```
Adds a delegate. All objects other than Event Queues should use this method

```
public void addEQDelegate (EventQueue delegate)
```
Adds a delegate. Event Queues should use this method

```
public void removeDelegate (Identifiable delegate)
```
Removes a delegate.

```
public void rollbackEvent (PropertyEvent pe)
```
Rolls back an event

public interface edu.arizona.cmi.collab.events.EventProducer

An event producer that holds a list of listening consumers. This is normally a Property object.

Methods

public void addEventRoute (Object route)
Notifies the producer to add this route to its list of listening consumers

public void removeEventRoute (Object route)
Notifies the producer to delete this route from its list of listening consumers

public class edu.arizona.cmi.collab.events.RemoveChildEvent
extends edu.arizona.cmi.collab.events.PropertyEvent

Automatically thrown when a child is remove from a property.

Constructors

public RemoveChildEvent (EventProducer producer, GUID guid, long seq)
Creates a new AddChildEvent

Methods
public void doEvent (Identifiable dest)
Performs this event's action when it hits the target viewer object

public java.lang.Object clone ()
Clones this object

Package edu.arizona.cmi.collab.security

public class edu.arizona.cmi.collab.security.DefaultPermission
implements java.security.acl.Permission, java.io.Serializable

The default permission to use with the framework.

Constructors

public DefaultPermission ()
Constructor – gives the default value of all permissions

public DefaultPermission (int permission)
Constructor – gives a specific permission in the range of the static constant permission values in this class

Methods

public int getPermission ()
Returns the permission granted by this object

public boolean equals (Object p)
Checks to see if the this permission implies the same rights as the given permission p

public java.lang.String toString ()
Returns this permission as a string

public class edu.arizona.cmi.collab.security.PropertyACL
implements java.security.acl.Acl, java.io.Serializable

Holds the entries (and Permissions) who have access to a PropertyObject

Constructors
public PropertyACL ()

Methods

public boolean addOwner (Principal caller, Principal owner)
Always returns false. Owners are specified through their Permissions instead.

public boolean deleteOwner (Principal caller, Principal owner)
Here for interface consistency only. Owners are specified though their Permissions instead.

public boolean isOwner (Principal owner)
Returns true if the given principal is an owner of the ACL.
Parameters: owner - the principal to be checked to determine whether or not it is an owner.
Returns: true if the passed principal is in the list of owners, false if not.

public void setName (Principal caller, String name)
Sets the name of this ACL. Does nothing in the framework since names add no value.

public java.lang.String getName ()
Returns the name of this ACL. Returns a static name since individual names add no value to the framework.

```
public boolean addEntry (Principal caller, AclEntry entry)
```
Adds an ACL entry to this ACL. An entry associates a principal (e.g., an individual or a group) with a set of permissions. Each principal can have at most one positive ACL entry (specifying permissions to be granted to the principal) and one negative ACL entry (specifying permissions to be denied). If there is already an ACL entry of the same type (negative or positive) already in the ACL, false is returned.
Parameters: caller - the principal invoking this method. It must be an owner of this ACL.
entry - the ACL entry to be added to this ACL.
Returns: true on success, false if an entry of the same type (positive or negative) for the same principal is already present in this ACL.
Throws: NotOwnerException - if the caller principal is not an owner of this ACL.

```
public synchronized boolean removeEntry (Principal caller, AclEntry entry)
```
Removes an ACL entry from this ACL.
Parameters: caller - the principal invoking this method. It must be an owner of this ACL.
entry - the ACL entry to be removed from this ACL.
Returns: true on success, false if the entry is not part of this ACL.
Throws: NotOwnerException - if the caller principal is not an owner of this Acl.

```
public java.util.Enumeration getPermissions (Principal user)
```
Returns an enumeration for the set of allowed permissions for the specified principal (representing an entity such as an individual or a group). This set of allowed permissions is calculated as follows: If there is no entry in this Access Control List for the specified principal, an empty permission set is returned. Otherwise, the principal's group permission sets are determined. (A principal can belong to one or more groups, where a group is a group of entries, represented by the Group interface.) The group positive permission set is the union of all the positive permissions of each group that the principal belongs to. The group negative permission set is the union of all the negative permissions of each group that the principal belongs to. If there is a specific permission that occurs in both the positive permission set and the negative permission set, it is removed from both. The individual positive and negative permission sets are also determined. The positive permission set contains the permissions specified in the positive ACL entry (if any) for the principal. Similarly, the negative permission set contains the permissions specified in the negative ACL entry (if any) for the principal. The individual positive (or negative) permission set is considered to be null if there is not a positive (negative) ACL entry for the principal in this ACL. The set of permissions granted to the principal is then calculated using the simple rule that individual permissions always override the group permissions. That is, the principal's individual negative permission set (specific denial of permissions) overrides the group positive permission set, and the principal's individual positive permission set overrides the group negative permission set.
Parameters: user - the principal whose permission set is to be returned.
Returns: the permission set specifying the permissions the principal is allowed.

```
public java.util.Enumeration entries ()
```
Returns an enumeration of the entries in this ACL. Each element in the enumeration is of type AclEntry.
Returns: an enumeration of the entries in this ACL.

```
public int size ()
```
Returns the number of entries in this PropertyACL

```
public boolean checkPermission (Principal principal, Permission permission)
```
Checks whether or not the specified principal has the specified permission. If it does, true is returned, otherwise false is returned. More specifically, this method checks whether the passed permission is a member of the allowed permission set of the specified principal. The allowed permission set is determined by the same algorithm as is used by the getPermissions method.
Parameters: principal - the principal, assumed to be a valid authenticated Principal.

permission - the permission to be checked for.
Returns: true if the principal has the specified permission, false otherwise.

`public java.lang.String toString ()`
Returns a string representation of the ACL contents.

`public class edu.arizona.cmi.collab.security.PropertyACLEntry`
`implements java.io.Serializable, java.security.acl.AclEntry`

An ACL Entry for the framework

Constructors

`public PropertyACLEntry ()`
Constructor

`public PropertyACLEntry (Principal user)`
Constructor for server-side PropertyACLEntry creation

`public PropertyACLEntry (String username)`
Constructor for client-side PropertyACLEntry creation. Since clients do not know other user's Principals, a username will suffice. The server will dereference the username to the principal when the AclEntry is added to a Property's real ACL.

Methods

`public synchronized boolean setPrincipal (Principal user)`
Specifies the principal for which permissions are granted or denied by this ACL entry. If a principal was already set for this ACL entry, false is returned, otherwise true is returned.
Parameters: user - the principal to be set for this entry.
Returns: true if the principal is set, false if there was already a principal set for this entry.

`public java.security.Principal getPrincipal ()`
Returns the principal for which permissions are granted or denied by this ACL entry. Returns null if there is no principal set for this entry yet.
Returns: the principal associated with this entry.

`public void setNegativePermissions ()`
Sets this ACL entry to be a negative one. That is, the associated principal (e.g., a user or a group) will be denied the permission set specified in the entry. Note: ACL entries are by default positive. An entry becomes a negative entry only if this setNegativePermissions method is called on it.

`public boolean isNegative ()`
Returns true if this is a negative ACL entry (one denying the associated principal the set of permissions in the entry), false otherwise.
Returns: true if this is a negative ACL entry, false if it's not.

`public boolean addPermission (Permission permission)`
Adds the specified permission to this ACL entry. Note: An entry can have multiple permissions.
Parameters: permission - the permission to be associated with the principal in this entry.
Returns: true if the permission was added, false if the permission was already part of this entry's permission set.

`public boolean removePermission (Permission permission)`
Removes the specified permission from this ACL entry.
Parameters: permission - the permission to be removed from this entry.

Returns: true if the permission is removed, false if the permission was not part of this entry's permission set.

`public boolean checkPermission (Permission permission)`
Checks if the specified permission is part of the permission set in this entry.
Parameters: permission - the permission to be checked for.
Returns: true if the permission is part of the permission set in this entry, false otherwise.

`public java.util.Enumeration permissions ()`
Returns an enumeration of the permissions in this ACL entry.
Returns: an enumeration of the permissions in this ACL entry.

`public java.lang.String toString ()`
Returns the username of this AclEntry, if it has been set
Returns: a string representation of the contents.

`public java.lang.Object clone ()`
Clones this ACL entry.
Returns: a clone of this ACL entry.

`public class edu.arizona.cmi.collab.security.AccessControlException`
`extends java.lang.Exception`
`implements java.io.Serializable`

Thrown when the user doesn't have rights to an object

Constructors

`public AccessControlException ()`
Constructor

`public AccessControlException (String action)`
Constructor

`public class edu.arizona.cmi.collab.security.LockedException`
`extends edu.arizona.cmi.collab.security.AccessControlException`
`implements java.io.Serializable`

Thrown when a user tries to access a locked object

Constructors

`public LockedException ()`
Constructor

`public LockedException (String action)`
Constructor

`public class edu.arizona.cmi.collab.security.PrincipalImpl`
`implements java.security.Principal, java.io.Serializable`

Represents a user in the framework. This implementation is password-centric, giving a user name and password. Application Admin modules are responsible for ensuring duplicate user names are not created.

Constructors

`public PrincipalImpl ()`
Constructor

`public PrincipalImpl (String id)`
Constructor

Methods

`public boolean equals (Object another)`
Compares this principal to the specified object. Returns true if the object passed in has at least the same entries as this object.
Parameters: another - principal to compare with.
Returns: true if the principal passed in is the same as that encapsulated by this principal, and false otherwise.

`public java.lang.String toString ()`
Returns a string representation of this principal.

`public int hashCode ()`
Returns a hashcode for this principal.

`public java.lang.String getName ()`
Returns the username of this principal.

`public java.security.Principal getFirst ()`
Returns only the first entry in this principal

`public void concat (PrincipalImpl p)`
Adds the entries of the given Principal to this one

`public void subtract (PrincipalImpl p)`
Removes all entries from this Principal that match those in the given p

`public class edu.arizona.cmi.collab.security.Lock`
`implements java.io.Serializable`

A lock for a permission for a user on an object

Constructors

`public Lock (Principal sessionID, Permission perm)`
Constructor

Methods

`public boolean isExpired ()`
Returns true if this lock has expired

`public boolean allows (Principal sessionID, Permission p)`
Determines whether this lock allows the given permission. If the sessionID matches the sessionID of this lock, the permission is always allowed. If the given permission p equals the lock's permission, the permission is disallowed. This method does NOT check whether the lock has expired (this is often checked by the calling program, so it is excluded here for efficiency).

`public boolean equals (Object lock)`
Checks whether this lock is equal to another (principals permissions the same)

public class edu.arizona.cmi.collab.security.SessionExpiredException
extends edu.arizona.cmi.collab.security.AccessControlException
implements java.io.Serializable

Thrown when the user's session has expired (when it has been dormant for > 30 minutes)

Constructors
public SessionExpiredException ()
Constructor

public SessionExpiredException (String action)
Constructor

Package edu.arizona.cmi.collab.user

public interface edu.arizona.cmi.collab.user.Directory
implements edu.arizona.cmi.collab.Property

A directory to all the users on the system

Methods

public java.security.Principal login (String username, String password)
Logs a user in and returns a session id (valid for 30 minutes between each request. Throws an AccessControlException if the username password do not verify.

public void logout (Principal sessionID)
Logs a user out and disables his/her sessionid

public void addUser (User user)
Adds a new user to the directory. The given Principal must have correct permissions to the user directory object.

public void addUser (User user, Principal sessionID)
Adds a new user to the directory. The given Principal must have correct permissions to the user directory object.

public edu.arizona.cmi.collab.Property getUser (String username)
Gets a user from the child list. We do not extend NamingPropertyObject because we would have to index by name. This would work well until a user changed his/her username. We have no way of updating a the namingProperty hashtable since the user object is for a child one level below this object.

public java.security.Principal getPrincipal (Principal guid)
Translates the session id into the user's Principal, assuming the user has a valid session GUID.

public java.security.Principal getPrincipal (String username)
Translates the given string into the user's Principal, assuming the username exists.

public class edu.arizona.cmi.collab.user.User
implements java.io.Serializable

Represents a user or a group

Constructors

public User ()
Creates a new user

Methods

public edu.arizona.cmi.util.GUID getID ()
Returns this user's GUID

public void setUsername (String username)
Sets the user name

public java.lang.String getUsername ()
returns the username

`public boolean setPassword (String newPassword)`
Sets a new password. The password must be null or this method fails

`public boolean setPassword (String oldPassword, String newPassword)`
Sets a new password. The old password must match the current one or the method fails

`public void setLastName (String lName)`
Sets the last name

`public java.lang.String getLastName ()`
Returns the last name

`public void setFirstName (String fName)`
Sets the first name

`public java.lang.String getFirstName ()`
Returns the first name

`public java.lang.String getName ()`
Returns the full name of this person. Guaranteed to return a non-null result (although it might be "")

`public java.lang.String getReversedName ()`
Returns the full name of this person, last name first. Guaranteed to return a non-null result (although it might be "")

`public void setOrganization (String org)`
Sets the user's organization

`public java.lang.String getOrganization ()`
Returns the user's organization

`public void setPhoneNumber (String phone)`
Sets the user's phone number

`public java.lang.String getPhoneNumber ()`
Returns the user's phone number

`public void setEmail (String email)`
Sets the user's e-mail address

`public java.lang.String getEmail ()`
Returns the user's email address

`public void setUserObject (Serializable obj)`
Sets the custom user object (can store any type of serialized object for custom use)

`public java.io.Serializable getUserObject ()`
Returns the custom user object

`public long getTimeCreated ()`
Returns the time this user was created in the system

`public java.lang.String getDateCreated ()`
Returns the time this user was created, formatted in the default mm/dd/yyyy format

```
public java.security.Principal getPrincipal ()
```
Returns the user's principal object

```
public boolean verify (String pass)
```
Verifies a password

```
public void addGroup (PrincipalImpl group)
```
Adds this user to a group

```
public void removeGroup (PrincipalImpl group)
```
Removes this user from a group

```
public java.lang.String toString ()
```
Displays a textual view of this object

Package edu.arizona.cmi.util

public class edu.arizona.cmi.util.GUID
implements edu.arizona.cmi.util.Identifier, java.io.Serializable, java.security.Principal, edu.arizona.cmi.collab.IndexKey

GUID stands for Globally Unique IDentifier. The GUID class creates a two long based upon two pieces of information: The current system time and The local machine's IP address. This combination creates a number that should be unique in both space and time.

Constructors

`public GUID ()`
Constructs a new GUID from the current system time and IP address

`public GUID (long ip, long timestamp)`
Constructor

`public GUID (String hex)`
Constructor

`public GUID (Socket sock)`
Constructor

`public GUID (GUID copy)`
Creates a deep copy of the GUID by cloning the mybits attribute.
Parameters: copy - The GUID to copy.

Methods

`protected synchronized void setGUID ()`
Sets the timestamp and ip

`protected synchronized void setHere (InetAddress a)`
Sets the local address part of the long

`public boolean equals (Identifier ID)`
This is used to check the equality of the current GUID and another GUID
Parameters: ID - The GUID that should be compared.

`public boolean equals (Object o)`
Equals in keeping with Object.java

`public boolean equals (GUID guid)`
Equals given a GUID

`public java.lang.String toHex ()`
Returns this GUID in hex format

`public java.lang.String toString ()`
Returns the bit set as a hex string.

`public java.lang.String asDecimal ()`
Returns the two pieces of the bit set

```
public int hashCode ()
```
Returns the hashcode for this object. This method is very important as GUIDs are usually the key for hashtables, trees, DBs, etc.

```
public static java.lang.String generate ()
```
Generates a hexadecimal String representing a unique GUID

```
public java.lang.String getName ()
```
Returns the name of this principal.

```
public static long hexToLong (String hex)
```
Returns a long from the given hex string

```
public java.lang.Object getIndexObject ()
```
Returns the object denoting the index of the child. This is included for the IndexKey interface. This GUID serves as both the wrapper object and the object itself for efficiency purposes.

```
public java.lang.String getKeyType ()
```
Returns the fully-qualitifed name of this class. This return type should match the Indexer's getValidKeyType() method.

```
public java.lang.String getIndexerType ()
```
Returns the fully-qualified name of the related Indexer

`public class edu.arizona.cmi.util.Utility`

Static utility methods used throughout the Framework classes.

`Constructors`

```
public Utility ()
```

`Methods`

```
public static java.lang.Object instantiate (String c)
```
Creates an instance of the class defined by the String c and returns it as an Object

```
public static java.rmi.registry.Registry getRMIConnection ()
```
This may be a misnomer. Perhaps a better name is getRegistry(). This method returns a reference to the RMI registry running at the URL composed from the server and port Strings sent into setServer (String h, String p)

```
public static void rebind (String name, Remote o)
```
Rebinds an object in the Naming Service

```
public static void setServer (String s, String p)
```
Sets the URL to use for finding the RMI registry

```
public static java.util.Hashtable readProperties (String filename)
```
This method reads system configuration properties from an ASCII text file. The file should contain only properties of the form: : The properties are returned in a hashtable that indexes the values by their labels. If an error occurs this method attempts to recover as gracefully as possible by returning the properly identified properties along with the Exception object

```
public static java.util.Hashtable readProperties (Reader rdr)
```
This method reads system configuration properties from an ASCII text file. The file should contain only properties of the form: : The properties are returned in a hashtable that indexes the values by their labels.

If an error occurs this method attempts to recover as gracefully as possible by returning the properly identified properties along with the Exception object

`public static java.lang.Object newRoute (Director d)`
This method generates a new Object for encapsulating an event route.

`public static java.lang.Object addRouteID (GUID routeID, Object route)`
This method appends an ID for an Identifiable object to the end of the event route and returns a new event route.

`public static boolean emptyRoute (Object route)`
This method returns true only when the route Object is empty.

`public static void resetRoute (Object route)`
Resets the index on a route

`public static boolean routeEquals (Object route1, Object route2)`
Compares two routes

`public static edu.arizona.cmi.util.GUID popNextRouteID (Object route)`
This method retrieves the next Identifiable object from the event route.

`public static edu.arizona.cmi.util.GUID rollbackRouteID (Object route)`
Rolls back the event route one object. Opposite method of popNextRouteID()

`public static boolean fullRoute (Object route)`
Returns whether this route is full or not

`public static java.lang.Object cloneRoute (Object route)`
Clones a route

`public static java.lang.Object getRoute (Director d, Identifiable ql)`
This constructs the complete event route for the Identifiable object based upon routing information stored in the Director object.

`public static void printRoute (Object route)`
Simply prints a route for debugging purposes (to System.err)

`public static edu.arizona.cmi.util.GUID getGUID ()`
This returns a unique long value that is guaranteed to be a unique identifier for an object.

`public interface edu.arizona.cmi.util.Identifiable`

Defines the interface for an object that has an ID.

Methods

`public void setID (GUID guid)`
Sets the id of this object.

`public edu.arizona.cmi.util.GUID getID ()`
Returns the id of this object.

public class edu.arizona.cmi.util.ThreadPool

Implements a thread pool. Assumes one thread pool per JVM (i.e. it's static) All objects that use the thread pool must implement the Runnable interface. To use, simply call: ThreadPool.start(runnableObject) The thread pool will call runnableObject.run() as soon as it has a free thread.

Constructors

`public ThreadPool ()`

Methods

`public static synchronized void init ()`
Initializes the thread pool (not required but can be called for explicit initialization)

`protected static synchronized void setPoolSize (int s)`
Sets the pool size

`public static synchronized void setMinPoolSize (int min)`
Sets the minimum pool size.

`public static synchronized void setMaxPoolSize (int max)`
Sets the maximum pool size. max public static synchronized void setCheckDelay (long ms)
Sets the time between load checking, in milliseconds

`public static void setLoadFactor (float load)`
Sets the load factor (must be 0 public static synchronized void setMaxInactivity (long ms)
Sets the maximum time for inactivity before threads are reclaimed

`public static synchronized void checkLoad ()`
Explicitly checks the load on the thread pool to ensure it conforms to settings

`public static void start (Runnable obj)`
Runs a runnable object within this thread pool

`public static synchronized void checkWaiting ()`
Explicitly tries to assign threads from the pool to run any waiting objects

`public static java.lang.String debugString ()`
Returns the object for debugging. Cannot override toString since it is static here

REFERENCES

ACI (1999) Net24. http://www.appliedcommunications.com/tsainc/aci/Solutions/sysman/systran.asp.

Abdel-Wahab, H. & Jeffay, K. (1994) Issues, Problems and Solutions in Sharing Clients on Multiple Displays. Internetworking – Research and Practice, vol. 5, no. 1 (March), 37-45.

Applegate, L. M., Chen, T. Teng, Konsynski, B. R., & Nunamaker Jr., J. F. (1987) Knowledge Management in Organizational Planning. Journal of Information Systems, vol. 3, no. 4 (Spring), 20-38.

Banavar, G., Kaplan, M., Shaw, K., Strom, R. E., Sturman, D. C., & Wei-Tao (1999) Information Flow Based Event Distribution Middleware. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications, 114-121.

Bell Labs (2000) The Creation of the UNIX Operating System http://www.bell-labs.com/history/unix/.

Bentley, R., Rodden, T., Sawyer, P., & Sommerville, I. (1994) Architectural Support for Cooperative Multiuser Interfaces. IEEE Computer (May), 37-45.

Bier, E. A. & Freeman, S. (1991) MMM: A User Interface Architecture for Shared Editors on a Single Screen. UIST (November 11-13), 79-88. Association for Computing Machinery.

Bowers, J. & Rodden, T. (1993) Exploding the Interface: Experiences of a CSCW Network. INTERCHI (April 24-29) Association for Computing Machinery.

Burridge, R. (1999) Java Shared Data Toolkit User Guide. (October 4), 1-87. Sun Microsystems. http://java.sun.com/.

Chabert, A., Grossman, E., Jackson, L., & Pietrovicz, S. (2000) NCSA Habanero - Synchronous Collaborative Framework and Environment. Software Development Division at the National Center for Supercomputing Applications. http://havefun.cvsa.uiuc.edu/habanero/.

Chiozzi, G. (1996) Extending the Model-View-Controller Paradigm to the Development of Interactive Graphical Applications. Rivista di Informatica, vol. 26, no. 1, 25-35.

Clip2 (2000) Bandwidth Barriers to Gnutella Network Scalability Clip2 Distributed Search Services, http://dss.clip2.com/dss_barrier.html.

Codd, E.F. (1970) A Relational Model of Data for Large Shared Databases. Communications of the ACM, vol 13, no. 6. (June).

Coutaz, J., Cockton, G. (1990) Architecture Models for Interactive Software Failures and Trends. Engineering for Human-Computer Interaction, 137-153. Elsevier Science Publishers.

Coyle, F. P. (1996) Smalltalk's Model-View-Controller Architecture: The Evolution of a Design Pattern. Object Oriented Programming (January 1), 142-144. SIGS Conferences, Bergisch Gladbach, Germany.

Coyle, F. P. (1994) Smalltalk's Model-View-Controller Architecture. Object Expo, 43-49. SIGS Publications.

Dennis, A. R., George, J. F., Jessup, L. M., Nunamaker, Jr., J. F., & Vogel, D. R. (1988) Information Technology to Support Electronic Meetings. MIS Quarterly, vol. 12, no. 4 (December), 591-624.

Dewan, P. (1999a) Designing and Implementing Multi-User Applications: A Case Study. Technical Report http://www.cs.unc.edu/~dewan/papers/case_ps.

Dewan, P. (1999b) A Tour of the Suite User Interface System. Technical Report http://www.cs.unc.edu/~dewan/papers/tour_ps.

Dewan, P. (1999c) A Guide to Suite. Technical Report http://www.cs.unc.edu/~dewan/papers/suite-guide_ps.

Dewan, P. (1999d) Principles of Designing Multi-User User Interface Development Environments. Technical Report http://www.cs.unc.edu/~dewan/papers/prin_ps.

Dewan, P. (1999e) Multiuser Architectures. Technical Report http://www.cs.unc.edu/~dewan/papers/arch_ps.

Dewan, P. (1999f) A Survey of Applications of CSCW Including Some in Educational Settings. Technical Report http://www.cs.unc.edu/~dewan/papers/edmedia_ps.

Dewan, P. & Choudhary, R. (1999a) A High-Level and Flexible Framework for Implementing Multi-User User Interfaces. Technical Report http://www.cs.unc.edu/~dewan/papers/framework_ps.

Dewan, P. & Choudhary, R. (1999b) Coupling the User Interfaces of a Multiuser Program. Technical Report http://www.cs.unc.edu/~dewan/papers/coupling_ps.

Dewan, P., Choudhary, R., & Shen, H. (1999) An Editing-Based Characterization of the Design Space of Collaborative Applications. Technical Report http://www.cs.unc.edu/~dewan/papers/edit_ps.

Dewan, P. & Riedl, J. (1999) Towards Computer-Supported Concurrent Software Engineering. Technical Report http://www.cs.unc.edu/~dewan/papers/cse_ps.

Dewan, P. & Sharma, A. (1999) An Experiment in Interoperating Heterogeneous Collaborative Systems. Technical Report http://www.cs.unc.edu/~dewan/papers/interop_ps.

Dewan, P. & Shen, H. (1999) Flexible Meta Access-Control for Collaborative Applications. Technical Report http://www.cs.unc.edu/~dewan/papers/meta_ps.

Dewan, P. & Beaudouin-Lafon, M. (1998) Architectures for Collaborative Applications. Collaborative Systems, Chapter 9. John Wiley & Sons Ltd..

Dewan, P. & Choudhary, R. (1994) Primitives for Programming Multi-User Interfaces. UIST (November 11-13) Association for Computing Machinery.

Dewan, P (1993) Tools for Implementing Multiuser User Interfaces. User Interface Software, vol. Chapter 8 John Wiley & Sons Ltd..

Dewan, P. & Solomon, M. (1990) An Approach to Support Automatic Generation of User Interfaces. ACM Transactions on Programming Languages and Systems, vol. 12, no. 4 (October), 566-609.

Dorohonceanu, B., Sletterink, B., & Marsic, I. (2000) A Novel User Interface for Group Collaboration. Proceedings of the Hawaii International Conference on System Sciences (January 4-7) IEEE.

Dorohonceanu, B. & Marsic, I. (1999) A Desktop Design for Synchronous Communication. Proceedings of the Graphics Interface, 27-35.

Ellis, C., Gibbs, S., & Reln, G. (1991) Groupware: Some Issues and Experiences. Communications of the ACM, vol. 34, no. 1 (January), 39-58.

Fagan, M. E. (1976) Design and code inspections to reduce errors in program development. IBM Systems Journal, vol. 15, no. 3, 182-211.

Gall, U. & Hauck, F. J. (1999) Promondia: A Java-Based Framework for Real-Time Group Communication in the Web. Department of Computer Science, University of Erlangen-Nurnberg, Germany. http://www4.informatik.uni-erlangen.de/Projects/promondia/.

Gnutella (2000) Gnutella Distributed File System. http://gnutella.wego.com.

Goldberg, A. (1990) Information models, views, and controllers (software re-use). Dr. Dobb's Journal of Software Tools, vol. 15, no. 7 (July 1), 54, 56-9, 61, 106-7.

Goldberg, A. (1983) SmallTalk-80: The Interactive Programming Environment Addison-Wesley Publishers.

Goldberg, A. & Robson, D. (1983) Smalltalk-80: The Language and its Implementation Addison-Wesley Publishers.

Graham, T. Nicolas (1999) The Clock Language: Preliminary Reference Manual. Technical Report http://dundee.cs.queensu.ca/~graham/stl/pubs/clockReference_ps.

Graham, T. Nicolas, Morton, C. A., & Urnes, T. (1999) ClockWorks: Visual Programming of Component-Based Software Architectures. Technical Report http://dundee.cs.queensu.ca/~graham/stl/pubs/cw_ps.

Graham, T. Nicolas & Grundy, J. (1998) External Requirements of Groupware Development Tools. Workshop on Requirements of Groupware Development Tools EHCI.

Graham, T. Nicolas (1997) GroupScape: Integrating Synchronous Groupware and the World Wide Web. Technical Report http://dundee.cs.queensu.ca/~graham/stl/pubs/interact97.pdf.

Graham, T. Nicolas & Urnes, T. (1996) Linguistics Support for the Evolutionary Design of Software Architectures. Technical Report (March) http://dundee.cs.queensu.ca/~graham/stl/pubs/icse18_ps.

Graham, T. Nicolas, Urnes, T., & Nejabi, R. (1996) Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware. Technical Report http://dundee.cs.queensu.ca/~graham/stl/pubs/uist96_ps.

Graham, T. Nicolas (1994) Declarative Development of Interactive Systems http://dundee.cs.queensu.ca/~graham/stl/pubs/ddis.html.

Greenberg, S., Roseman, M, & Beaudouin-Lafon, M. (1996) Groupware Toolkits for Synchronous Work. Trends in CSCW John Wiley & Sons Ltd..

GroupSystems.com (2000) GroupSystems Software Suite http://www.groupsystems.com.

Gundoju, V. & Minoura, T. (1999) Distributed Observable/Observer: A Distributed Real-Time Object-Communication Mechanism. Proceedings First International Symposium on Object-Oriented Real-Time Distributed Computing, 358-362. IEEE.

Hamid, T., Xu, W., Dollimore, J., & Miranda, E. (1993) Towards Distributed MVC. Technical Report (March 1) http://http://www.dcs.qmw.ac.uk/publications/report_abstracts/1993/624.

Harrison, T., O'Ryan, C., & Levine, D. (1999) The Design and Performance of a Real-time CORBA Event Service. IEEE Journal on Selected Areas in Communications (May 4)

Heinckiens, P., Tromp, H., & Hoffman, C. (1995) A Model-View-Controller Approach to Object Persistence. , 307-319. Proceedings of the Seventeenth International Conference on Technology of Object-Oriented Languages and Systems.

Hill, R. D., Brinck, T., Rohall, S. L., Patterson, J. F., & Wilner, W. (1994) The Rendezvous Architecture and Language for Constructing Multiuser Applications. ACM Transactions on Computer-Human Interaction, vol. 1, no. 2 (June), 81-125.

Hopkins, J. (1999) Using The Model-View-Controller Architecture in an Object-Oriented Application. Object EXPO Europe, 81-85. SIGS Publications.

Horstmann, T. & Wasserschaff, M.Raymond, K. & Armstrong, L. (1995) Experiences with Groupware Development Under CORBA. Open Distributed Processing: Experiences with Distributed Environments, 297-308. Amprun & Hall.

Hurfin, M. & Raynal, M. (1998) Asynchronous Protocols to Meet Real-Time Constraints: Is It Really Sensible? How to Proceed?. The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (April 20) Kyoto, Japan. http://www.computer.org/conferen/proceed/isorc/8430/8430toc.htm.

IBM (1999a) MQSeries. http://www.software.ibm.com/mqseries/.

IBM (1999b) Lotus Domino Collaborative Server. http://www.lotus.com.

IONA (1999) OrbixTalk. http://www.iona.com/.

Inprise (1999) Visibroker. http://www.inprise.com/visibroker/.

Java Developers Journal (1999) Event Management and Enterprise JavaBeans. (October 24) http://http://www.sys-con.com/java/feature/4-1/EventManagementEnterpriseJavaBeans/.

Johnson, P. (1996) State as an Organizing Principle for CSCW Architectures. Technical Report 96-05 Collaborative Software Development Laboratory, University of Hawaii.

Johnson, P. (1995a) The Egret Primer: A Tutorial Guide To Coordination and Control in Interactive Client-Server-Agent Applications. Technical Report 95-10 (October 6) Collaborative Software Development Laboratory, University of Hawaii.

Johnson, P. (1995b) Egret: A Framework for Advanced CSCW Applications. Technical Report 95-23 Collaborative Software Development Laboratory, University of Hawaii.

Johnson, P. M. (1994) Collaboration in the Small vs. Collaboration in the Large. Technical Report (October 13) http://csdl.ics.hawaii.edu/techreports/94-15/citl.ps.

Johnson, P. & Moore, C. (1994) Investigating Strong Collaboration with the Annotated Egret Navigator. Technical Report 94-20 Collaborative Software Development Laboratory, University of Hawaii.

Johnson, P. (1993) Experiences with EGRET: An Exploratory Group Work Environment. Technical Report 93-18 Collaborative Software Development Laboratory, University of Hawaii.

Johnson, P., Tjahjono, D., Wan, D., & Brewer, R. (1993) Gtables: From Egret 2.x.x to Egret 3.0.x. Technical Report 93-20 (November) Collaborative Software Development Laboratory, University of Hawaii.

Johnson, P. (1992) Supporting Exploratory CSCW with the EGRET Framework. Proceedings of the 1992 Conference on Computer Supported Cooperative Work

Juth, S. (1998) Collaboration Components for Programming Real-Time Synchronous Groupware Applications. Masters Thesis (October) Graduate School of Electrical and Computer Engineering, Rutgers University.

Krasner, G. E. & Pope, S. T. (1988) A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming (August/September), 26-49.

Lantz, K. A., Nowicki, W. I., & Theimer, M. M. (1985) An Empirical Study of Distributed Application Performance. IEEE Transactions on Software Engineering, vol. SE-11, no. 10 (October), 1162-1174.

Lantz, K. A. & Nowicki, W. I. (1984) Structured Graphics in Distributed Systems. ACM Transactions on Graphics, vol. 3, no. 1, 23-51. Association for Computing Machinery.

Lee, B., Park, T., Yeom, H., & Cho, Y. (1998) An Efficient Algorithm for Causal Message Logging. 17th IEEE Symposium on Reliable Distributed Systems (SRDS), 19-25.

Li, W., Wang, W., & Marsic, I. (1999) Collaboration Transparency in the DISCIPLE Framework. Proceedings of the ACM International Conference on Supporting Group Work (November 14-17) Association for Computing Machinery.

M/Ware (1999) InterPlay. http://www.mware.com/.

Malan, G. Robert, Jahanian, F., & Knoop, P. (1997) Comparison of Two Middleware Data Disseminatino Services in a Wide-Area Distributed System. IEEE, 411-419.

Marsic, I. (1999a) A Software Framework for Collaborative Applications. Proceedings of the Collaborative Technologies Workshop (November 10-11)

Marsic, I. (1999b) Real-Time Collaboration in Heterogeneous Computing Environments. Technical Report http://www.caip.rutgers.edu/disciple/Publications/itcc00.pdf.

Marsic, I. (1999c) DISCIPLE: A Framework for Multimodal Collaboration in Heterogeneous Environments. Technical Report http://www.caip.rutgers.edu/disciple/Publications/acm-cs.pdf.

Marsic, I. (1999d) DISCIPLE-Distributed System for Collaborative Information Processing and Learning. http://http://www.caip.rutgers.edu/disciple/.

Marsic, I. & Dorohonceanu, B. (1999) An Application Framework for Synchronous Collaboration using Java Beans. Proceedings of the Hawaii International Conference on Systems Sciences (January 5-8) IEEE.

Marsic, I. & Jonnalagadda, L. S. (1999) Using Network Traffic Statistics in Learning Object Migration Policies. Technical Report http://www.caip.rutgers.edu/disciple/Publications/hicss-31_ps.

Marsic, I. & Talapadi, S. (1999) Data-Centric Collaboration Using coZmo. Technical Report http://www.caip.rutgers.edu/disciple/Publications/coZmo.pdf.

Marsic, I. (1998) Data-Centric Collaboration in Heterogeneous Environments. Technical Report http://www.caip.rutgers.edu/disciple/Publications/iswc-98.pdf.

Marsic, I. & Jonnalagadda, K. S. L. (1996) The Role of Network Traffic Statistics in Devising Object Migration Policies. Real-Time Systems Symposium: Workshop on Resource Allocation Problems in Multimedia Systems (December) IEEE.

Menges, J. & Kevin, J. (1994) Inverting X: An Architecture for a Shared Distributed Window System. Proceedings of the Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (April), 53-64. IEEE Computer Society Press.

MessageQ (1999) The Independent Site for Application Integration. http://www.messageq.com.

MicroScript (1999) MicroScript Server. http://www.neonsoft.com/.

Microsoft (2000a) Microsoft Access, Microsoft NetMeeting, and Microsoft Foxpro Applications http://www.microsoft.com/.

Microsoft (2000b) Active Directory. http://www.microsoft.com/windows2000/library/howitworks/activedirectory/adarch.asp.

Microsoft (1999) Microsoft Message Queue. http://www.microsoft.com/ntserver/guide/msmq_rev_spectra.asp.

Mitchell, J. R. & Garg, V. K. (1998) A Non-Blocking Recovery Algorithm for Casual Message Logging. 17th IEEE Symposium on Reliable Distributed Systems (SRDS), 3-9.

MITRE (1994a) Reference for Extending the CVW Core Database. Technical Report The MITRE Corporation.

MITRE (1994b) Collaborative Virtual Workspace. Technical Report The MITRE Corporation. http://cvw.mitre.org/cvw/info/docs/CVWOverview.pdf.

MITRE (1994c) Reference for Extending the CVW Updater. Technical Report The MITRE Corporation.

Moore, C. (1995) Strong Collaboration in AEN. Technical Report http://csdl.ics.hawaii.edu/pub/tr/ics-tr-95-22.pdf .

Munson, J. P. & Dewan, P. (1999a) An Editing-Based Characterization of the Design Space of Collaborative Applications. Technical Report http://www.cs.unc.edu/~dewan/papers/edit_ps.

Munson, J. P. & Dewan, P. (1999b) A Flexible Object Mergine Framework. Technical Report http://www.cs.unc.edu/~dewan/papers/sync.ps.

Munson, J. P. & Dewan, P. (1997) Sync: A System for Mobile Collaborative Applications. Technical Report (March 14) http://www.cs.unc.edu/~dewan/papers/sync.ps.

Nejabi, R. (1995) Linguistic Support for Developing Groupware Systems (Clock Comparison). Masters Thesis (July) Graduate Program in Computer Science, York University. http://dundee.cs.queensu.ca/~graham/stl/pubs/nejabiMSC_ps.

Nelson, B. J. (1981) Remote Procedure Call, XEROX PARC CSL-81-9 (May)

Novell (2000) Novell Directory Services. http://www.novell.com/products/nds/.

Nunamaker, J. F., Briggs, R. O., & Mittleman, D. D. (1990) Electronic Meeting Systems: Ten Years of Lessons Learned. Technical Report CMI, University of Arizona

Nunamaker, J. F., Dennis, A. R., Valacich, J. S., Vogel, D. R., & George, J. F. (1991) Electronic Meeting Systems to Support Group Work. Communications of the ACM, vol. 34, no. 7 (July), 41-60.

Nunamaker, J.F., (1992). Build and learn, evaluate and learn. Informatica, vol 1, no. 1 (December), 1-6.

Nunamaker, J. F., Chen, M., & Purdin, T. (1990-1991) Systems Development in Information Systems Research. Journal of Management Information Systems, vol. 7, no. 3 (Winter), 86-106.

OMG (2000) The Object Management Group. http://www.omg.org/.

Oracle (1999) Oracle8i Advanced Queueing. http://www.oracle.com/.

Outback (1999) jEvents - a Java-based CORBA Event Service. Outback Research Group. http://http://www.outbackinc.com/products/jevents/intro.html.

Patterson, J. F. (1994) Comparing the Programming Demands of Single-User and Multi-User Applications. UIST (November 11-13), 87-94. Association for Computing Machinery.

Rakotonirainy, A., Berry, A., Crawley, S., & Milosesevic, Z. (1997) Describing Open Distributed Systems: A Foundation. , vol. 40, no. 8, 479-488. The Computer Journal.

Rao, S., Lorenzo, A., & Vin, H. (1998) The Cost of Recovery in Message Logging Protocols. 17th IEEE Symposium on Reliable Distributed Systems (SRDS), 10-18.

Reaves, M. S. (1998) Publish/Subscribe Now Breaking out of the Box. Wall Street Technology (Spring), 17-20.

Ritchie, D. M. & Thompson, K. (1978) The UNIX Time-Sharing System. The Bell System Technical Journal, vol. 57, num. 6 (July-August).

Rodgers, T. R. (1999) Software Inspections: Collaboration and Feedback. Dissertation Presented to the Faculty of the Department of Management Information Systems. University of Arizona.

Roseman, M. & Greenberg, S. (1999a) A Tour of TeamRooms. Technical Report http://www.cpsc.ucalgary.ca/projects/grouplab/projects/groupkit/.

Roseman, M. & Greenberg, S. (1999b) Building Groupware with GroupKit. Tcl/Tk Tools, 535-564. O'Reilly Press.

Roseman, M. & Greenberg, S. (1999c) Simplifying Component Development in an Integrated Groupware Environment. Technical Report http://www.cpsc.ucalgary.ca/projects/grouplab/projects/groupkit/.

Roseman, M. (1998) CSCW Toolkits. Technical Report (May 4) http://http://www.cpsc.ucalgary.ca/~roseman/.

Roseman, M. & Greenberg, S. (1995) Building Real Time Groupware with GroupKit, A Groupware Toolkit. Technical Report (July) http://www.cpsc.ucalgary.ca/projects/grouplab/projects/groupkit/.

Shefrin, E. A. & Purtilo, J. M. (1995) Tool Support for Collaborative Software Prototyping. IEEE Computer, 25-35.

Shen, H. & Dewan, P. (1999) Access Control for Collaborative Environments. Technical Report http://www.cs.unc.edu/~dewan/papers/access-control_ps.

Siemon, E., Buchner, J., & Frisch, M. (1999) Team Design of User Interfaces with Use of Agents, Based on a Distributed MVC (Model View Controller). ITG-Fachberichte, vol. 137, 45-55.

Spiteri, M. D. & Bates, J. (1998) An Architecture to Support Storage and Retrieval of Events. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, 443-458. Springer-Verlag London, London, UK.

Stefik, M., Foster, G., Bobrow, D. G., Kahn, K., Lanning, S., & Suchman, L. (1987) Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. Computing Practices, vol. 30, no. 1 (January), 32-47. Association for Computing Machinery.

Stevens, W. R. (1997) Unix Network Programming: Network Apis Sockets and Xti. Prentice Hall, 2nd ed.

Strom, R., Banavar, G., Miller, K., Prakash, A., & Ward, M. (1997) Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects. 17th International Conference on Distributed Computing Systems (May 27-30), 194-203. IEEE Computer Society Press.

Sun Microsystems (2000a) The Java Language. http://java.sun.com/

Sun Microsystems (2000b) RMI Architecture. http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmi-arch6.html.

Sun Microsystems (2000c) Java Authentication and Authorization Service. http://java.sun.com/products/jaas/.

Sun Microsystems (2000d) Jini Connection Technology. http://www.sun.com/jini/.

Sun Microsystems (1999) Java Message Queue. http://www.sun.com/workshop/jmq/.

Sundaram, S. (1998) A Collaboration-Enabling Framework for Java Beans. Masters Thesis (January) Graduate School of Electrical and Computer Engineering, Rutgers University.

TIBCO (1999) The TIB. http://www.tibco.com/.

Talarian (1999) SmartSockets. http://www.talarian.com/products/smartsockets/index.shtml.

Taylor, R. N, Medvidovic, N., Anderson, K. M., Whitehead, E. James, Robbins, J. E., Nies, K. A., Oreizy, P., & Dubrow, D. L. (1996) A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, vol. 22, no. 6 (June), 390-398. IEEE.

Travostino, F., Feeney, L., Bernadat, P., & Reynolds, F. (1998) Building Middleware for Real-Time Dependable Distributed Services. The First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (April 20) Kyoto, Japan. http://www.computer.org/conferen/proceed/isorc/8430/8430toc.htm.

Urnes, T. & Graham, T. Nicolas (1999) Flexibility Mapping Synchronous Groupware: Architectures to Distributed Implementations. Technical Report http://dundee.cs.queensu.ca/~graham/stl/pubs/dsvis99.pdf.

Urnes, T. & Nejabi, R. (1994) Tools for Implementing Groupware: Survey and Evaluation. Technical Report No. CS-94-03 Department of Computer Science, York University.

W3C (2000) W3C Architecture Domain: Hypertext Transfer Protocol. World Wide Web Consortium. http://www.w3c.org/Protocols.

Wang, W. & Marsic, I. (1999) DISCIPLE System Design. Technical Report CAIP Center, Rutgers University. http://www.caip.rutgers.edu/~weicong/DISCIPLE/doc/DISCIPLE3.doc.

Wang, W., Dorohonceanu, B., & Marsic, I. (1999) Design of the DISCIPLE Synchronous Collaboration Framework. Proceedings of the IASTED International Conference: Internet and Multimedia Systems and Applications (October 18-21)

Wu, D. (1997) Archive Server for Real-Time Collaboration in Disciple. Masters Thesis (October) Graduate Program in Electrical and Computer Engineering, Rutgers University. http://www.caip.rutgers.edu/~marsic/Publications/students/dawei/thesis_ps.

Xerox (1981) The Remote Procedure Call Protocol. Xerox Corporation XSIS 038112 (December)

Xing (1999) OpenMOM. http://www.xing.com/prodinfo.html.

Young, M., Taylor, R. N., & Troup, D. B. (1988) Software Environment Architectures and user Interface Facilities. IEEE Transactions on Software Engineering, vol. 14, no. 6 (June), 697-708. IEEE.