Rendezvous: An Architecture for Synchronous Multi-User Applications

John F. Patterson
Ralph D. Hill
Steven L. Rohall

Bellcore
445 South St.
Morristown, NJ 07960-1910

W. Scott Meeks

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

ABSTRACT

Rendezvous is an architecture for creating synchronous multi-user applications. It consists of two parts: a run-time architecture for managing the multi-user session and a start-up architecture for managing the network connectivity. The run-time architecture is based on a User Interface Management System called MEL, which is a language extension to Common Lisp providing support for graphics operations, object-oriented programming, and constraints. Constraints are used to manage three dimensions of sharing: sharing of underlying information, sharing of views, and sharing of access. The start-up architecture decouples invoking and joining an application so that not all users need be known when the application is started. At present, the run-time architecture is completed and running test applications. As a first test of the complete Rendezvous architecture, we will implement a multi-user card game by the end of the summer.

INTRODUCTION

We envision a future in which computer workstations and terminals support simultaneous interaction among users who are distributed across a communication network. This is frequently referred to as synchronous computer conferencing and includes systems for sharing windows (Lantz [1986], Ensor et al. [1988], Gust [1988], and Patterson [1990]) or sharing viewing surfaces (Stefik et al. [1987], Forsdick [1985]). These systems provide a content-independent form of sharing; they are not greatly concerned with how the users plan to interact or about what they hope to communicate. A different type of synchronous computer conferencing is an application that structures the user interactions in a very specific way, e.g., a multi-user card game or a multi-user training simulation. We refer to these customized computer conferences as multi-user applications.

Rendezvous is an architecture for creating multi-user applications. It provides support for managing a multi-user session, for performing fundamental input and output activities, and for controlling the degree to which the multiple users either share or do not share both information and control. We start from a User Interface Management System (UIMS), called MEL (Hill [1990]), which was designed to support multiple users. We believe that the UIMS is exactly the level in which support for multiple users must be addressed. At some level of abstraction a multi-user application is like any other application, but with more input and output devices to control and coordinate; this is what a UIMS provides. We hope that by providing both UIMS and other support, multi-user applications can become at least as easy to develop as single-user applications.

This paper provides a broad overview of Rendezvous. We start by identifying some of the requirements of multi-user applications. Second, we present the architecture of Rendezvous and discuss how its features can meet the requirements. Next, we discuss several issues that are left unaddressed by Rendezvous. Finally, we conclude by mentioning our progress and plans.

REQUIREMENTS

In addition to the requirements of single-user applications, multi-user applications make several new demands. The most obvious is a need to control the degree of sharing and nonsharing among the users. Also, since the multi-user applications that concern us are network-based, an architecture must help manage this network connectivity. The following sections describe these requirements, which we feel any architecture for multi-user applications must address.

Provide flexible control over the dimensions of sharing

The essence of multi-user applications is sharing and control of sharing. Consider a multi-user card game. Players will have a public view of a "table" and a private view of their "hand"; "cards" will be moved among the players; and control may shift systematically among the players as they take turns. Even a simple application like a card game can lead to complicated combinations of sharing and nonsharing.

We start by defining a notion similar to the Andrew Toolkit's distinction between data objects and view objects (Palay et al. [1988]). For us, Andrew's data objects are *underlying objects;* its view objects are *interaction objects.* The underlying objects correspond to the abstract information comprising the application, but without any indication of how this information should be displayed. The information about how to display and interact with an underlying object is contained within an interaction object. We speak of the *view,* i.e., output, and *access,* i.e., input, provided by an interaction object.

Given these definitions, we think in terms of three dimensions of sharing.[1]

1. **These dimensions were first developed in the Groupware Technology Workshop sponsored by IFIP Working Group 8.4 and ACM SIGOIS that was held in Palo Alto, California on August 24-25, 1989. The participants in the synchronous groupware subgroup in which the ideas developed were: Brad Chen, Terry Crowley, Bob Ensor, Keith Lantz, Sergio Mujica, John Nosek, John Patterson, Gail Rein, and Sylvia Wilbur.**

The first is sharing of the underlying objects. Ignoring how they are depicted, the "cards" in one's "hand" are a private underlying object and the "cards" on the "table" are a public underlying object. There must be ways to control how interaction objects become attached to or detached from underlying objects, which in turn controls whether the underlying objects are shared or not.

The second dimension of sharing is sharing of the presentation or view provided by the interaction objects. It is reasonable to suppose that one player might have one perspective on the "table", while another might see the "table" rotated. The underlying objects are the same, but the view on these objects are individualized. Alternatively, in some situations, such as training simulations, an instructor might wish to have an absolutely literal copy of the student's view. Not only is the underlying information being shared, but the view is shared as well.

The final dimension of sharing is sharing of the input authorizations or access provided by the interaction objects. Consider a spectator of a card game. This participant looks over the shoulder of a real player, but does not actually manipulate the cards. This user might have a literal copy of some player's view, but be prevented from interacting with the cards in any way. Alternatively, a card game might permit all players equal access to cards on the "table" and leave it to the users to behave sensibly.

We do not claim that these dimensions of sharing are independent. Sharing of views and access presume sharing of the underlying object. Also, access is, generally, unavailable without a view. Moreover, the various types of sharing will be combined in complicated ways within a single application. Our claim is that an architecture for multi-user applications must provide flexible control over these dimensions of sharing.

Provide robust session management

The typical mechanisms for starting and stopping single-user applications are inadequate for multi-user applications. A single-user application is usually started from another single-user application and inherits information about how to communicate with the user. When it is time to leave the single-user application, the application is simply stopped and the communication to the user is lost. If a multi-user application is started from a single-user application, it will be unable to inherit the necessary information about how to communicate with its users. Indeed, the eventual set of users may not even be known at start up. Also, when one user leaves a multi-user application, this cannot be taken as a signal that the application has ended. Clearly, a different approach to these issues is required.

To better understand what is needed, let's first define a few terms. A *session* is the duration of an application process from its first invocation to its termination. *Joining* a session is the act of connecting a user's display resources to the session. *Dropping* a session is the act of disconnecting a user's display resources from the session.

Whereas single-user applications join a user as soon as they start and drop the user as soon as they end, multi-user applications must support a richer regime. A session may be started by someone, e.g., a conference administrator, on behalf of others without the participation of that individual in the actual session. Users will join as they can and will drop as necessary without halting the session. Additionally, sessions may or may not be terminated by the departure of the last user. These aspects of session management are necessary if multi-user applications are to be readily accessible to their users.

Maximize the joint probability of successful execution

If a multi-user application is to be successfully deployed, it must maximize the joint probability that several users will be able to join the application. If the community of users is open or very heterogeneous, this can be difficult to achieve. Operating systems may differ; terminals may vary; and communications protocols may not be standardized. An architecture for multi-user applications should attempt to minimize the restrictions that it places on a user's computing environment. At a minimum it is necessary to adopt a common protocol for establishing communication paths and a common protocol for controlling user terminals, but these common protocols should be as widely available as possible.

ARCHITECTURE

The Rendezvous architecture can be thought of as two architectures. One is a run-time architecture that supports the ongoing interactions among users during the application session. The other is a start-up architecture that supports many of the session management issues mentioned earlier. After discussing these two architectures in an abstract way, we will discuss the implementation environment for Rendezvous.

Run-Time Architecture

The run-time architecture is the organization of the software resources once all users have joined a session; it is not concerned with the problems of session management. Figure 1 depicts a Rendezvous session for several connected users. Closest to the user is a virtual terminal. This controls all direct input and output to the user and presents an abstract model of the terminal capabilities. Examples of potential virtual terminals are the X Window System™ (Scheifler, Gettys, and Newman [1988]) and NeWS™ (Sun Microsystems [1987]). We have elected to use the X Window System, but the more important point is that there must be some network-based virtual terminal used by all potential participants. Otherwise, a Rendezvous application must be prepared for a plethora of alternative terminal types and somehow support the interoperability among these types.

The virtual terminals of the connected users all communicate with one centralized process that controls the application. This application process can be decomposed into several light-weight processes contained within it. For each connected user there is an interaction process that contains the interaction objects for that user. In addition, there is one underlying process that contains the underlying objects for the application. The underlying objects model the application, provide support for session management, and generally facilitate the coordination among users. The interaction objects manage the display and interaction for their respective users without any direct concern for the presence of other users.

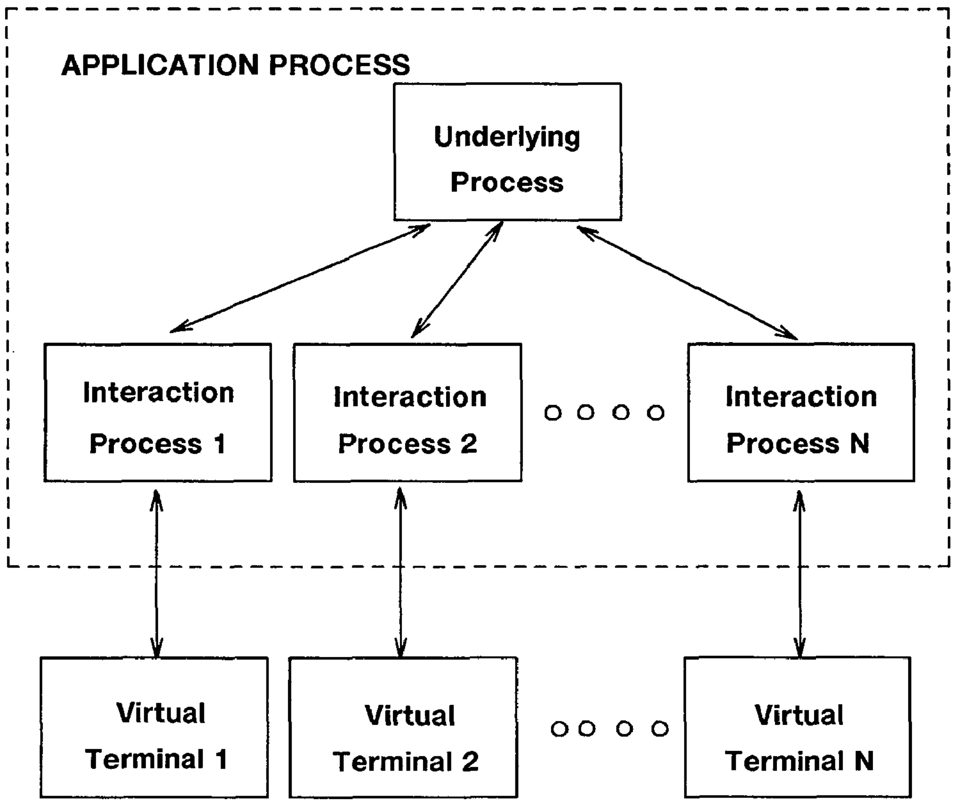

Figure 1. Rendezvous Run-Time Architecture

The underlying process and the interaction processes in Rendezvous are each instances of MEL light-weight processes. In the next two sections we will provide an overview of MEL's capabilities and then discuss how one of MEL's capabilities, constraints, facilitates multi-user programming.

MEL Capabilities

In a mouthful, MEL is an object-oriented, event-based language extension to Common Lisp incorporating mechanisms for constraints and fundamental graphics control (Hill [1990]). MEL's object-oriented programming facility is largely an adaptation of the Common Lisp structure facility. The object classes are in a single hierarchy with one line of inheritance.

MEL's graphics support starts with a set of primitive objects and mechanisms for combining these primitive objects into composites. For any such graphical object, MEL takes care of redraw under movement or collision, sensitivity to input, and notification of collisions. For composite objects, MEL provides mechanisms to manage the layout of the composite's constituents.

Perhaps the most intriguing aspect of MEL and the most interesting for multi-user applications is its control structure. MEL provides two mechanisms by which activities may be expressed: rules and constraints. Borrowing from earlier work (Hill [1986] and Hill [1987]), rules are used to express procedural information; they associate a triggering event with an action in a fashion similar to a production system (Hayes-Roth, Watterman, and Lenat [1978]).

Events can arrive from outside, e.g., a button press, or can be initiated by one object as a signal to another. They are queued in the order of their arrival.

Constraints are used to express declarative information about relationships among objects that must be maintained. These are simple one-way constraints of the form *assure B is some function F of A.* Two-way constraints can be expressed as a pair of one-way constraints, for example by adding *assure A is the inverse of function F of B.*

Roughly speaking, the flow of control in MEL is an endless cycle of processing an action triggered by an event and then reconciling any constraints. The rules provide a mechanism for activities outside Rendezvous to cause changes in an application's objects. The constraints express the relationships among objects that must be maintained despite these exogenous changes. In a sense, the constraints describe an equilibrium to which an application constantly returns.

Constraints

Although MEL's constraints were originally devised for maintaining graphical relationships as in Szekely and Myers [1988] or Hill and Herrmann [1989], it has become very clear that they are also well suited to multi-user applications. In retrospect this seems obvious; multi-user applications possess a great many state variables that must be kept consistent. Constraints are an ideal mechanism for expressing such relationships and ensuring that they are maintained.

To illustrate the value of constraints in multi-user applications, let's consider three examples. The first is the use of constraints to maintain a viewing transformation or, in our terminology, the relationship between an interaction object and the underlying object to which it provides access. For example, a simple bar graph would consist of an underlying object containing some statistical information to be plotted and an interaction object containing information about the height of the bar. Initially, one constraint might indicate that the height should be maintained as some linear transform of the underlying value. This would ensure that any changes to the underlying value would be readily reflected in the displayed height. It would not, however, permit a change to the height to affect the underlying value. This is accomplished by a second constraint indicating that the underlying value should be related to the height by the inverse of the earlier transform. Now any changes to the height will change the underlying value.

This use of constraints to express a viewing relationship between interaction objects and underlying objects is invaluable. The separation of interaction and underlying objects is necessary to ensure that several users can have different views on the underlying information. If one user changes some aspect of an interaction object, this can cause a change in the underlying object which will propagate a change to any other interaction objects. Once the constraints are expressed, the constraint maintenance system ensures that all viewing relationships are maintained regardless of the source of a change to an object.

A second use of constraints in multi-user applications arises when it is desirable to ensure that two users are viewing exactly the same information, which is sometimes referred to as WYSIWIS (Stefik et al. [1987]) or What-You-See-Is-What-I-See. Frequently, it is not sufficient to simply share via the underlying objects. Consider, for example, a multi-user chess game in which two users have essentially the same view of the board, but are free to rotate their views separately. The degree of rotation is not part of the underlying representation; it is part of each user's interaction representation. Now, suppose that the two users wish to see an identical view in order to discuss the board. Somehow, the degrees of rotation must be made equal and maintained as equal under any future modifications.

Rendezvous supports WYSIWIS by introducing a special type of underlying object called a *sharing object*. This object possesses slots that mimic those of the interaction objects requiring WYSIWIS consistency. The slots in the interaction objects are constrained to be equal to those of the sharing object and vice versa, which assures that the WYSIWIS experience is provided.

The third use of constraints in multi-user applications is for access control such as turn-taking in a game. In MEL, user input arrives as an event that is delivered to the relevant interaction object. The event is handled by the object with a rule that triggers some action in response to the event. If two users have essentially the same interaction objects, then both users can provide input. The goal, however, is to ensure that only one user may provide input and that the other is blocked.

To accomplish turn-taking we rely on three of MEL's capabilities: 1) the ability of an interaction object to sense a value assigned uniquely to its interaction process, 2) the ability to control rule-activation via local object state, and 3) constraints. An interaction object can include in the precondition of a rule a reference to a guard variable. If the value of the guard variable is equal to the value of some other variable, e.g., the interaction process's value, then the rule can fire; otherwise it cannot. By constraining the guard variables of the user's interaction objects to equal an underlying variable indicating which interaction process has the turn, access can be limited. Only the interaction objects of the authorized user will process any input.

These three examples illustrate two ways in which MEL's constraints provide value in multi-user programming. The first is by propagating state changes through the application. The programmer may register constraints in a local fashion, but MEL will ensure their global consistency. The second is by hiding issues of concurrent access. Rules are constrained to act only on the state associated with the light-weight process that activated them. Consistency across these light-weight process boundaries is maintained via constraints. When the constraints are reconciled, all state is locked from being changed until a new equilibrium is established. The programmer need not be concerned with which variables require locking and which light-weight processes are interested in the variables. MEL locks all variables and assumes that any process might be interested in any state.

Start-Up Architecture

The Rendezvous start-up architecture is depicted in Figure 2. Every user has access to a virtual terminal. From this terminal they have access to a program called the Rendezvous Access Point (RAP), which is their entry into

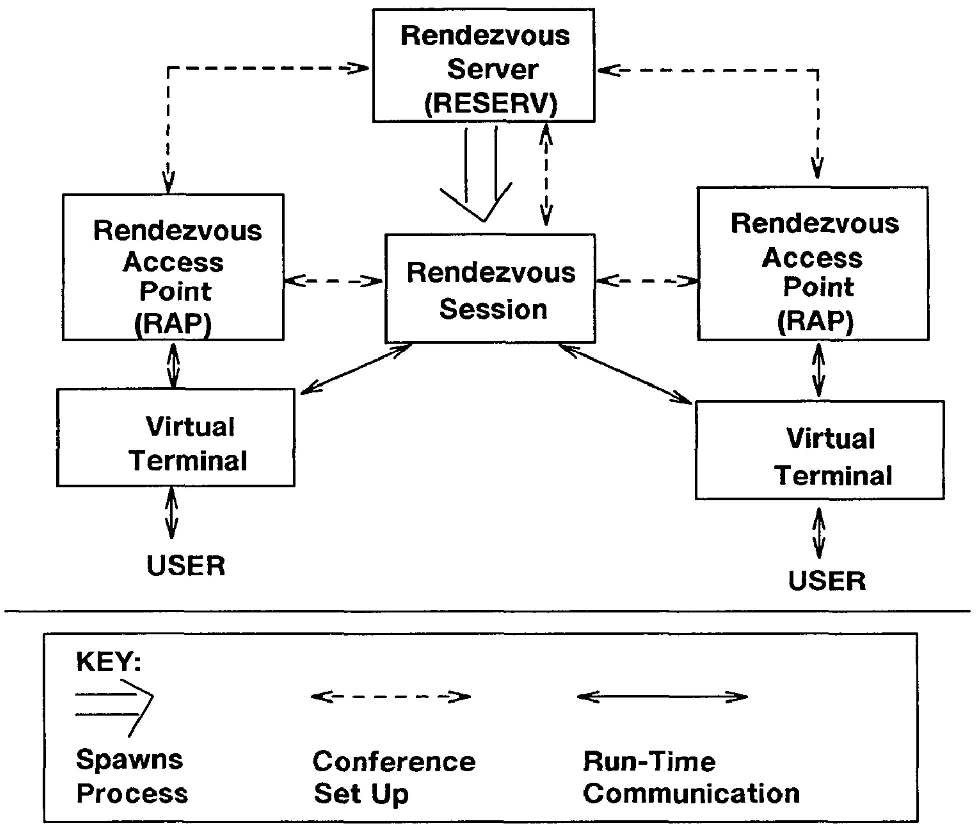

Figure 2. Rendezvous Start-Up Architecture

Rendezvous. The RAP is the program from which they can start or join a Rendezvous session. It is the Rendezvous equivalent of a UNIX^{TM} shell, but with far fewer options than the typical shell.

The RAP does not actually start a session; this is done by the Rendezvous Server (RESERV). It is a name server for Rendezvous sessions as well as the parent process from which these sessions are invoked. To start a Rendezvous application the RAP sends a message to RESERV asking it to start the application. Once the Rendezvous session is running it will register an address with RESERV indicating a communication "channel" that will be monitored for incoming calls.[2] The RAP requests this information from RESERV and then calls the Rendezvous session. Upon establishing communication with the Rendezvous session, the RAP informs the session of the address of the user's virtual terminal. The Rendezvous session can now connect to the user's virtual terminal and the session is joined.

The Rendezvous start-up architecture provides several benefits. First, application invocation and session joining are decoupled so that the eventual

2. In the context of sockets, this is the address of a passive socket. A call is an attempt to open the passive socket by another process, which leads to the establishment of an active socket within the Rendezvous session.

participants need not be known at start-up. Second, this architecture insulates the user from issues about the machine on which the application is running. The application is therefore permitted to use a configuration and resources that might be unavailable to the user machines. Third, given a well-known address for each user's RAP, they can act as recipients of invitations to join ongoing Rendezvous sessions. Finally, the presence of a name server for Rendezvous sessions permits users to obtain a list of active sessions, which makes it possible to support sessions that permit anyone to join at any time. We see these features as important to the eventual success of Rendezvous or any other architecture for multi-user applications.

Implementation Environment

The implementation environment for Rendezvous[3] is Sun™ workstations connected via Ethernet™ and running the UNIX™ operating system with BSD UNIX sockets for inter-process communication and the X Window System™ for user interaction. The multi-user applications are expressed in MEL, which is an extension of Allegro CL™, Common Lisp from Franz, Inc.

It is one thing to identify the implementation environment for an architecture and quite another to clarify which aspects of this environment are essential to the architectural model. For Rendezvous we divide this into two questions: what does a user need to participate in a session and what does an application need? We believe that only three things are essential for a user to participate: network access, a standard interprocess communication mechanism, and a standard virtual terminal protocol. We have chosen to use sockets and the X Window System because they are widespread and therefore help to maximize the probability that multiple users can interact. Other choices could be reasonable, however.

Looking now from the application side, RESERV and the Rendezvous application are dependent on two aspects of our implementation. First, RESERV relies on the UNIX support for multitasking, but we take little advantage of any other UNIX-specific capabilities. Second, we use the light-weight process capability of Allegro CL. Strictly speaking we do not need light-weight processes, assuming the operating system provides processes, but they are convenient.

DEFERRED ISSUES

There are many aspects of multi-user applications not supported by our first version of Rendezvous. Here, we mention three such issues both to recognize their importance and to register our intent to address them in the future.

3. It is the policy of Bellcore to avoid any statements of comparative analysis or evaluation of vendors' products. Any mention of products or vendors in this paper is done where necessary for the sake of scientific accuracy and precision, or for background information to a point of technology analysis, or to provide an example of a technology for illustrative purposes, and should not be construed as either positive or negative commentary on that product or that vendor. Neither the inclusion of a product or a vendor in this paper, nor the omission of a product or a vendor, should be interpreted as indicating a position or opinion of that product or vendor on the part of the author(s) or of Bellcore.

User-to-User Communication

Multi-user applications can benefit from more channels of communication than are typically provided by a computer terminal. A typical interface to a multi-user application must provide both ways to interact with the computer-based application and ways to communicate with the other users. For example, in a multi-user card game there will be ways to interact with the cards, which are the object of everyone's attention, but there must also be ways to communicate with the other players. At present, we satisfy this need with a phone call, but managing this phone connection should be part of the application and not merely a parallel conference.

This user-to-user communication is an essential part of any multi-user application and will be supported in the future. Building on other projects at Bellcore, we will incorporate control of audio/video connections among users.

Enhanced Session Management

Our first approach to session management in Rendezvous skirts two issues that arise when users can enter a session late. The first is concerned with a user who joins a session that is well-advanced. What will Rendezvous do to help that user catch up? The second issue arises when a multi-user session must be suspended indefinitely and restarted later. How will Rendezvous represent this persistent information and how will the session be reinstated?

Late joining and session suspension, while not essential, greatly enhance the utility of multi-user applications. Somehow the late joiner must be accommodated by copying information from earlier participants to initialize the late joiner. As for session suspension, some concept of persistent objects, at least for the underlying objects, will be required.

Structural Consistency

Structural consistency is a problem that arises because Rendezvous expresses its viewing transformations with constraints. If the underlying objects form a hierarchy that is mapped onto a hierarchy of interaction objects, then a problem arises when one or the other hierarchy changes by adding a child or removing a child. Presumably the other hierarchy should also either add or remove the corresponding child, but this relationship can be difficult to express with simple constraints.

We view this issue as a research question. We do not doubt that structural consistency can be maintained within Rendezvous, but our goal is to formulate a simple, robust mechanism that will insulate programmers from this complexity in the future.

PROGRESS AND PLANS

At present (February, 1990), Rendezvous is still being assembled. The run-time architecture is largely complete and running several test applications such as a multi-user Tic-Tac-Toe game. The start-up architecture has not yet been incorporated, but is largely implemented for a different multi-user project (Patterson [1990]). By the summer we hope to have the whole infrastructure tied together.

With this infrastructure we will first address multi-user games as an application domain. We have chosen this class for its generality, its simplicity, and our shared familiarity with the domain. We will start by implementing a generic form of card game that supports the physical (e.g., card movement, shuffling), but not the social (e.g., rules of the game), aspects of playing cards. Using audio connections, the users will be expected to coordinate their activities to achieve any particular instance (e.g., bridge, crazy eights) of a card game.

With time and experience, we hope to address two questions. First, we would like to determine how to improve Rendezvous so that multi-user applications become easy to write. While we do not expect to support non-programmers, we would like Rendezvous to be accessible to junior and entry-level programmers. Second, we would like to use Rendezvous applications to understand the degree to which multi-user applications can substitute for face-to-face encounters.

REFERENCES

J. R. Ensor, S. R. Ahuja, D. N. Horn, and S. E. Lucco, The Rapport Multimedia Conferencing System - A Software Overview, *Proceedings of the IEEE Conference on Computer Workstations*, Santa Clara, California, March 7-10, 1988, 52-58.

H. Forsdick, Exploration into Real-time Multimedia Conferencing, *Proceedings of the Second International Symposium on Computer Message Systems*, September 1985, 299-315.

P. Gust, Shared X: X in a Distributed Group Work Environment, Unpublished paper presented at the Second Annual X Technical Conference, January, 1988.

F. Hayes-Roth, D. A. Watterman, and D. B. Lenat, Principles of Pattern-Directed Inference Systems, in *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, 577-601.

R. D. Hill, Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction — the Sassafras UIMS, *ACM Trans. on Graphics* 5,3 (1986), 179-210.

R.D. Hill, Event-Response Systems -- A Technique for Specifying Multi-Threaded Dialogues, *Proceedings of CHI+GI 1987,* New York, 1987, 241-248. (Toronto, April 5-9.).

R.D. Hill and M. Herrmann, The Structure of Tube — A Tool for Implementing Advanced User Interfaces, *EuroGraphics'89,* 1989, 15-25. (Proceedings of the European Computer Graphics Conference and Exhibition, Hamburg, Federal Republic of Germany, Sept. 4-8, 1989.).

R. D. Hill, A 2-D Graphics System for Multi-User Interactive Graphics Based on Objects and Constraints, in *Advances in Object Oriented Graphics,* E. Blake and P. Wisskirchen (editors), Springer-Verlag, Berlin, 1990.

K. A. Lantz, An Experiment in Integrated Multimedia Conferencing, *Proceedings of CSCW '86 Conference on Computer-supported Cooperative Work*, Austin, TX, 1986, 267-275.

A. J. Palay, W. J. Hanson, M. Sherman, M. G. Wadlow, T. P. Neuendorffer, Z. Stern, M. Bader, and T. Peters, The Andrew Toolkit - An Overview, *Proceedings of the USENIX Winter Conference*, Dallas, TX, February 9-12, 1988.

J. F. Patterson, The Good, the Bad, and the Ugly of Window Sharing in X, *Proceedings of the Fourth Annual X Technical Conference (Boston, January 15-17)*, January, 1990.

R. W. Scheifler, J. Gettys, and R. Newman, *X Window System: C Library and Protocol Reference*, Digital Press, Bedford, Massachusetts , 1988.

M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar, WYSIWIS Revised: Early Experiences with Multiuser Interfaces, *ACM Transactions on Office Information Systems* 5,2 (April 1987), 147-167.

Sun Microsystems, NeWS Manual, 800-1632-10, Revision A, 29 March 1987.

P. Szekely and B. Myers, A User Interface Toolkit Based on Graphical Objects and Constraints, *Sigplan Notices 23*,11 (1988), 36-45. (OOPSLA'88 Conference Proceedings, September 25-30, 1988, San Diego, California.).