GROUPDESIGN: SHARED EDITING IN A HETEROGENEOUS ENVIRONMENT

Alain Karsenty , Christophe Tronche and Michel Beaudouin-Lafon

Laboratoire de Recherche en Informatique (CNRS URA 410)
Université de Paris-Sud - Bâtiment 490
91405 ORSAY Cedex - FRANCE
(33) 1-69-41-65-91, ak@lri.fr
(33) 1-69-41-69-10, tronche@lri.fr
(33) 1-69-41-69-10, mbl@lri.fr

ABSTRACT

This article describes GroupDesign, a multi-user drawing tool that runs in a heterogeneous environment (a network of Apple Macintosh computers and Unix workstations). From the perspective of the users, we present a number of functions that we have developed for supporting the collaborative aspect of work - Graphic & Audio Echo, Localization, Identification, Age, History, Teleconference, Private Editing - and the new user interface issues raised by the shared editing of a document. From the perspective of the designers, we introduce the notion of purely replicated architecture and we describe the tools we have developed to implement this architecture in a heterogeneous environment. We also demonstrate the possibility of creating a multi-user application from a single-user one and address the issues in developing synchronous heterogeneous groupware.

KEYWORDS

Shared Editors, Synchronous Groupware, CSCW, User Interfaces, Purely Replicated Architecture, Distributed Systems, Engineering of Interactive Systems.

INTRODUCTION

Within the field of CSCW, a number of systems have been developed to support synchronous communication between geographically dispersed participants. Some systems provide teleconferencing facilities such as audio and/or video links, while other systems provide computer artifacts that can be shared by the participants. Shared editors fall in the latter category. A shared editor allows several users to edit simultaneously the same document (e.g. text, graphics, bitmap), hence supporting the work practices that we have developed when working together in the same location: shared usage of a blackboard, annotation of a paper document by several people, etc.

As an interactive system, a shared editor introduces a new dimension from the perspective of the user, namely the dimension of the group. An important consequence is that the user is no longer the only agent modifying the document. The document "modifies itself" when other users change it. With the exception of process control systems, few interactive systems exhibit such a behavior. In addition, whereas a process control system merely updates its state in response to changes in the environment, a shared editor must support a group activity. As a consequence, a shared editor must be designed to make each user aware not only of the modifications of the document but also of the activity of the other participants. As emphasized in [Tang91], a shared document not only stores information, it also mediates interaction.

The development of a shared editor raises a number of implementation issues. As an interactive system, a shared editor must feature a good response time, which means that the system must respond to a user's action in less than a tenth of a second. Since a shared editor is inherently a distributed system (the users are sitting at different workstations), the requirement for a short response time will be a key factor in the design of the architecture of the system. In addition, we see as essential the ability for the system to be used in a heterogeneous environment, i.e. in a situation where the different participants use different hardware and software platforms. A user working with a PC should be able to edit a shared document together with users working on Macintoshes and Unix workstations. This raises the question of knowing whether shared editing of a document is still a viable concept when the participants use different interfaces for editing the document.

In order to investigate these issues, we have designed and implemented a synchronous shared drawing tool called GroupDesign. The first version of the system worked on a network of Macintoshes [Beau92]. Our recent work has concerned the implementation of GroupDesign in a heterogeneous environment, namely Macintoshes and Unix workstations. The reasons for choosing a drawing tool are two-fold. First, most existing real-time groupware tools are pixel editors or text editors. Our tool is a structured graphics editor, which we feel is more representative of direct manipulation systems. Second, we did not want to create the system from scratch. Instead we used an existing single-user extensible drawing tool from each environment. This gave us an opportunity to understand the issues involved in turning a single-user application into a multi-user one and to get some insight into an appropriate architecture for synchronous groupware.

Many shared editors only support tightly coupled groups in situations such as brainstorming and design. These tools are not used to produce a document per se but instead a solution or set of ideas shared by the group. As a consequence most of these tools are bitmap editors [Stef87b, Minn91] that facilitate the creation and annotation of drawings. GroupDesign is not meant to be used in such situations. Instead it is meant to be used for collaboratively producing documents that happen to be schemas, e.g. Petri nets, SADT diagrams, PERT diagrams, organigrams. Hence this work is of interest to designers of shared editors as opposed to designers of meeting support systems.

This article presents this work from two perspectives: the users', and the designer's. From the perspective of the users, we present a number of functions that we feel necessary in any shared editor. This can be paralleled with the functions that have been identified for single-user systems, such as help, undo, semantic feedback, etc. From the designer's point of view, we demonstrate the possibility of creating a multi-user application from a single-user one, and we present the notion of a purely replicated architecture. In particular, we describe the issues involved in developing the heterogeneous version of GroupDesign and the implementation technique that we have used.

AN OVERVIEW OF GROUPDESIGN

GroupDesign is a multi-user drawing tool for structured graphics. It runs on Apple Macintosh computers connected by Apple LocalTalk or EtherTalk and on Unix workstations connected by Ethernet/IP. The architecture is replicated: an instance of the application (a *replica*) runs on the computer of each user.

A GroupDesign diagram is a set of pages, either independent or connected in a hierarchical way. Within a page, graphical objects (e.g. rectangles, texts) and connectors (i.e. links between objects) can be edited as in MacDraw. The complexity of a diagram is such that one can have a large number of users working on different pages. GroupDesign uses a relaxed WYSIWIS (What-You-See-Is-What-I-See) paradigm [Stef87a], since a strict WYSIWIS approach would not have allowed users to work independently on different diagram areas. The document is the same for all replicas but each user has his or her own view of the diagram. For example, users have independent control over the scroll bars and window placement.

GroupDesign sessions can last indefinitely with participants entering and leaving during the session. As a consequence, a user may enter a session without knowing the recent history of the document. If changes have been made to the diagram, and the user does not agree with the changes, he or she needs a means of identifying the user(s) who made the change to discuss it with them. The features we provide to address these issues are History, Age and Identification, which allow one to be informed of when, how and by whom changes have been made to the diagram. We have also developed a set of features to provide the group with a means of understanding simultaneous actions on the document. These features are Graphic & Audio Echo, Localization, and Teleconference. Finally, we have introduced the notion of Private Editing to support a more asynchronous editing style.

Most features use color to identify each user. The name of the users and their associated colors appear in a menu. The menu can be displayed permanently, so that there is no need to remember the colors. Moreover, we use light colors to avoid cluttering the display and interfering with the application’s use of color. Although it works fine for small groups, this color coding does not scale well: it is difficult or even impossible to generate a large number of visually distinct colors and memorizing/recognizing them puts an additional cognitive load on the user. Displaying the user's name provides an alternative to using colors. However this may raise other problems such as cluttering the display.

Graphic & Audio Echo

We define Echo as the representation of a user's action upon the other users' interfaces; it is the equivalent of feedback for the other users. The echo is graphic if users share a common view of the diagram or audio if operations occur out of the window. The goal of the echo is to provide a peripheral awareness of the activity of the group similar to that of several users working in the same office and mutually aware of each other. It must be comprehensible yet not too distracting so that one can choose to observe the group at work or decide to focus on a task without being disturbed. Showing small-grained changes suh as cursor motion is useful in situations where the group is engaged in a cooperative design task [Tang91] but proves distracting when the group works in a more loosely coupled way, which is frequent when editing large documents. This is why our echo is different from ordinary feedback.

Graphic Echo is a two-phase process, illustrated in figure 1. The first phase takes place when an operation is initiated. It displays an icon indicating that a change is about to be made by another user. This icon has the same function as the busy signal of the Colab [Stef87b]. The shape of the icon indicates which operation is underway and the color indicates the author of the operation. While the icon is displayed, the object is partially locked. A user can still modify the object unless the operations are not compatible, such as moving an object that is already being moved. The second phase of Graphic Echo takes place when the operation is completed and uses animation to make it easy to understand the operation.

	User	Group
Phase 1	presses the mouse on the rectangle.	"move" icon displayed on top of the rectangle. The icon has the user's color attribute.
	drags the rectangle to the new position.	the icon is still displayed.
Phase 2	releases the mouse. The rectangle is drawn to the new position.	the icon is erased and the rectangle is moved smoothly to the new position.

Phase 1: busy icon   Phase 2: animation

Figure 1 - Graphical echo

Audio Echo complements Graphic Echo, since the latter does not cover modifications occurring outside of the window view. A sound is associated with every operation that changes the document. We use two kinds of sampled sounds: percussive sounds for short operations (e.g. creation, destruction) and continuous sounds for long operations (e.g. move, resize). When possible, the sounds match the operations they describe. For example, we use a breaking sound for destruction, a scraping sound for moving and a spring sound for resizing. The sounds for other operations are more arbitrary; we chose them to be consistent with the "style" of the other sounds and yet easy to distinguish. In order to provide an effective peripheral awareness, the sounds must be easy to recognize even when played at a low level. Conveying more semantic information would be valuable. We plan to explore this further by using synthesized auditory icons [Gave93] and simple spatialization using a stereo output.

Both graphic and audio echo are modes that users can enable or disable. Users can also control the set of operations for which an echo is provided.

Localization and Identification

Localization makes it possible to coordinate views with another user. In this mode, the participants' front windows are displayed as rectangles in each participant's color so that users can see each other's current view (figure 2). This gives the users a sense of territory - when they modify objects, they can see whether other users are currently viewing the changes. Selecting a user's name from the menu “teleports” one's view to the area currently viewed by that user. Thus, one can both have an independent view of the diagram and be able to synchronize views with another user.

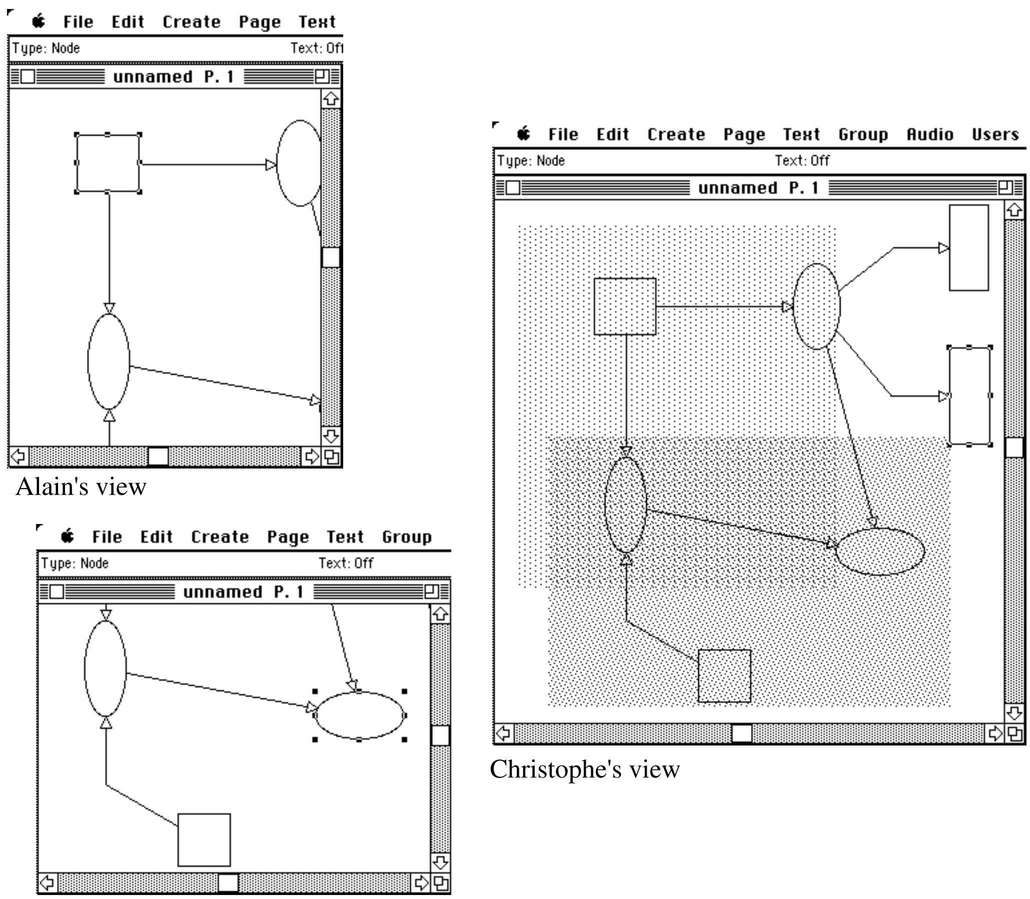

Figure 2 - Three users in the same session viewing an overlapping part of a page.
Christophe is in Localization mode and thus can see Alain's and Michel's views

Localization alone is not sufficient to identify who has just modified a diagram, since several users can view the same part of the diagram. Moreover, once the changes are done, there would be no means to identify each user's operations. The Identification feature is used to identify the users who modified the diagram. Identification is done through colors: every object is displayed with the color corresponding to either the user who created it or to the last user who modified it. This is useful for discussing design issues on a particular area of the diagram without disrupting the session participants. For instance, using Identification, one can contact the specific users who made the changes.

Both Localization and Identification are modes. They do not interfere with the drawing activity, but add information about users to the participants. While they are active, the display is updated in real-time to reflect the state of the users' locations and authorship as they change.

Other techniques have been used to locate other users [Gree92]. Gestalt views display the whole document in a small window together with the users' locations. A gestalt view has the advantage of displaying the locations of all users at once, while our Localization mode only displays the users working in the area being viewed. On the other hand, gestalt views require additional windows which clutter the screen and may distract the user. Another localization technique is view-slaving which allows a user to follow another user's view in real-time. View-slaving stands between our Teleport command and the Teleconference mode described below. We believe that Teleconference is more intuitive to use than several users view-slaving to each other, and that Teleport is valuable for peeking at someone else's view without having to slave to it.

Age and History

The only time-related information usually available concerning a diagram is the last time it was saved into a file. No information is given about the last date of modification of the objects in the diagram or the history of the document. In a multi-user application this information is important: accessing the history of the document has proved useful for supporting the cooperative process and bridging the gap between synchronous and asynchronous work [Rhyn92]. If many objects in the same area have been recently modified, this area is probably a hot spot of the group's activity. We provide two features to support such access to the document: Age and History.

In Age mode, the age of objects is displayed using colors which vary from red (recently modified) to blue, in a way similiar to GROVE [Elli91]. This allows one to find out when objects have been modified, but not how. This is why we also provide a History feature. The last actions of the group can be replayed using a control panel similar to a tape recorder. The actions and user’s names are displayed when an operation is replayed either forward or backward. Whenever one enters History mode, the system is off-line. It stores the other user's actions in a queue and processes them as soon as the user closes the control panel.

History is a command, which means the cannot change the diagram while replaying the last actions. On the other hand, Age is a mode. When it is active, the display is updated to reflect the age of objects as they are modified.

Teleconference and Private Editing

GroupDesign defaults to a tight coupling in time (all users see the same state of the document at any point in time) but no coupling in space (different users may view different parts of the document). Instead of providing a variety of coupling schemes as in Suite [Dewa91], we have chosen to provide two additional coupling modes: a tighter coupling called Teleconferencep and a looser coupling called Private Editing.

Teleconference allows one or several subgroups to work under a more strict WYSIWIS interface. This is useful when some members of the session want to work in a tightly coordinated way. This usually require additional communication channels such as audio/video links so that the users can coordinate their actions. In Teleconference mode, the users see each other's windows modifications (resize, scroll, etc.), but not the movements of the cursor. Ideally the cursors of the other users should be visible in order to support "gesturing" [Tang91], but this proved to be too difficult to implement given our software environment. Similarly the current version does not provide telepointers, although it is possible to use a GroupDesign object as a telepointer.

Private Editing relaxes WYSIWIS in time. In this mode, the user's modifications do not appear on the other participant's windows. When satisfied with the resulting diagram the user can decide to commit himself or herself and the other participant's diagram is then updated. This feature is useful in a variety of situations. For example, if a user is not familiar with the application, it might be inhibiting to show his or her clumsiness to the group. Another example is when a user modifies the diagram and is not satisfied with the result. He or she can cancel the modifications without disrupting the session. More generally, this mode gives the users a way to get some privacy during a session. Users in Private Editing mode are showed to the other users via the menu of participants.

ARCHITECTURE

Distributed groupware systems generally use either a fully centralized architecture or a partially replicated architecture. For example, the LIZA [Gibb89] toolkit uses a central process with replicated clients while RendezVous [Patt90] is a centralized system. In a centralized architecture, only one process interacts with the different users through a generic agent such as the X Window system [Sche86]. Partially replicated architectures also use a centralized process, but each user runs an application that talks to the central application-dependent process. This is similar to a client-server model. Depending on the systems, the division of labor between the server and the clients varies. At one extreme, the server can be used only for the serialization of events. At the other extreme, the server can manage most of the state of the application while the clients only manage the display and low-level interaction.

A centralized or moderately replicated architecture has the advantage of simplifying the developer's task since there is no need to manage a distributed state. In particular, synchronization and concurrency control are trivial. However there are two major drawbacks to such an architecture. The first is the unpredictable response time of the interface. If every user’s action has to go to the central process before giving any feedback to the user who initiated it, then the response time depends on the networks and the load of the central process. The second is that the system is not likely to be fault tolerant to server crashes. Both aspects contrast with single-user applications, whose performance and reliability only depend on the workstation on which they are running.

These drawbacks become of prime importance when one looks at the scalability of groupware systems: when shared editors become widely available, it is likely that (i) the numbers of participants per session will grow (ii) the geographical distribution and hence the number of networks involved in a session will grow and (iii) the overall number of sessions will grow. It then becomes crucial to devise an architecture that does not break down when the system scales up, and it is clear that a centralized architecture cannot work in such a context.

Purely Replicated Architecture

We have implemented what we call a *purely replicated* architecture. A replica of the application runs on each computer. GroupDesign does not use any central process for the coordination of the replicas nor does it give a special role to the user who first launches a session. All replicas play the same role, and no other process is required. As discussed later on, intermediate processes manage the flow of data for better efficiency under the Unix environment, but they are application independent and the system could run without them. On Macintoshes, the replicas communicate by sending each other events through LocalTalk, using the facilities of the Apple Event Manager of Apple's new System 7.0 [Appl91]. Under Unix, and between Unix and Macintoshes, the replicas send each other events with datagrams using the UDP protocol.

The events that a replica sends to and receives from other replicas allow it to maintain a state of the document that is consistent with the other replicas. More precisely, when no event is in between two sites, the states held by all replicas are identical. This is achieved by a distributed algorithm that implements the concurrency control needed to ensure this property.

This concurrency control algorithm is based on the semantics of the application, and the ability to define a total order of the events using timestamps [Lamp78]. The timestamp of an event is a logical clock, i.e. a counter incremented each time an event is sent and adjusted each time an event with a higher timestamp is received. The total order is defined by the order of timestamps, plus an arbitrary order of the replicas for events with the same timestamp. Ideally, every replica should receive the same events in the same order, and no concurrency control would be needed. But in practice, because of network latency, events may arrive in different orders at the replicas. Handling events in the correct order could be achieved by holding back events that arrive in advance while waiting for late events. However, this would be incompatible with an optimal response time since the local actions of a user could be delayed by his or her replica because of late events sent by other replicas.

Our solution [Beau92] is to execute the events in a partial order that is compatible with the total order, using the semantics of the application to define the compatibility relation. For example, if a user moves an object and another user changes its color, the order of execution of these actions is irrelevant: we say that the events carrying these action commute. On the other hand, if a user changes the color of an object to red and another user then changes it to green, the corresponding events do not commute. However, if ‘change to green’ has been received and executed by a replica and ‘change to red’ arrives later, the latter can simply be discarded: we say that the second event (in the total order) has masked the first one. In GroupDesign, events always commute or mask. Therefore they are always handled immediately, which provides the best response time possible for the interface. [Kars92] describes a more general version that handles incompatible events, i.e. events that do not commute nor mask.

Our technique is simpler than the operation transformation used in GROVE [Elli90]. This technique consists of transforming out of date operations so that they can be executed without disrupting the session. If there are n operations, this requires n^2 transformation procedures, some of which are not trivial to write. In our case, one only needs to define an n^2 matrix describing whether each couple of operations commute, mask, or are incompatible.

The purely replicated architecture combined with our concurrency control algorithm give the system two important properties. First, the user's actions are handled immediately by the system, resulting in the best possible response time, namely the response time of a single-user application. In addition, incoming events are handled immediately, giving the most up-to-date possible view on the document. Second, the system is fault-tolerant: a replica continues to work even though another replica crashes or the network goes down.

This contrasts with a number of existing systems. For example, MMConf [Crow90] has an architecture close to ours, but it encourages turn-taking floor control because when running an open floor, the events are not guaranteed to arrive in the same order at all sites. DistEdit [Knis90], a toolkit for programming multi-user text editors, relies on the ISIS toolkit [Birm89] for the distributed aspect of the architecture. This may cause performance problems, as stated by the authors, because of the concurrency control algorithms implemented by ISIS. Aspects [Biel91], a commercial product for shared editing, locks objects as soon as they are selected. This certainly simplifies concurrency control but reduces access to the document since a user not currently working may have inadvertantly selected objects.

Heterogeneous architecture

The purely replicated architecture is suitable for implementation in a heterogeneous environment. First, the overall performance of the system only depends on the performance of the network. More precisely, a replica connected with a low-speed network does not slow down the whole system. Instead, its actions are seen by the other users with a longer delay. Hence the system does not run at the speed of its slowest component. Secondly, the architecture only requires the replicas to comply with the protocol of the application. As long as network connectivity can be assured, the replicas can run on different hardware/software platforms, and they can be developed either from scratch or on top of existing toolkits or applications. Indeed this is how we implemented GroupDesign in both environments (Macintosh and Unix/XWindows).

We found that in practice the implementation of a distributed heterogeneous system was more complicated than expected. We now describe the main reasons for this: inappropriate software tools, complexity of data transfer and routing, variety of naming schemes and data formats.

Widely available operating systems and computer networks provide little support for implementing distributed systems [Patt91]. For example, the Unix operating system and the Ethernet network cannot guarantee bounded time execution. Also, most existing tools for developing distributed systems are either not widely available, too low-level, or not suitable for developing groupware. Most systems were developed for applications such as file and message transfer. They support point-to-point communication and focus on reliability, fault tolerance and authentication, whereas groupware requires multipoint communication and efficient transfer of usually small amounts of data. The situation is even worse in a heterogeneous environment, where one can only rely on the network connectivity at a very low level. As a consequence we had to write our own tools and implement our own communication schemes.

The performance of the network is critical for heterogeneous groupware since, roughly speaking, the speed of a message along a path in the network is the speed of the slowest subnet. With the purely replicated architecture, a balance must be found between the number of links a message traverses and the number of messages (frames) transmitted through the net. Increasing the number of links slows the transmission down (users are then more loosely coupled) but it allows for greater flexibility and better usage of network resources. On the other hand, minimizing the number of links traversed by a message means that each replica must send its messages to all other replicas, since multicasting is not available on the most widespread network protocols (both AppleTalk on the Macintosh and TCP/IP only provide point-to-point communication). The load of the network then grows quadratically with the number of replicas. This is acceptable only on a local area network with a small number of users.

In a heterogeneous environment the situation is more complex, since different incompatible subnets are connected via gateways or routers. Using the same multicasting schemes (sending each message to all other replicas) wastes network bandwidth since several identical messages sent by a replica to recipients on the other side of a gateway or router traverse the same subnet. We addressed this problem by introducing special processes called *dispatchers*. The dispatchers avoid duplicated messages on common subnets, hence improving the overall performance and network load. The dispatchers do not introduce a bottleneck. Even though they increase the latency (each packet now goes through at least one additional process) we found in practice that this additional lattency is negligible as long as the network of dispatchers is correctly configured. The typical round-trip time on an Ethernet cable is one to four milliseconds, and a dispatcher needs less than a tenth of a millisecond to handle a packet. Hence, on a LAN the bottleneck is not from the dispatchers but from the replicas themselves: a dispatcher can handle more packets per second than the network can transmit whereas the replicas have to process each event. This processing turns out to be especially expensive as soon as it includes graphics. These results confirm the observations of [Gree92] that in a real-time groupware system, communication is less of a problem than processing power. However, adding dispatchers makes the system substantially more complex since the best performance is achieved when they follow the toppology of the network, which might change dynamically. On the other hand, since the dispatchers handle all the complexity of multicasting in a heterogeneous environment in an application-independent way, they simplify the replicas: a replica only has to send events to its dispatcher.

Finally, we have to address the variety of naming schemes and data formats. A mechanism to address any host on a network always exists in a homogeneous net. We need a similar functionality in a heterogeneous environment, even when the naming conventions are not compatible. This raises non-trivial problems: connecting two nets with the same name space through a third net may result in a name collision. We solved this by a global name server. The problems with data formats are different, but not simpler. Applications exchange various kinds of data, e.g. text, picture, sound, live video, using various formats. Most of them are specific to a machine, an operating system, or even an application. Working in a heterogeneous environment requires converters for the different formats (assuming that the formats to be converted are documented). One then has to choose between developing a quadratic number of converters, or define a new format and a linear number of converters. Since we have only two platforms, we used the first approach.

We are not aware of any shared editor or groupware system that runs in a heterogeneous environment. The existing systems run on one of the following hardware/software platforms: Macintosh under MacOS, PC under Windows or Unix workstation under XWindows. Although some interoperability between different systems does exist, this does not achieve support of a heterogeneous environment. For example, a Macintosh can run an X Window server and hence allow users to run X applications on their displays. But then the users have to use the look and feel of Macintosh applications and XWindows applications at the same time, which is not desirable.

IMPLEMENTATION

GroupDesign is implemented in the following heterogeneous environment:

	Unix	**Macintosh**
Hardware	Unix workstation	Apple Macintosh
Network	IP	AppleTalk
Intermediate Protocol	UDP	AppleEvents
Operating System	Unix	MacOS
Graphical system	X11	Quickdraw
Programming interface	Graph Widget	Design/OA

The environment is as heterogeneous as possible, since each layer is different and incompatible. To handle this environment, we have developed an implementation model based on the concept of actor. The programming interfaces provided us with the needed flexibility to implement such a model. In the next section we describe both the programming interfaces and the model.

Programming interfaces

Rather than building GroupDesign from scratch or modifying an existing program, we used existing toolkits and extensible applications. On the Macintosh platform, we used Design/OA to develop an extension of MetaDesign [Meta89]. MetaDesign is a MacDraw-like editor with two significant differences: a document is a set of pages, which can be linked together, and the drawing on each page can have connectors, i.e. links between objects. Under Unix, we used the Graph Widget [Beau91], a Motif widget that allows interactive editing of arbitrary graphs.

The programming interfaces of Design/OA and the Graph Widget both offer two sets of functions: the first set contains functions to create and modify a graph, providing the application with a similar set of functions as the user for editing the graph; the second set contains a set of functions to define so-called filters (in Design/OA) or callbacks (in the XToolkit). Filters and callbacks give the application control over the actions of the user: when the user issues a command, a filter or callback defined by the application is activated. A number of non-groupware applications have been implemented with these programming interfaces. For example, an extension of MetaDesign is a Petri-Net simulator, and an application that uses the Graph Widget is a file system browser.

In GroupDesign, we used the first set of functions to handle incoming events from the other replicas, and we used the second set of functions to send events to the other replicas whenever changes are made by the user. Hence, the protocol was defined so that the events could be matched to both set of functions in both environments. This proved to be relatively easy, compared to the task of managing the communication between the replicas, and the limitations of both Design/OA and the Graph Widget for implementing some of the groupware features, which we discuss in a later section.

Implementation model

The implementation model for the different processes that constitute our distributed system is based on the concept of actor, as first introduced by Hewitt (see for example [Agha87]). This approach is quite natural with the purely replicated architecture: each replica is an instance of the same actor model. We describe in this section the Reactive Engine, a C++ library that we have developed to implement our actor model.

The reactive engine provides high-level primitives to describe communication with and naming of remote actors or group of remote actors. An actor is an instance of an actor model (or actor class). An actor model has two sections: a specification which describes the name of the actor and the messages it can respond to, and an implementation. An actor is known to the outside world only by its specification. For example, figure 3 describes the specification of an actor model named UpdateDisplay, with two methods CreateNodeAtX:Y: and DragNodeToX:Y:. We use a Smalltalk-like convention for the method names; the colons indicate arguments which are untyped character strings. From this specification are derived two implementations (figure 4); one is specific to Design/OA while the other is specific to the Graph widget. The bodies of the methods are written in C++. The definition of a single specification enforces consistency between the different actors created on the same model, even if they run on incompatible machines. Of course, we cannot ensure that the bodies of the implementation are equivalent. This is not even desirable since the replicas are only required to conform to the protocol; they can interpret the messages with different semantics, if they so wish.

```
actor UpdateDisplay
   CreateNodeAtX:Y:
   DragNodeToX:Y:
```

Figure 3 - Specification of an actor

```
actor body UpdateDisplay
    // C++ code calling Design/OA functions
    CreateNodeAtX:Y: {
         DScr_node(...);
    }
    DragNodeToX:Y: {
        DSmove(...);
    }
```

```
actor body UpdateDisplay
    // C++ code calling Graph Widget functions
    CreateNodeAtX:Y: {
        XtCreateWidget (...);
    }
    CreateNodeAtX:Y: {
        XtSetValues (...);
    }
```

Figure 4 - Two implementations of the same specification

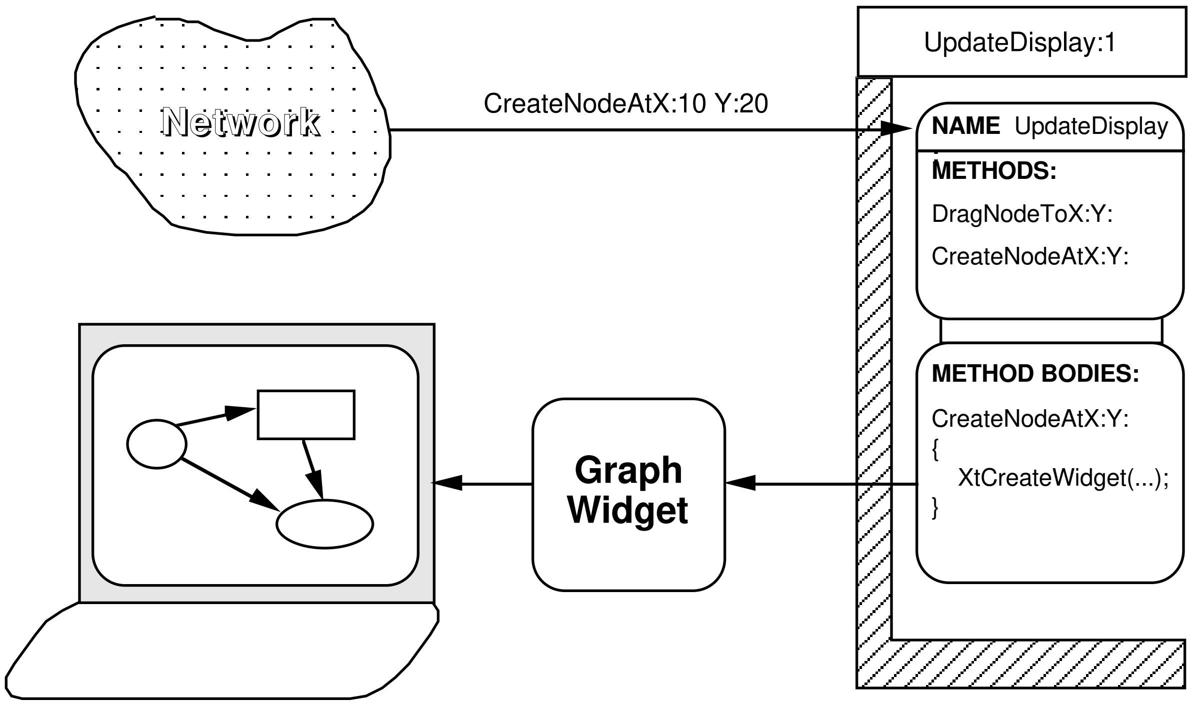

Figure 5 - A running instance of an actor

A pre-processor reads the specification and the method bodies and then produces an equivalent C++ program. After compilation by the C++ compiler, each running process of this program is an actor instance, such as the UpdateDisplay:1 instance depicted in figure 5. The instance receives and sends messages through the reactive engine. For example, in figure 5, the method CreateNode of UpdateDisplay:1 has been activated by a message from a remote actor. UpdateDisplay:1 then calls a function of the Graph Widget to update the display.

It is the responsibility of the application to invoke the methods of distant actors. In GroupDesign, the callbacks of the Graph Widget and the filters of Design/OA call actor methods, so that modifications of the graph by the users are multicast to other replicas by the reactive engine. This is exemplified by the following statement:

```
localDispatcher-> send("CreateNodeAtX: %d Y: %d", xcoord, ycoord);
```

Some appropriate C++ objects, such as localDispatcher in this example, are used as stubs for the remote actor or remote actor group. When first elaborated, an object of this class queries a name server for the nearest dispatcher. The name server has a similar role as the IP Domain Name System [Mock87], except that it can store AppleTalk addresses as well as IP addresses. An actor automatically notifies the name server of its existence when entering its top level loop. The local dispatcher then puts the actor on its list of actors to which the incoming messages have to be dispatched.

Figure 6 shows a complete GroupDesign session. Since we use our purely replicated architecture, the session contains one actor for each user, implementing the replica of GroupDesign. In addition, the session uses several other actors: the name server and the dispatchers. The dispatchers implement an efficient multicasting facility in a heterogeneous network, similar to the Multicast Router [Deer89]. The dispatcher's name (which defines the session name) is the only information an actor needs to know in order to send a message to every actor in the session. The actor sends the message to the dispatcher and the dispatcher delivers the message to the targeted actors, possibly by sending it to other dispatchers. Since the name server and the dispatchers are application-independent and are used as daemon processes, the application developer need not be aware of their existence.

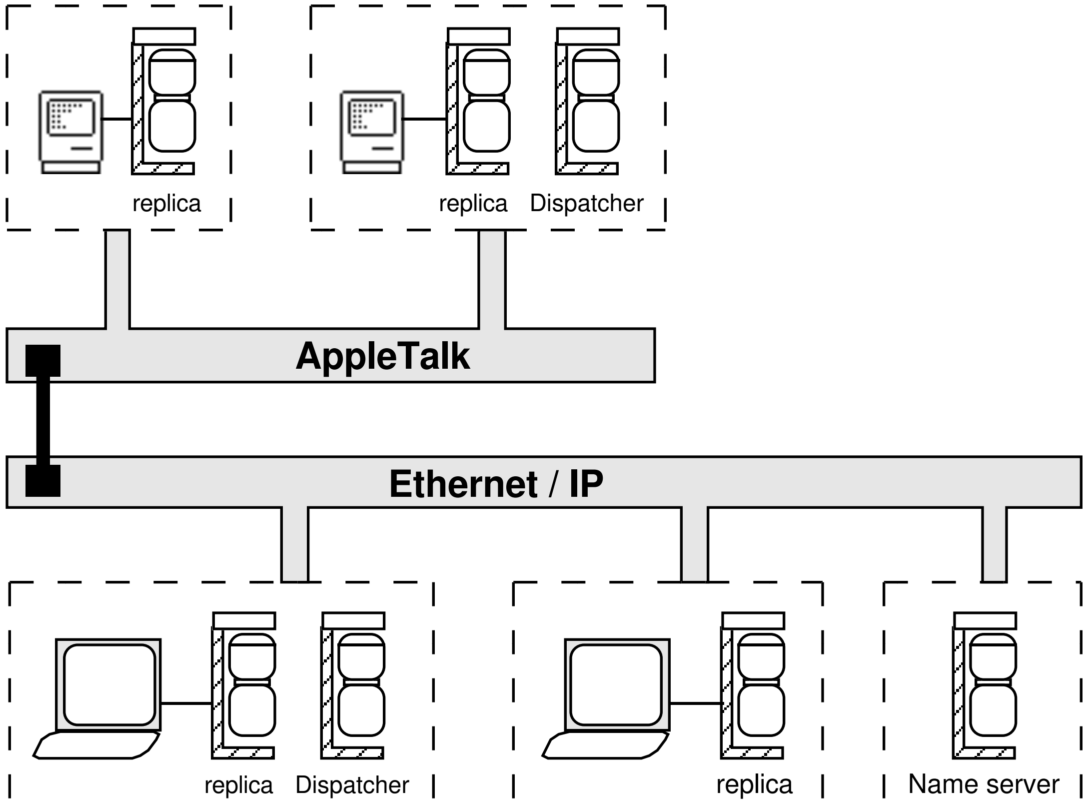

Figure 6 - Actors in a heterogeneous GroupDesign session

The messages of the reactive engine are sent as AppleEvents between Macintoshes and as UDP datagrams under Unix. We chose to use UDP datagrams instead of TCP streams for several reasons. First, connection-oriented protocols like TCP are known to be too heavyweight for the efficient exchange of requests and replies [Cher88], which is the main communication pattern between actors. Second, datagrams simplify programming: they preserve message boundaries while streams do not; one datagram socket can receive messages from several sources while streams require a different socket for each active connection. Finally the lack of reliability of datagrams (duplication and loss of packets) is handled "for free" in upper layers of our protocol, namely by the distributed concurrency control algorithm [Kars92]. We have observed that datagrams prove to be very reliable on a LAN. If the reliability of datagrams turns out to be a problem over wider area networks, we can implement a dispatcher-to-dispatcher loss recovery protocol on selected parts of the network without impairing the performance of more reliable parts of the network.

To summarize, the reactive engine provides the following functions:

- multicast a message to a group of actors;
- wait for messages from other actors, parse them, call the corresponding methods and eventually return the result;
- monitor external event sources, such as the events coming from an X Window server;
- map C++ objects to unique identifiers (IDs) that can be exchanged between actors;
- provide built-in debugging facilities, such as the "whatMethodsDoYouKnow" and "traceYourMessages" methods.

Using the reactive engine, we have implemented the initial version of the Unix replica that uses the Graph Widget in less than three hours. This is one to two orders of magnitude faster than the original implementation of GroupDesign on the Macintosh (using AppleEvents and Design/OA).

DISCUSSION

In this section we discuss our approach to the design of the user interface of GroupDesign and the impact of a purely replicated architecture on the engineering of synchronous groupware systems.

User interface

The features that we have developed have been designed to cover the continuum from synchronous work of the group on a single part of the document to an almost asynchronous work. Our approach consists in analyzing the design space and proposing solutions according to several criteria. This approach contrasts with an ethnographic approach to CSCW, in which a researcher observes a group carrying out a given task in order to derive a set of requirements for a computer system that supports that task. We chose a top-down approach because we expect the user's behaviour when using a shared editor to differ from editing on paper. Of course, it is important to follow up this work with appropriate usability testing.

We have designed the user interface to provide transparent access and awareness of the group. Transparency means that the system does not bring obstacles in the way of the task the user is carrying out. This includes running an open floor and not having to lock objects explicitely. Conflicts have never proved to be a problem in our case. The users develop social protocols spontaneously and the interface provides the appropriate awareness to virtually eliminate them. Echo and Localization are especially useful for that purpose. The challenge of a good echo is to provide an accurate, non-disturbing and efficient feeling of what the other users are doing. Using animation and sound for echo is particularly appropriate due to the characteristics of our sensory system. For instance, Card et al. [Card91] have shown that short (less than 1 second) animation gives enough time to the other users to understand what is about to happen. Similarly, the Arkola bottling plant simulation [Gave91] showed that sound can play an important role in groupware systems.

Relaxing the strict WYSIWIS paradigm so that users have independent views provides a natural interface: the users can use GroupDesign as a single-user tool without being disturbed by the group activity. This is further supported by providing a range of coupling modes (Teleconference and Private Editing) so that the same tool can be used in a variety of group tasks. Relaxing WYSIWIS creates some well-known problems [Stef87b] due to the fact that the reference to an object by a user may not be understandable to another user. These problems might even be more important in a heterogeneous environment since the users can modify the same document through different interfaces. These problems can be overcome by a careful design of the interface. In GroupDesign we use the document itself to convey the information related to the group activity instead of using additional artifacts. For example the location of other users is displayed in the document, not in a separate window. This provides a natural way to share reference information and at the same time supports a better awareness of the group activity.

Another aspect of transparency is the ability to access the document and observe it in a variety of ways. Not only can a user navigate geographically in the document by scrolling, but he or she can also navigate in time and over the dimension of the group. Navigation over time is achieved by the Age and History functions. Navigation over the dimension of the users is carried out by means of the Identification, Localization and Teleport functions. We have observed that providing modes as well as commands was important for supporting group work. When using a mode (e.g. Age, Localization), the user has a real-time feedback of the group actiity that does not interfere with his or her task. Commands (e.g. History, Teleport) are used to access information directly relevant to the user's task.

Engineering synchronous groupware

Implementing real-time groupware systems is a challenge. Not only has one to devise new interaction techniques and artifacts, but one has also to face the difficulty of implementing a distributed system. We think it is essential for a real-time groupware system to provide an immediate response to each user's actions. This requires some degree of replication, implemented by a distributed algorithm. We have chosen in GroupDesign to replicate the whole application. This might not be feasible with other systems nor even desirable. In such situations, the best architecture probably is to centralize the functional core of the application and to replicate its interface. Indeed, the communications between the replicas of the user interface can bypass the functional core in order to achieve the kind of functions that we have introduced in this article (Echo, Localization, etc.), which are independent of the functional core.

An advantage of the purely replicated application is to make it possible to use existing toolkits or applications, as we did for GroupDesign. Using our reactive engine, turning a single-user application into heterogeneous groupware was relatively easy compared to the work we would have had to do if we had built the application from scratch. Nevertheless, the fact that the programming interfaces were not built as groupware toolkits raised some problems. For example, the Echo feature was difficult to implement because neither Design/OA nor the Graph Widget provides functions to animate objects in the background. Hence we had to bypass the programming interfaces and use the low level graphic system (Quickdraw on Macintosh, Xlib under XWindows). This created additional problems since the application assumes that it has complete control over the display. One solution would be to have animation facilities available in the programming interface, such as the ones described in [Chat92]. Another solution is to provide a more general facility similar to the overlays described in [Rose92], so that the application can draw on the overlay without interfering with the programming interface.

Designing the protocol used by the replicas raised a number of interesting issues, in particular with respect to the heterogeneous environment. Since the replicas must use the same protocol but use different programming interfaces, we had to make sure that the mapping between them was possible. We also had to take into account the constraints of the concurrency control algorithm, which requires that operations either commute or mask each other. This led us to define the protocol as a set of logical operations instead of the low-level actions of the programming interface. Of course, the mapping between the logical operations and the low-level actions need not be one-to-one. A replica can even ignore logical operations if it does not implement the corresponding function.

Characterizing the application by such a logical protocol proved to be a very powerful concept. For instance, we were able to take advantage of the automatic layout facilities of the Graph Widget that do not exist in Design/OA: a user of the Unix version of GroupDesign can activate the automatic layout of the graph, which gets dispatched to the other replicas as a set of MoveObject operations, as if the user had made the layout by hand. Another example of this concept was our implementation of a non-interactive replica that randomly moves, deletes or creates objects in the graph. Smarter non-interactive replicas could be used to clean-up the drawing, store it, or perform whatever automatic task.

CONCLUSION AND FUTURE WORK

We have presented GroupDesign, a synchronous shared drawing tool that runs in a heterogeneous environment of Apple Macintoshes and Unix workstations. The design of GroupDesign was driven by the ability of the system to support awareness of the group and transparency of the interface. This led us to define a number of features that are general enough to apply to other shared editors. In order to obtain a response time of the system similar to the response time of a single user application, we introduced the notion of purely replicated architecture. Finally, in order to implement the system in a heterogeneous environment, we developed a distributed actor-based system called the reactive engine.

There is no doubt that groupware is the next trend in interactive systems and that it will be a participant in the social revolution in computer science. However, there are several crucial aspects to be solved before groupware is widely available and accepted. In this article we address both user interface issues and software engineering issues. Designing appropriate user interface features is crucial to the acceptance of groupware because the users of computer systems are now used to advanced user interface features in single user applications.

Mastering the engineering of groupware systems addresses their availability and acceptance. Groupware systems will be available when tools and techniques will be more widely available for the implementation of distributed interactive systems. In addition, we believe that they will actually be used only if they run in heterogeneous environments and if the seam between single-user and multi-user applications is invisible. One way to achieve this is to make it possible to turn single-user applications into multi-user applications so that software vendors need not implement their products from scratch.

GroupDesign is a proof of feasibility along these lines; it is but a first step in what we see as a promising direction. Our future work will address three aspects. The first is to apply the features and architecture of our system to another shared editor in order to validate our model and gain more insight in the design of these types of systems. The second is to work on the session level issues, such as the management of several simultaneous sessions and the storage, retrieval and transfer of shared documents. The last is to define a groupware toolkit that supports the purely replicated architecture by extending the reactive engine.

ACKNOWLEDGMENTS

This work is partially supported by Apple France. We thank MetaSoftware for providing us with MetaDesign and Design/OA. Heather Sacco and Wendy Mackay helped enhancing the readability of this article. We also thank the reviewers for their helpful comments on an earlier version of this article.

REFERENCES

[Agha87] Agha Gul, Hewitt, Carl E., Actors a Conceptual Foundation for Concurrent Object-Oriented Programming. In Shriver, B. et Wegner, P. editors, *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, Mass. 1987.

[Appl91] Apple Computer, *Inside Macintosh, Volume VI*, Addison Wesley, Reading, MA, 1991.

[Beau91] Beaudouin-Lafon, M., *The graph widget - user's manual*. Technical report, LRI, Université de Paris-Sud, France, June 1991. version 1.5.

[Beau92] Beaudouin-Lafon, M., Karsenty, A., Transparency and Awareness in a Real-Time Groupware System. In *Proc. ACM Symposium on User Interface Software and Technology UIST'92*, Monterey, CA, November 1992, pp 171-180.

[Biel91] von Biel, V., Groupware Grows Up. In *MacUser*, June 1991, pp. 207-211.

[Birm89] Birman, K., Cooper, R., Joseph, T., Kane, and K., Schmuck, F., *The ISIS System Manual*, June 1989.

[Card91] Card, S. K., Mackinlay, J. D., and Robertson, G. G., The Information Visualiser, an Information Workspace. In *Proc. Human Factors in Computer Systems (CHI'91)* (New Orleans, LA, April 1991), pp. 181-188. ACM, New York, 1991.

[Chat92] Chatty, S., Integrating Animation with User Interfaces. In Engineering for Human Computer Interaction, *Proc. of IFIP WG2.7 Working Conference ,* Larson, J. and Unger, C. Editor, North-Holland, August 1992.

[Cher88] Cheriton, David R., *VMTP: Versatile Message Transaction Protocol: Protocol Specification*, RFC-1045, Stanford University, February 1988.

[Crow90] Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R., MMConf: An Infrastructure for Building Shared Multimedia Applications. In *Proc. Third Conference on Computer-Supported Coopera-ive Work (CSCW'90)* (Los Angeles, CA., October 1990), ACM, New York, 1990.

[Deer89] Deering, Steve E., *Hosts extensions for IP multicasting*, RFC-1112, Stanford University, August 1989.

[Dewa91] Dewan, P., Choudary, R., Flexible user interface coupling in collaborative systems. In *Proc. Human Factors in Computer Systems (CHI'91)* (New Orleans, LA, April 1991), pp. 41-49. ACM, New York, 1991.

[Elli90] Ellis, C.A., and Gibbs, S.J., Concurrency Control in Groupware Systems. In *Proc. ACM SIGMOD'89 Conference on the Management of Data*, (Seattle WA, May 1989). ACM, New York, 1990.

[Elli91] Ellis, C.A., Gibbs, S.J., and Rein, G.L., Groupware: Some Issues and Experiences. In *Communications of the ACM*, January 1991, Vol. 34, No 1, pp. 39-58.

[Gave91] Gaver, W. W., Smith, R. B., and O'Shea, T., Effective Sounds in Complex Systems: The Arkola Simulation. In *Proc. Human Factors in Computer Systems (CHI'91)* (New Orleans, LA, April 1991), pp. 85-90. ACM, New York, 1991.

[Gave93] Gaver, W. W., Synthesizing Auditory Icons. In *Proc. Human Factors in Computer Systems (CHI'93)* (Amsterdam, The Netherlands, April 1993), ACM, New York, 1993.

[Gibb89] Gibbs, S. J., LIZA: An Extensible Groupware Toolkit. In *Proc. Human Factors in Computer Systems (CHI'89)* (Austin, TX, May 1989), pp. 29-35. ACM, New York, 1989.

[Gree92] Greenberg, S., Roseman, M. and Webster, D., Human and Technical Factors of Distributed Group Drawing Tools. In *Interacting with Computers*, Vol 4, No 3, pp. 364-392, 1992.

[Kars92] Karsenty, A., Beaudouin-Lafon, M., An Algorithm for Distributed Groupware Applications. In *Proc. International Conference on Distributed Computing Systems (ICDCS'93)* (Pittsburgh, PA, May 1993).

[Knis90] Knister, M. J., and Prakash, A., DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors. In *Proc. Third Conference on Computer-Supported Cooperative Work (CSCW'90)* (Los Angeles, CA, October 1990). ACM, New York, 1990.

[Lamp78] Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, July 1978, Vol. 21, No. 7, pp. 558-565.

[Meta89] Meta Software Corporation, *Design/OA Manual*, 150 CambridgePark Drive, Cambridge, MA, March 1989.

[Minn91] Minneman, S. L., and Bly, S. A., Managing a Trois: a Study of a Multi-User Drawing Tool in Distributed Design Work. In *Proc. Human Factors in Computer Systems (CHI'91)* (New Orleans, LA, April 1991), pp. 217-224. ACM, New-York, 1991.

[Mock87] Mockapetris, P.V., *Domain Names - Concepts and Facilities*, RFC-1034, Information Sciences Institute, November 1987.

[Patt90] Patterson, J. F., Hill, R. D., and Rohall, S. L., Rendezvous: An Architecture for Synchronous Multi-User Applications. In *Proc. Third Conference on Computer-Supported Cooperative Work (CSCW'90)* (Los Angeles, CA, October 1990). ACM, New York, 1990.

[Patt91] Patterson, J., F., Comparing the Programming Demands of Single-User and Multi-User Applications. In *Proceedings of the fourth Symposium on User Interface Software and Technology (UIST'91)*, pages 87-94. ACM SIGCHI, ACM Press, November 1991.

[Rhyn92] Rhyne, J. R. and Wolf, C. G., Tools for Supporting the Collaborative Process. In *Proc. ACM Symposium on User Interface Software and Technology (UIST'92)* pp 161-170 (Monterey, CA, November 1992).

[Rose92] Roseman, M. and Greenberg, S., GROUPKIT A Groupware Toolkit for Building Real-Time Conferencing Systems. In *Proc. Conference on Computer-Supported Cooperative Work (CSCW'92)* (Toronto, Ontario, November 1992), pp 43-50. ACM, New York, 1990.

[Sche86] Scheifler, R. W., Gettys, J., The X Window System, In *ACM Transactions on Graphics* 5(2), April 1986, pp. 79–109.

[Stef87a] Stefik, M., Bobrow, D.G., Foster, G., Lanning, S., and Tartar, D. WYSIWIS Revisited: Early Experiences with Multiuser Interfaces. In *ACM Transactions on Office Information Systems*, Vol. 5, No 2, April 1987, pp. 147-186.

[Stef87b] Stefik, M., Foster, G., Bobrow, D. G., Keneth, K., Lanning, S., and Suchman, L., Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. In *Communications of the ACM*, January 1987, Vol. 30, No 1, pp. 32-47.

[Tang91] Tang, J. C., Findings from Observational Studies of Collaborative Work. *International Journal of Man-Machine Studies*, Vol 34, pp 143-160, 1991.