Supporting Interactive Collaboration on the Web with CORK

Philip L. Isenhour, Mary Beth Rosson, John M. Carroll
Center for Human-Computer Interaction, Department of Computer Science, Virginia Tech
660 McBryde Hall
Virginia Tech
Blacksburg, VA 24061 USA
{isenhour, rosson, carroll}@cs.vt.edu

(Accepted for publication in *Interacting with Computers*, special issue on
Interfaces for the Active Web)

Abstract

The World Wide Web has served as a medium for collaboration since its inception. Web-based collaboration has, however, been dominated by systems supporting asynchronous activities such as sharing documents and participating in discussion forums. Supporting interactive, synchronous collaboration on the Web has proven much more challenging. In this paper we describe three of the challenges encountered in the context of supporting network-based collaboration among middle and high school science students: integrating synchronous and asynchronous modes of interaction, minimizing consumption of bandwidth, and adapting non-collaborative software components for collaborative use. We then present the Content Object Replication Kit (CORK), a toolkit for building interactive Java-based collaborative systems for use on the Web.

Keywords: synchronous and asynchronous collaboration, collaborative software for education, groupware architecture, Java object replication

Introduction

Over the last four years, the Learning in Networked Communities (LiNC) project at Virginia Tech has developed and deployed a variety of network-based collaboration tools in middle- and high-school classrooms. Our goal has been to support synchronous and asynchronous collaboration on group projects among students at the same or different schools, as well as between students and community mentors. Supporting computer-mediated collaboration across classrooms highlights several challenges that are relevant to Web-based collaborative systems in general. Three of these challenges are integration of (and smooth transition between) synchronous and asynchronous modes of interaction, conservation of bandwidth, and adaptation

of single-user software for collaborative use.

The Web has traditionally presented a very natural medium for asynchronous collaboration, as evidenced by the success of systems that support structured asynchronous discussion and document sharing. HyperNews, for example, supports threaded discussions on the Web (HyperNews, 2000). Basic Support for Cooperative Work (BSCW) supports discussion as well as a range of document sharing and version control features (Bentley et al., 1997). Domino, an extension of IBM's Notes product, also supports discussions, messaging, and database access from the Web (IBM, 2000). These systems provide interactive interfaces for working within shared workspaces, but support for synchronous interaction between users is delegated to external conferencing and application sharing tools.

Introducing external tools for synchronous interaction not only mitigates the Web's application delivery and platform independence advantages, but it also forces users to repeatedly switch back and forth between different sets of tools. Different user interfaces must be learned, and users must remember to transfer synchronously generated content back into the asynchronous tools if it is to be made accessible to absent or late-joining collaborators. In settings where group members have unpredictable collaboration patterns, this overhead can be considerable. The classroom is an excellent example of this: class periods at different schools tend to only partially overlap, and illnesses, field trips, fire drills, or other unexpected events turn planned synchronous interactions into asynchronous interactions, sometimes with very little advance warning.

Delivery of tools that support synchronous interaction within Web pages is a significant technical challenge. Technologies like JavaScript and Dynamic HTML (Netscape, 2000) add a degree of client-side interactivity to otherwise static pages, but communication between applications based on these technologies is still limited by the stateless, transactional nature of the Web's underlying protocols. JavaScript and Dynamic HTML applications also tend to lack the rich, direct manipulation user interfaces found in traditional desktop applications. Modern Web browsers also often rely on "plug-ins" to handle new types of data (e.g., streaming audio or video). While plug-ins do not have communication restrictions and can support rich user interfaces, they suffer from many of the same issues as external applications launched from the web: they are platform-dependent, require manual installation, and raise security concerns. Java applets have been proposed as a solution to these problems, but size, stability, and performance issues have limited Java's use in client-side applications. As the Java platform matures, this technology is becoming increasingly practical as a means for providing secure, platform-independent, Web-delivered tools that have rich user interfaces and flexible communication capabilities.

Bandwidth consumption is an issue for any Web-based application that provides dynamic content. Educational content often relies considerably on rich multimedia data such as images, video, or sound. Such data tends to be considerably larger than textual Web page descriptions, exacerbating the bandwidth problem. Web-delivered code for interactive user interfaces (e.g., Java applets) also consumes considerable network bandwidth. In collaborative work settings, the network is further burdened by the messages that describe and synchronize each collaborator's actions.

Network bandwidth limitations are particularly acute in educational settings: While Internet access is increasingly available in elementary, middle, and high schools, it is likely that the network resources available to these public institutions will always be less than that available to corporations or universities. The limits that this places on the bandwidth available for collaboration software are compounded by problems such as students' attention spans, and more significantly by limitations on time available for synchronous interaction during a given class period. Sluggish network interactions that would merely be inconvenient or annoying in other settings can be disastrous in the classroom, contributing to the chaos that often accompanies in-class group activity sessions.

Although the Web provides an infrastructure for collaboration, many of the activities that groups carry out are supported by familiar standalone packages (e.g., spreadsheets, word processors, graphics editors). Thus one feature of many Web-based collaboration systems is support for sharing of arbitrary document files (Bentley et al., 1997; Spellman et al., 1997). However, the paradigm of sending documents around assumes that all members of a group have access to compatible versions of the documents' source applications. When simultaneous work is required, the issues are even more complicated. Familiar desktop applications typically do not support synchronous multi-user access, and development of a new generation of collaborative applications would be an expensive endeavor. It might also lead to the development (and subsequent learning burden for users) of a new set of user interfaces, losing the benefit of immediate integration with ongoing computing practices (Grudin, 1994).

A software infrastructure that supports reuse of single-user software is particularly interesting in the context of collaborative applications for classroom use, because there is an opportunity to appropriate curriculum-specific courseware from many sources. Collaborative use of these packages may significantly enhance their value, particularly in the context of mentoring. For example, the Educational Object Economy Web site (http://www.eoe.org/) provides links to several hundred physics-related Java applets, along with thousands of additional applets related to other fields. Adding collaborative interaction techniques to these applets would enable student groups to explore the physics concepts together, rather than simply watching as one student or

mentor demonstrates the applet to the other participants.

Architecture

We begin by describing critical characteristics of groupware architectures in general, to establish both points of comparison and a common terminology for describing the architecture of the Content Object Replication Kit (CORK), our toolkit for constructing Java-based synchronous and asynchronous collaborative systems suitable for delivery over the web. An overview of the CORK architecture and several usage examples follow.

Architectures for Collaborative Software

Dewan (1999) describes a classification of synchronous collaboration architectures based on common user interface "layers" found in single-user (window-based) applications. These five layers are an extension of Smalltalk's Model-View-Controller architecture described by Krasner and Pope (1988). At the highest level is the "model" layer, which encapsulates the content data that the user views or manipulates. Below the model is the "view" layer, which implements the logic for presenting the model. Below the view layer is the "widget" layer, which implements specific user interface components (such as buttons and text fields) based on the view. The "window" layer aggregates widgets into a window for display on the lowest layer, the screen.

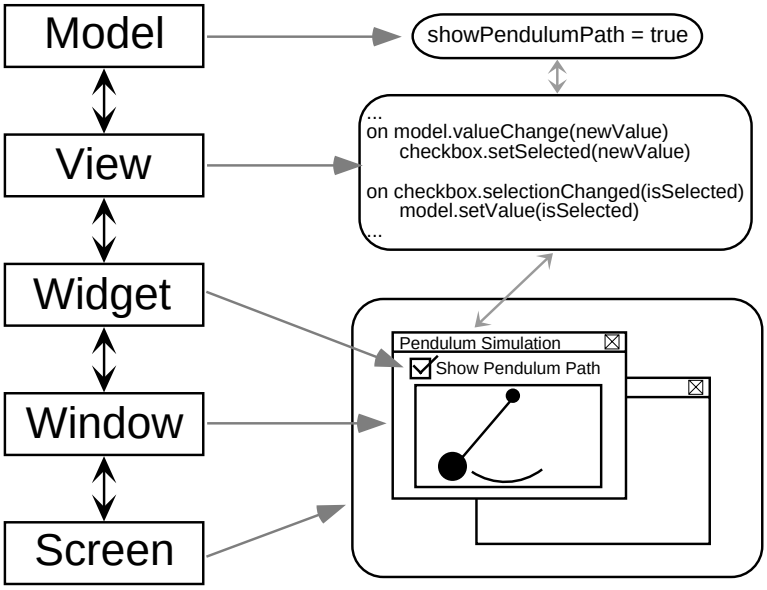

Figure 1. Architecture layers from Dewan (1999). Application data in the model layer is mapped to widgets by objects in the view layer. Widgets are displayed in windows on the user's screen.

Figure 1 shows a simple example of these layers. A boolean value in the model layer is mapped (by an object in the view layer) to a checkbox widget. The checkbox widget is contained in a window, which is displayed on the user's screen.

Depending on their design and implementation, some applications will not include all of these layers. In general, the most common layers are near the bottom of this list: Not all applications running on a particular window-based platform are designed with a separation between the model and view, but most (if not all) such applications make use of the same window and screen layers, since these are provided by the platform's operating system or window manager.

For distributed users to be able to collaboratively manipulate an application, some part of the application must be replicated on all participating machines. Groupware systems can be characterized by the layer at which this replication occurs. Typically the replication of one layer implies that all higher layers are not replicated, and exist on a single machine. Application-sharing packages like Shared X (Garfinkle et al., 1994) and Microsoft's NetMeeting (Microsoft, 1999) replicate the screen or window layers, allowing single-user applications to be simultaneously viewed and manipulated from different machines. These are typically referred to as *centralized* architectures, since the application logic executes on a single machine, with bitmaps of the window or screen sent to (and input events received from) the other machines participating in the session (Lauwers et al., 1990).

The layer at which replication occurs can have significant impact on bandwidth consumption. In architectures that replicate the window or screen layers, for example, transmitting bitmap changes may require considerably more bandwidth than transmitting the higher-level events that forced the those layers to change. If the model layer of the example in Figure 1 were replicated, a change to the `showPendulumPath` value could be represented in a relatively small message. If, however, the pendulum simulation was replicated at the window layer, the message describing the bitmap-level changes to the window when the pendulum path was shown or hidden could be quite large.

Architectures that support replication at the higher levels not only use smaller messages, but can also improve response time by allowing user actions to modify the replicated layers locally. Consider, for example, the effect of clicking the checkbox in the example from Figure 1. In a centralized system, a remote user who clicks the checkbox will not receive feedback until a message describing the mouse click is sent to the central copy of the application and the resulting changes to the window are sent back. If, however, the model layer were replicated, clicking the checkbox would change the local replica of the `showPendulumPath` value, immediately updating the view. A message describing the change to the model could be sent concurrent with

or immediately following the view update.

Because of network delays, users of a collaborative system will often have copies of replicated layers that are not fully synchronized with each other. In systems that replicate the lower layers (e.g., window or screen), this is less of a problem since only a single copy of the higher layers exists. If a user on a slow network connection invokes an operation based on an out-of-date image of an application window, the resulting behavior may not be as expected, but the application (and each user's snapshot of its interface) will eventually return to a consistent state. Where higher layers of the application are replicated, the problem is more difficult since users can simultaneously invoke conflicting operations on their copies of the replicated layers. Collaboration architectures that support replication of higher layers must provide means for resolving these conflicts.

A software component is collaboration-aware if it implements logic that is specific to collaborative interactions. As with replication, a groupware architecture can be characterized by the layer that contains collaboration-aware logic. Centralized systems, for example, provide collaboration-aware window or screen layers. Higher layers (widget, view, and model) then require no collaboration-specific logic.

Adapting single-user software for collaborative use requires that at least one architectural layer be made collaboration-aware. Collaboration transparency systems (Lauwers and Lantz, 1990) achieve this by dynamically replacing elements of a layer at application runtime. Application-sharing packages like NetMeeting and SharedX are collaboration transparency systems that implement collaboration-aware screen or window layers. These layers are used by nearly all single-user applications on a given platform, so this form of application sharing allows a wide range of familiar single-user software to be used collaboratively. This generality is, however, achieved at the expense of usability: All users get exactly the same view and only a single user at a time may interact with the application (Begole et al., 1999). Improving the flexibility of collaboration transparency systems is an open area of research, but recent work exploring the automatic transformation applications is promising. Flexible JAMM, for example, supports run time replacement of single-user views and widgets with multi-user counterparts. For widgets that do not have multi-user variants, Flexible JAMM provides input event distribution, so that interactions with the widget by any user are reflected in all other users' replicas, and a floor control mechanism to ensure that only one user interacts with a given widget at a time.

Collaboration transparency systems automatically adapt single-user software for collaborative use without modification to source code or executables, but require that the converted software use widget, window, or screen layers for which collaboration aware counterparts are available.

Where this is not the case, single-user software must be manually modified to introduce collaboration awareness. Many groupware toolkits have been developed that include collaboration-aware components at different architectural layers. DistView (Prakash and Shim, 1994) and Rendezvous (Patterson et al., 1990) provide collaboration-aware models. For Java-based groupware, Habanero (Chabert et al., 1998) and Java Collaborative Environment (Abdel-Wahab et al., 1997) provide collaboration-awareness at the widget layer. GroupKit includes collaboration-aware widgets for the Tcl/Tk scripting language, as well as components that support construction of collaboration-aware model and view layers (Roseman and Greenberg, 1997). The amount of effort required to convert a single-user application to use the components provided by these toolkits depends on the extent to which the behavior and programming interface of the components matches that of the components originally used in the application. Java includes a standard user interface widget library, so groupware toolkits for Java tend to provide multi-user counterparts for a widgets in this library.

In the example shown in Figure 1, the simulation could be shared with no modification by running it on a single machine and allowing others to access it using an application-sharing package like NetMeeting or Shared X. It could also be made collaborative by replacing the standard checkbox widget with a collaboration-aware checkbox that would notify all replicas of the simulation when any user changed the checkbox state. If a multi-user variant of the original checkbox was available, this replacement could conceivably be done automatically at run time. Otherwise, the simulation's source code would have to be modified to accommodate the collaboration-aware checkbox.

The layers at which an architecture supports replication and collaboration-awareness have significant implications for persistence of application state. Reconstructing application state at a later time (for late-joiner support or asynchronous interaction) typically requires that the higher architectural layers (e.g., the model layer) be restored. This is conceptually straightforward for architectures in which the model layer is replicated and collaboration-aware, since it is only necessary to maintain a persistent copy of one such replica for later access. For robustness, this copy can be independent of all interactive clients. The system could, for example, include a specialized client process that does nothing except maintain persistent copies of replicated data.

In architectures that only replicate the screen or window layers it is necessary for one participating client to take responsibility for saving the entire application state or, in the case of late-joiner support, transferring the replicated layers of the application to other clients. Support for asynchronous collaboration in these systems is largely beyond the control of the groupware architecture: The user who is executing the single instance of the application that contains the higher layers must remember to save the work and transfer it by some other means to the users

who will need it at a later time.

Architectures with replicated models but which implement collaboration awareness at the widget or view layers must either rely on one of the clients to manage state persistence (as with a centralized architecture), or must reconstruct state by saving and replaying the messages sent between the collaboration-aware layers. The former approach increases client complexity, while the latter can consume considerable space for long sessions. Message replay schemes can also be unreliable if the replayed messages were dependent on time or other external factors (Chung and Dewan, 1996).

Overview of CORK

The layers at which replication and collaboration awareness are implemented in a collaborative system represent significant design tradeoffs. Replicating higher levels may reduce bandwidth, but may also introduce synchronization problems. Implementing collaboration awareness at the lower architectural layers increases the opportunity for reuse, but may complicate persistence and late joiner support. These tradeoffs must be weighed in the context of specific applications. A toolkit that is to support construction of a broad range of collaborative applications must support replication and collaboration awareness at different architectural layers. CORK was designed to provide this kind of flexibility. In this section we give an overview of the five primary parts of the CORK architecture: replicated objects, listener objects, change objects, a central database, and permissions objects.

The fundamental goal of CORK is support for *replicated objects*: objects that are retrieved by multiple collaborating sessions and whose state is kept synchronized when any replica is changed. The Java platform includes object serialization capabilities, allowing most Java objects (those that do not encapsulate platform-specific information) to be written to and read from a stream of bytes (Sun Microsystems, 2000b). This means that a snapshot of an object can be captured on one machine, the serialized snapshot stored in a database or transferred using Java's networking support, and a copy of the original object reconstituted at a different place or time. This mechanism forms the basis for CORK's replication scheme, allowing a variety of objects in the model, view, and widget layers to be copied between collaborating sessions.

To maintain consistency across replicas of an object, local changes to the replicated object must be detected. Clients do this by attaching *listener objects* to each replicated object. Listener objects follow the Observer design pattern, receiving notification when the object they are attached to changes state (Gamma et al., 1995). This notification describes the state change, with the description typically encapsulated in an event object. Most standard Java classes used in

model, view, and widget layers support various types of event listeners. For example, a Java `Button` object supports attachment of `ActionListener` objects. These objects are notified and given an `ActionEvent` object when the button is pressed. Similarly, `Document` objects that represent the content of Java rich text editor components support attaching listeners that are notified when the document content or structure is changed. Complex objects support different types of listeners for different kinds of changes. It is therefore possible to attach listeners that are only notified about a subset of events that an object generates. (The term “listener” is taken from Java naming conventions. Objects assuming this role are also called “observers” or “handlers”.)

When a listener object detects a task-relevant modification to the replicated object it constructs a *change object* that is then broadcast to all other participants. Each change object includes data describing a modification, and also implements the logic for reproducing the modification on another replica of the object. When a change object is received on a remote client, CORK finds the local replica and “applies” the change to it by invoking the modification logic encapsulated in the change object. CORK derives much of its flexibility from this change abstraction technique, since changes provide a uniform method for propagating simple modifications to generic objects like lists and dictionaries as well as complex modifications to application-specific objects like rich text documents or virtual environment descriptions.

A *central database* provides persistent storage of all replicated objects. To initiate collaborative use of an object, a client retrieves a replica based on a query submitted to the database, or asks the database to create a new object and return a replica. Change objects broadcast by any client are received by the database and applied to the persistent copy of the modified object. CORK guarantees that the database contains a consistent copy of each replicated object at all times. This transparently supports both late-joiners to a session as well as fully asynchronous interaction, since clients retrieve a copy of a replicated object in the same way regardless of whether other clients are currently using replicas of the object or not.

Each replicated object may have security information associated with it in the form of a *permissions object*. The permissions object is consulted when a client attempts to retrieve a replicated object to determine whether the retrieval should be permitted. Permissions objects are also consulted before a change is applied to the database’s replica of an object. Permissions objects can choose to permit or deny all modifications attempted by a given user, or can inspect the change and deny only certain modifications. Clients can retrieve copies of permissions objects to determine, for example, whether some operations should be disabled in the user interface.

While permissions object implementations may be specialized for particular kinds of replicated

objects, this need not be the case. Similarly, the replicated objects do not need to know about their associated permissions objects. In practice a small number of generic permissions implementations are used, the most common being an implementation that restricts all access to an object to a specific user or group of users.

Using CORK

We first provide a detailed example of using CORK to replicate a single content object, and then describe a larger system we have recently deployed that uses many CORK-based collaborative components.

A CORK-Based Chat Tool

Text chat is a fundamental communication tool that, in the context of a long-term multi-party collaboration, has both synchronous and asynchronous interaction modes. For example, students who use the tool for cross-classroom collaboration will typically not enter at exactly the same time, and will need to "catch up" on the conversation if they are more than a few seconds late. Mentors, teachers, and other students who miss a conversation entirely will often benefit from reviewing the content of the chat session even after others have left. For those who participated in the chat synchronously, a persistent copy will be valuable as a transcript, perhaps serving as a record of task decomposition or other negotiation between group members. Finally, groups may decide to use the chat tool as a "message board" for fully asynchronous conversations. In this example we describe a simple chat tool that supports all of these modes of interaction.

When the user enters the chat they are presented with three widgets: a scrollable list of previous messages, an editable text box in which new messages can be entered, and a "send" button that adds a new message to the conversation. The chat tool's model layer contains an ordered list of message objects, which we refer to as the "message list". A replica of this list exists in the database and in each active client. Each message object in the replicated list contains the text and author of a single message. Our actual implementation of this tool also includes a timestamp for each message. The timestamp is not shown in this example.

Logic within the view layer updates the scrollable list of previous messages when a new message appears in the message list. When the "send" button is pressed, the content of the editable text box is added to the message list. Figure 2 illustrates the widget, view, and model layers.

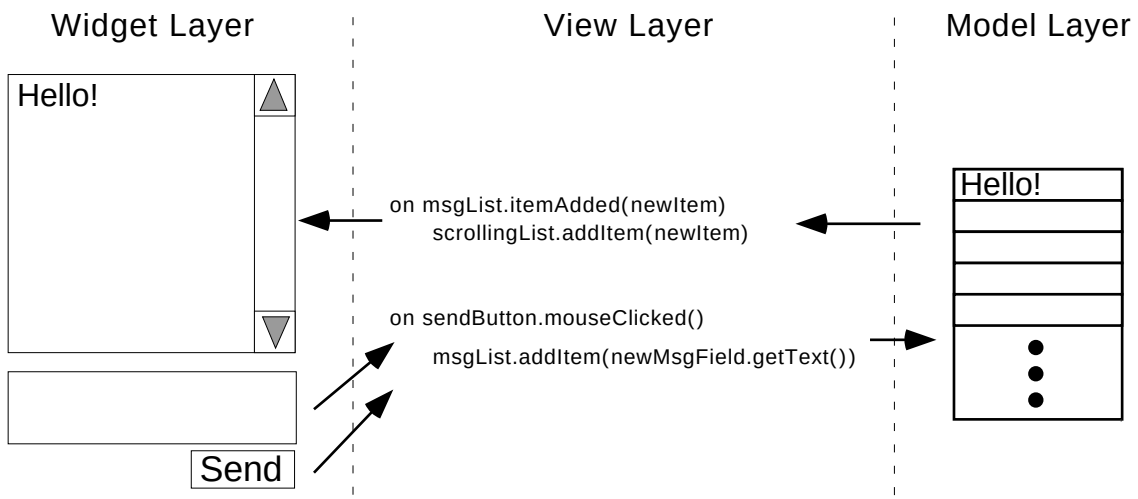

Figure 2. The widget, view, and model layers of a simple chat tool. Adding a message to the message list in the model results in the message being displayed in the scrolling list widget. Clicking the "send" button results in the contents of the new message field being added to the message list.

On startup, the chat client first retrieves a replica of the message list from the database. This replica will include all previously added messages. In the case of a late-joiner, some of these messages may be very recent. If no one is currently chatting, the messages may be older. From the perspective of the tool, however, these cases are identical. Figure 3 illustrates this scenario.

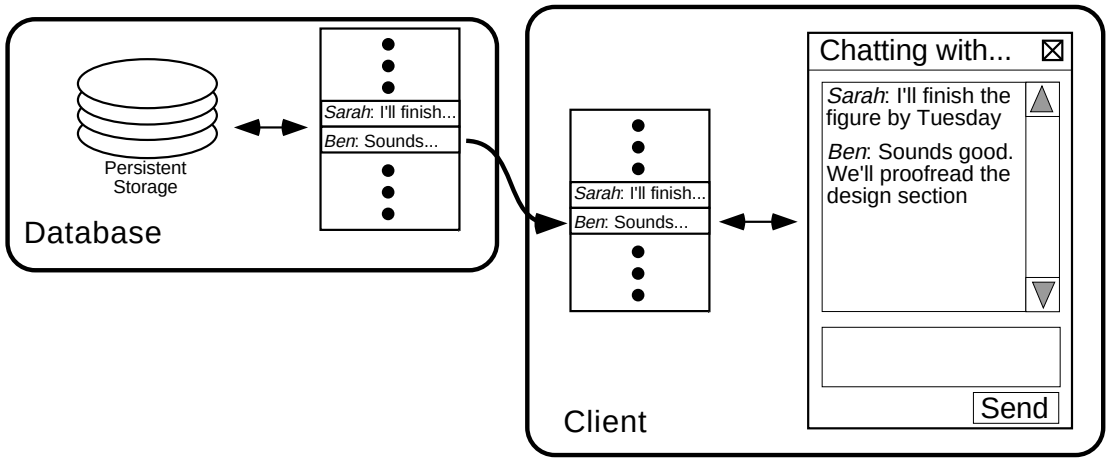

Figure 3. Message list replica transferred on chat tool startup. The replica will contain any messages previously added to the session.

This example assumes that the replicated message list already exists in the database. To initiate a new chat, any of the clients can request that the database create a message list with specific

permissions. A replica will be returned to the creator, and all other active clients to whom the associated permissions object allows access would be notified that the new object exists. As a result of such notification, a client might, for example, present the user with a dialog box inviting them to join the new chat. In this way CORK's security mechanism provides a replacement for "session" and "channel" constructs found in other groupware toolkits, since users can only discover and browse message lists for chats that their clients have permission to retrieve.

After the message list replica has been copied from the database, a listener object is attached to it. The listener detects additions to the message list and constructs a change object when such an addition occurs. The change object stores the content of the new message and implements the logic for adding it to another replica of the list. Figure 4 illustrates a listener and an instance of the type of change object that the listener would create when a new message is added to the list. Note that attaching the listener object does not change the behavior of objects in the collaboration-unaware widget, view, and model layers.

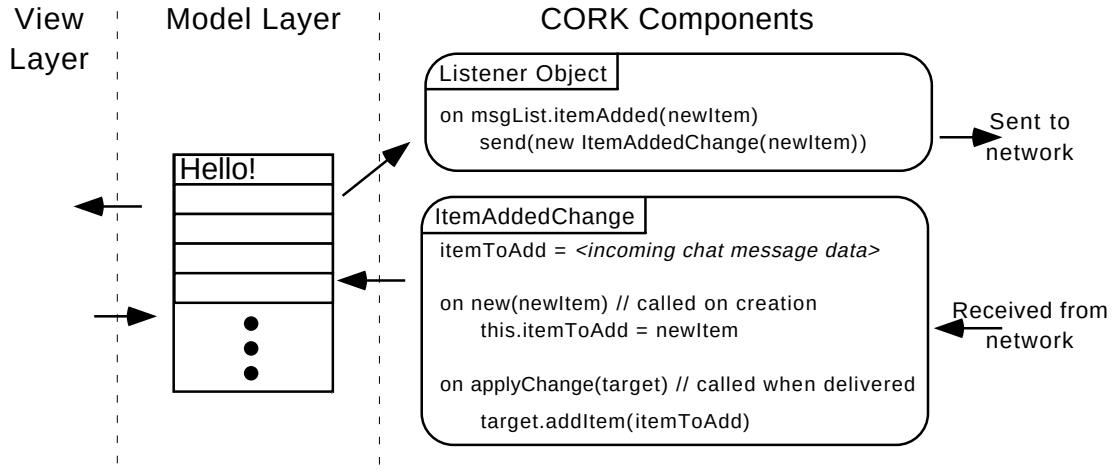

Figure 4. A CORK listener object added to the message list in the chat tool's model layer. Addition of a message to the message list triggers creation of a change object that includes the content of the new message and the logic for adding the message to a message list replica on another client.

Figure 5 illustrates what happens when the user enters a message and clicks the "send" button. Logic in the view layer adds the typed message to the message list replica. The listener detects the addition and constructs a change object. The change object is first sent to the central database, where the change is applied to the persistent copy of the message list. The CORK database implementation requires no knowledge of exactly what is happening when the change is applied: All application semantics are encapsulated in the change object. This means that the same server and database components can be used for any CORK-based application.

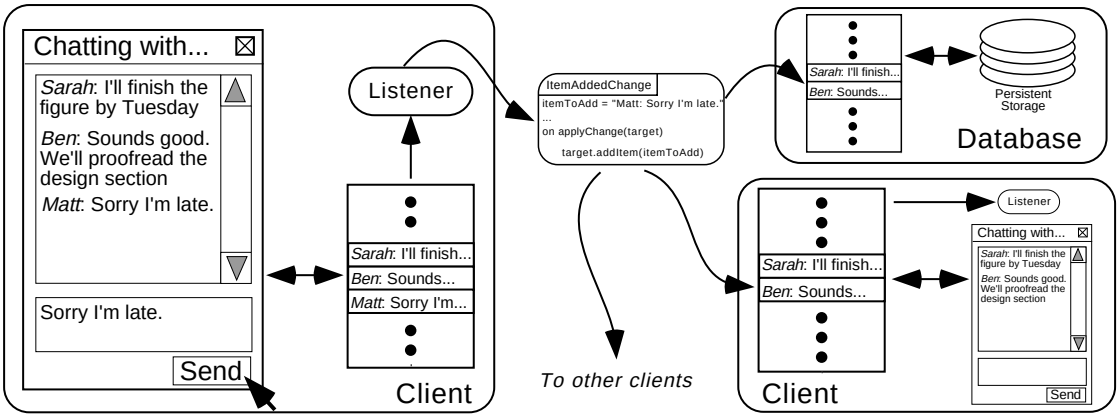

Figure 5. Synchronous use of the chat tool. A change object is generated by a listener object, then sent to the central database and any clients that have retrieved a copy of the message list. When received, the change is applied to the local replica.

If other clients have retrieved a replica of the message list then the change will be sent to those clients as well. When the change object adds the new message to each client's replica of the message list, the addition will be detected by logic in the view layer, updating the widget that presents the messages for viewing. The CORK listener object attached to the message list will also detect this message addition, but since this occurs in the same thread of execution that applied the incoming change, the CORK infrastructure can detect this situation and prevent message "echo".

Because the chat tool in this example replicates the model layer, it would be possible for two users to add new messages to their replicas of the same model at exactly the same time. This will produce an inconsistency, since the relative order of the two new messages will be different in the two replicas. The severity of such an inconsistency depends on the nature of the application. In the case of a chat tool, the relative ordering of two simultaneous messages may not be relevant. If consistent order is relevant, the change object could include more complex logic to detect and correct the inconsistency based, for example, on the order in which the two messages arrived at the central database.

In the description of this simple chat tool we have assumed that once a client has retrieved a message list replica it can add messages to the list without restriction. With a customized permissions object, however, CORK's security mechanism could provide fine-grained control over access to the chat. For example, it would be possible to allow different operations for different users, so that guest users could only view messages, normal users could view and contribute, and administrators could view, contribute, and modify messages.

Using CORK in the LiNC Virtual School

We have used CORK as the foundation for the "Virtual School" software developed by the Learning in Networked Communities (LiNC) project at Virginia Tech. Students in two high school and two middle school physics classes, as well as several mentors from the community, are using the software to collaborate on semester-long projects (Koenemann et al., 1998).

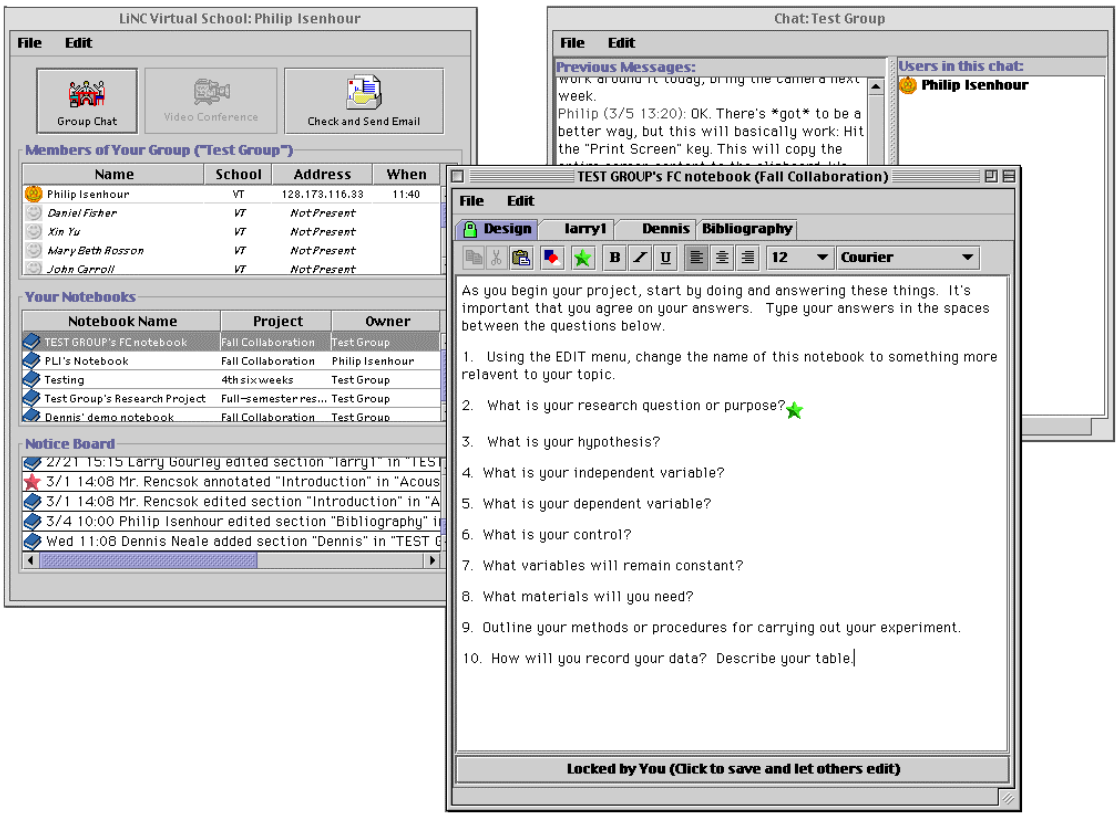

Figure 6. The LiNC Virtual School client software.

The Virtual School client, shown in Figure 6, includes a number of components that use CORK to provide asynchronous and synchronous interaction capabilities. These components include a user list, chat tools, a shared notebook, a notice board, and a collaborative Web browser. Providing a Web browser as a component of the Virtual School is a good indication of how we have extended the standard Web application paradigm: students can search for information on the Web, participate in Web-based discussions, even upload or download documents from the Web, but this is just one element of the diverse but integrated set of communication and collaboration tools. Providing a multi-user Web browser emphasizes the collaboration potential that has always been inherent in the Web.

The user list (upper left corner) shows which users are logged onto the system. It provides awareness of which group members are currently present, when they arrived in the system, and where they are logged in from. The underlying replicated object is a list of user data objects, each containing information on the identity, location, and activity of a user. A generic set of listener and change implementations support replication of this and other list-like objects.

Although third-party audio and video conferencing tools are integrated into the package, many of the students rely on text chat, primarily due to background noise in a typical classroom. Both group and private chat tools are available. (The window in the upper right corner shows a group chat.) Both tools use lists of chat messages similar to the one described in the previous example. The group chat also uses a shared user list to track who is participating in a chat session.

Students use shared notebooks both for intermediate work and to produce final reports for their collaborative projects. Shared notebooks are divided into sections, each of which can contain a different kind of data. The underlying data object for the notebook itself is a list of replicated object identifiers for accessing each section's data. Simple styled text sections are built on replicated `Document` objects the Swing user interface library, a standard part of the Java platform (Sun Microsystems, 2000a). A more feature-rich type of text section is also available, based on replicated HTML document model objects from the open-source KFC widget library (Yasumatsu, 2000). Bibliography sections provide access to replicated lists of reference records. Planner sections provide a view of replicated Swing `TreeModel` objects containing hierarchically structured information on tasks and subtasks. Whiteboard sections allow users to sketch using lines, rectangles, ellipses, and freehand drawings. A list of objects describing these shapes is replicated using CORK to provide collaborative functionality.

As with chat, the notebook supports both synchronous and asynchronous interaction. If two or more users open the same notebook at the same time, they will see each other's changes as they are made. (The exact granularity at which the modifications are propagated is based on conserving bandwidth. For example, we do not send a change representing every keystroke in a text section.) If a single user has the notebook open, then any change he or she makes will be made to the persistent copy of the content object in the database and will therefore be available to other users who open the notebook at some point in the future.

The notice board (at the bottom of the upper left window) describes modifications to the notebook and other shared data, providing users with asynchronous awareness information. The data displayed is stored in a replicated list, allowing the notice board to support both synchronous and asynchronous awareness: Users can check the notice board to see what has changed since their last login, but will also immediately see when new modifications are posted.

Additional tools to support synchronous awareness (e.g., through radar views (Gutwin et al., 1996)) are currently being added to the system.

A shared Web browser allows team members to browse the Web, maintain a shared bookmark list, and collaboratively annotate Web content. The replicated content object for this tool includes the current URL, a list of bookmarked URLs, a table of Web page annotations, and floor control information, allowing one user to take control of the browser to give a presentation.

Essentially all of the collaborative components of the Virtual School use replicated objects in the model layer, which can be presented to the user differently by different implementations of the lower architectural layers. The Virtual School client software provides a feature-rich interface for collaboratively manipulating these objects, while a separate Web interface provides lightweight, secure access to Virtual School content in situations where Java is not available or when editing capabilities are not needed. Users can go to the Virtual School web page, log in with their user id and password, and view pure HTML versions of notebook content, chat transcripts, and notice board entries from any web browser. A Java-based web server retrieves the appropriate content object replicas from the CORK database and renders them as HTML.

Discussion

In this section we discuss three primary goals of CORK's design: support for synchronous and asynchronous use, minimal bandwidth consumption, and support for reuse and adaptation of single-user components.

Synchronous and Asynchronous Support

CORK provides a unified mechanism for real-time updates, late-joiner support, and content persistence. The core components of the CORK architecture (replicated objects, listeners, change objects, the database, and permissions objects) are used the same way in applications intended to support purely synchronous interaction, purely asynchronous interaction, or a combination of the two. This allows developers to build or adapt software that supports each of these modes of collaborative interaction without having to implement separate mechanisms for each.

CORK's use of a central database for implicitly maintaining an up-to-date replica of each object is the critical feature that enables this functionality. The use of centralized state storage for replicated groupware architectures has been explored previously in systems like the Notification Server described by Patterson et al. (1996). The Notification Server provides storage of centralized state for collaborating clients in the form of key/value pairs. When one client changes

a value, the other clients are notified. The server treats the values as uninterpreted arrays of bytes. Clients are responsible for interpreting these values correctly, allowing the server to be implemented without knowledge of application semantics. The result is that the Notification Server can be used to store shared state for a wide range of groupware applications.

CORK leverages Java's dynamic class loading capabilities to support both centralized state and application semantics in the database. The objects stored in the database are of the same types (i.e., use the same class definitions) as those on the clients. The database can dynamically load application logic, in the form of class definitions, as needed. This makes the database implementation only slightly more complex than if it simply stored uninterpreted binary data, and prevents clients from having to perform interpretation of raw data values.

For users, the primary implication of CORK's unified support for synchronous and asynchronous modes of interaction is the transparency with which transitions between modes can be made. The mode of interaction becomes a function of how a tool is used, rather than a function of which tool is used. For example, collaborative activity within a Virtual School notebook or shared web browser is synchronous because two or more users access the tool at the same time, not because they explicitly opened it under an application-sharing package.

Minimizing Bandwidth Consumption with Lightweight, Flexible Change Objects

The change objects used by CORK to update replicated objects permit transmission of the minimum amount of data necessary to describe a modification, with the change object’s logic ensuring that the data is used to properly update remote replicas and resolve conflicts. The implementation of the change object’s logic (in the form of a Java class file) only has to be transferred to each client once, and is subsequently used for all instances of a given type of change.

Change objects used in a collaborative text editor, for example, could include information about each keystroke (or each set of continuous keystrokes), along with position information. The logic implemented in the change could handle resolving the position information at each remote replica of the editor’s content based on other modifications that had been made in parallel. The code that implements this logic would have to exist at each client, but would likely be downloaded as part of a compressed archive when the text editor is first accessed.

Depending on the complexity of a given type of replicated object, optimizing change object implementations to minimize network usage can require considerable development effort. In these cases, CORK also supports update mechanisms that use more bandwidth but are easier to implement. We have, for example, developed a simple replicated wrapper object that implements

a token-based locking scheme for sharing an arbitrary object. Any number of clients can retrieve the wrapper and read the contained object. A client can also request the wrapper's lock and, when the lock is granted, push a new version of the contained object to all other replicas. In the context of a collaborative text editor, this mechanism could be used to allow multiple users to simultaneously view a document in the editor and allow a single user at a time to have write access. A revised version of the document could then be pushed to other users when the controlling user releases the lock.

The flexibility to trade design complexity for network efficiency has allowed us to quickly integrate third-party (single-user) tools into the Virtual School and other CORK-based collaborative systems. When an interesting tool is found, support for replicating the state of the tool can often be quickly implemented using a bulk-transfer approach. If the tool proves to be successful, optimized change objects can be implemented to provide both more efficient network usage and more flexible collaborative interactions.

Supporting Reuse and Adaptation of Non-Collaborative Components

A large number of Java applets and applications are currently available on the Web. In addition to applets designed as courseware, there are more generic tools that support design, analysis, and authoring tasks. As the stability and performance of the Java platform improves, the number and sophistication of these tools will increase.

A number of toolkits and libraries have been developed to support creation of Java-based software for synchronous collaboration. Habanero and the Java Collaboration Environment are two such toolkits that provide replacements for standard user interface widget classes from Java's Abstract Windowing Toolkit (AWT). Applications use or extend these toolkit-specific replacements rather than the standard Java classes. The custom classes provided by the toolkits handle distributing local user interface events to remote participants in a synchronous collaborative session. Similar approaches are used in toolkits for systems written in other languages. For example, DistView for NeXTStep and COAST for Smalltalk (Schuckmann et al., 1996) provide classes for building collaboration-aware model and view layers.

CORK takes a different approach, providing infrastructure for attaching collaborative capabilities to collaboration-unaware objects in the model, view, or widget layers of an application. Adding collaborative functionality by composition rather than by replacing or modifying existing implementations offers considerable flexibility compared to toolkits that, for example, provide custom, collaboration-aware replacements for common Java AWT classes. Systems that provide such replacements potentially require that code be re-written to use the new

superclasses, presenting a problem if source code is not available. Collaboration transparency systems such as Flexible JAMM perform this replacement dynamically at runtime, significantly reducing the effort required to produce a collaborative application. However, the full benefits of collaboration-awareness at the widget layer can be provided only when collaboration-aware versions of the specific widgets used by the application exist.

A more difficult problem for toolkits that provide collaboration-aware replacement classes is that they must be updated whenever the APIs that they replace change. They will also often have to provide implementations for several different versions of the replaced APIs that may be in widespread concurrent use. This problem is particularly acute for systems based on maturing technologies such as Java. Slightly different implementations of standard classes are included with different Web browsers and new versions of rapidly evolving components tend to be incompatible with older ones—leading to a condition of "continuous obsolescence" (Brown et al., 1999). In contrast, listeners of the type used by CORK-based applications tend to rely on component *interfaces* (in the general sense of "sets of methods") rather than the class *implementations*, and are therefore less likely to be made obsolete when APIs change.

In implementing the Virtual School and other CORK-based systems we have developed listeners and change objects that can be used to replicate instances of the `ListModel`, `TableModel`, and `Document` classes in Sun's Swing user interface library. These implementations can then be reused to replicate lists, tables, and editors in any application that uses the corresponding Swing widgets, even if the application uses custom subclasses of the standard widget or model classes.

While standard components (such as those in the Swing library) are likely to be widely used, third-party libraries often provide better functionality for specific tasks. The primary component used for text editing in the Virtual School notebook is based on the KFC widget library, an independent, open-source development effort. Replacements for components from libraries such as KFC are unlikely to be included in collaborative toolkits. With CORK, however, we were able to replicate the model layer of the KFC text editor component, making it suitable for synchronous and asynchronous collaborative use.

Another illustration of CORK's generality is a generic JavaBean (Sun Microsystems, 2000c) replicator, which can be used to propagate atomic property changes to any Java object which implements JavaBean-style bound properties. This makes trivial the replication of objects that consist of a fixed set of atomic fields. We have, for example, used this technique to adapt applets that implement physics simulations, such as a Venturi tube simulation. Users can interactively adjust the radius and airflow through a tube to see the resulting pressure at different

points. The state of the simulation is represented in a Java object (a “bean”) with two numeric properties, one for tube radius and one for rate of airflow. Using the CORK-based generic bean replicator, multiple remote users can interact with the simulation and see each other’s changes. The change objects used to propagate modifications contain the name of the property that was modified and the property’s new value, but include no knowledge of any specific bean implementation. When applying these changes to remote replicas, logic in the change object uses standard tools in the JavaBeans package to set the property to the new value.

Conclusions and Future Work

The Web has always been a medium for collaboration. As bandwidth increases and technologies such as Java mature, opportunity and demand will increase for tools that support richer, more flexible collaborative interactions than traditional Web-based forums and document-sharing tools have provided.

The use of separate tools for synchronous and asynchronous interaction presents significant usability issues, as users must learn and switch between different interfaces. Existing tools to support adaptation of single-user software, however, tend to require considerable bandwidth, support relatively inflexible collaborative interaction, or require modification of existing software implementations. Even if new technologies increase available bandwidth, schools and homes will likely always have less capacity than corporations and universities. Collaborative applications deployed in these environments must therefore be designed to consume minimal bandwidth.

In this paper we have described the techniques that CORK uses to address these issues. CORK provides uniform support for synchronous and asynchronous manipulation of replicated objects, reducing development effort required to support both modes within a single application. The change objects that are used to update remote object replicas can include only the minimal amount of data necessary to describe a modification, thereby conserving bandwidth. CORK supports attachment of collaborative functionality to otherwise unmodified classes. The ability to add collaboration support through composition rather than replacement or extension simplifies reuse of existing software and supports collaborative use of classes whose implementations evolve over time.

The Virtual School software has been deployed and actively used since October 1998. MOOsburg, a more recent project, is a web-based graphical virtual environment implemented with CORK and employing the paradigm of a MOO (object oriented multi-user domain) for navigation and interaction (Carroll et al., 2000). We are also working with research groups from

the U.S. Navy to explore shipboard applications for CORK-based collaborative technologies.

Current work on the CORK infrastructure will address the potential performance bottleneck presented by the database. Systems of federated servers and redundant databases are possible solutions to performance and reliability issues as larger and more complex applications are built using CORK. We are also exploring ways to extend CORK to support intermittently connected clients, both to enhance reliability in the presence of network failures and to better support CORK-based applications on mobile devices.

Acknowledgements

Support for this work has been provided by the National Science Foundation under grant REC-9554206.

References

Abdel-Wahab H., Kvande B., Kim O., and Favreau J.P. (1997). An Internet Collaborative Environment for Sharing Java Applications. In *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97)*. IEEE Press.

Begole J., Rosson M.B., and Shaffer C.A. (1999). Flexible collaboration transparency: Supporting worker independence in replicated application-sharing system. *ACM Transactions on Computer-Human Interaction 6(2)*, pp. 95-132.

Bentley R., Appelt W., Busbach U., Hinrichs E., Kerr D., Sikkel K., Trevor J., and Woetzel G. (1997). Basic Support for Cooperative Work on the World Wide Web. *International Journal of Human Computer Studies: Special Issue on Novel Applications of the WWW*, Spring 1997, pp. 827-846.

Brown W., Malveau R., McCormick H., and Mowbray T. (1999). Continuous Obsolescence. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, pp. 85-86. New York: John Wiley & Sons.

Carroll J.M., Rosson M.B., Isenhour P.L., Van Metre C.A., Schaefer W.A., and Ganoe C.H. (2000). MOOsburg: Supplementing a real community with a virtual community. Submitted to the International Networking Conference, Plymouth, UK.

Chabert A. , Grossman E., Jackson L., Peitrowicz S., and Seguin C. (1998). Java Object-Sharing in Habanero. *Communications of the ACM* 41(6), pp. 69-76.

Chung G. and Dewan P. (1996). A Mechanism for Supporting Client Migration in a Shared Window System. In *Proceedings of the ACM UIST '96 Symposium on User Interface Software and Technology*, pp. 11-20. New York: ACM Press.

Dewan P. (1999). Architectures for Collaborative Applications. *Computer Supported Co-operative Work*, M. Beaudouin-Lafon (ed.), pp. 169-193. Chichester: John Wiley & Sons.

Gamma E., Helm R., Johnson R., and Vlissides J. (1995). Observer. *Design Patterns*, pp. 293-303. Reading, MA: Addison Wesley.

Garfinkle D., Welti W., and Yip T., (1994). Shared X: A Tool for Real-Time Collaboration. *Hewlett-Packard Journal* April 1994, pp. 23-24.

Gutwin C., Greenberg S., and Roseman M. (1996). Workspace Awareness Support with Radar Views. In *ACM CHI '96 Conference on Human Factors in Computing Systems, Companion Proceedings*, pp. 210-211. New York: ACM Press.

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

HyperNews (2000). HyperNews project home page, http://www.hypernews.org/

IBM (2000). Lotus Domino product home page, http://www.lotus.com/home.nsf/welcome/domino

Koenemann, J., Carroll, J. M., Shaffer, C. A., Rosson, M. B., and Abrams, M. (1998). Designing Collaborative Applications for Classroom Use: The LiNC Project. *The Design of Children's Technology*, A. Druin, (ed.). San Francisco, CA: Morgan-Kaufmann.

Krasner G. and Pope S. (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* 1(3), pp. 26-49.

Lauwers J.C., Joseph T.A., Lantz K.A., and Romanow A. (1990). Replicated Architectures for Shared Window Systems: A Critique. In *Proceedings of the Conference on Office Automation Systems*, pp. 249-260. New York: ACM Press.

Lauwers J.C. and Lantz K.A. (1990). Collaboration Awareness in Support of Collaboration Transparency: Requirements for the Next Generation of Shared Window Systems. In *Proceedings of ACM CHI '90 Conference on Human Factors in Computing Systems*, pp. 303-311. New York: ACM Press.

Microsoft (1999). NetMeeting product home page, http://www.microsoft.com/netmeeting/

Netscape (2000). Online JavaScript Reference Manual, http://developer.netscape.com/docs/manuals/communicator/jsref/index.htm

Patterson J.F., Hill R.D., Rohall S.L., and Meeks W.S. (1990). Rendezvous: An Architecture for Synchronous Multi-User Applications. In *Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work*, pp. 317-328. New York: ACM Press.

Patterson J.F., Day M., and Kucan J. (1996). Notification Servers for Synchronous Groupware. In *Proceedings of the ACM 1996 Conference on Computer-Supported Cooperative Work*, pp. 122-129. New York: ACM Press.

Prakash A. and Shim H. (1994). DistView: Support for Building Efficient Collaborative Applications using Replicated Objects. In *Proceedings of the ACM 1994 Conference on Computer Supported Cooperative Work*, pp. 153-164. New York: ACM Press.

Roseman M. and Greenberg S. (1997). Simplifying Component Development in an Integrated Groupware Environment. In *Proceedings of the ACM UIST '97 Symposium on User Interface Software and Technology*, pp. 65-72. New York: ACM Press.

Schuckmann C., Kirchner L., Schummer J., and Haake J. (1996). Designing Object-Oriented Synchronous Groupware with COAST. In *Proceedings of the ACM 1996 Conference on Computer-Supported Cooperative Work*, pp. 30-38. New York: ACM Press.

Spellman P., Mosier J., Deus L., and Carlson J. (1997). Collaborative Virtual Workspace. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work (GROUP '97)*, pp. 197-203. New York: ACM Press.

Sun Microsystems (2000a). Java Foundation Classes (JFC) home page, http://www.javasoft.com/products/jfc/

Sun Microsystems (2000b). Java Object Serialization specification, http://java.sun.com/products/jdk/1.1/docs/guide/serialization/index.html

Sun Microsystems (2000c). JavaBeans home page, http://www.javasoft.com/beans/

Yasumatsu K. (2000). KFC class library home page, http://openlab.ring.gr.jp/kyasu/.