A Non-Invasive Platform Supporting Distributed, Real-Time, Multimedia Collaboration

Ming C. Hao, Joesph S. Sventek
Software Technology Laboratory
HPL-98-101
May, 1998

distributed, real-time, multimedia, collaboration

This report describes the architecture and an implementation of SharedApp, a platform supporting distributed, real-time, multimedia collaboration.

Internal Accession Date Only

A Non-invasive Platform Supporting Distributed, Real-Time, Multimedia Collaboration

Ming C. Hao and Joseph S. Sventek

This report describes the architecture and an implementation of SharedApp, a platform supporting distributed, real-time, multimedia collaboration.

May 21, 1998
HPL/STL/DSA-98-001
Version 1.0

1.0 Introduction

1.1 Business Need

One of the frequently-mentioned promises of ubiquitous distributed computing is the ability to support real-time electronic collaboration. By coupling such electronic interactions with voice (and potentially, video) channels, non-collocated individuals should be able to produce collaborative artifacts without incurring the costs and inconvenience of travel to a central location for face-to-face collaboration.

This is especially true for those industries in which there is a heavy dependence upon sophisticated, computer-aided design (CAD) tools, such as the automotive and aerospace industries. In addition to collaborative design, other forms of interaction around the CAD artifacts must occur for these companies to successfully conduct business - e.g. interactions between design engineers and factory-floor engineers when setting up the manufacturing lines or between design engineers and maintenance engineers when servicing a product. In each of these cases, it is difficult or impossible to achieve collocation between the communicants. As a result, distributed, collaborative use of design tools is a high priority for these industries.

The current state of the art for collaboration software is limited to providing a shared 3D view [4], only. Most software collaboration research prototypes and products are based on file/image/store-forward sharing.

1.2 Invasive Solutions

There has been a substantial amount of research into the infrastructure needs for real-time, collaborative tools [1-12]. These infrastructures have predominantly required that applications be specially constructed to avail themselves of features in the infrastructures in order to support collaborative use. Those that have taken a less invasive approach [8] still require that the application code be relinked with a special library in order to support collaborative use. An application vendor wishing to support collaboration using these infrastructures, in addition to the equivalent stand-alone product, would have to support two or more products in the marketplace.

Unfortunately, the commercial imperative in the computing industry today is to produce best-in-class applications targeted for use by individual users. The sheer size of the market for such applications easily dwarfs that for collaboration-aware applications. As a result, few vendors go to the trouble of producing collaboration-ready applications for consumption by the public.

Remote, real-time collaboration relies heavily upon additional, isochronous channels of communication, in addition to the non-isochronous, data interaction provided by collaboration-aware applications. A voice channel is absolutely necessary, and a some systems [8, 18] have experimented with video channels, as well. The requisite degree of coupling/synchronization of the isochronous channel[s] to the data channel is not generally agreed, and many of the research efforts in collaboration infrastructures have spent considerable effort in providing support for these isochronous channels.

1.3 Our Non-invasive Solution

The platform that we have developed (SharedApp), described herein, was designed to meet the following requirements:

- it must be non-invasive to applications, window systems, and operating system platforms;
- it must operate well in network environments with limited sustainable bandwidth; and
- it must support fully-synchronized collaboration among a small (2-10 participants) group of distributed collaborators using their application[s] of choice.

An explicit non-goal of the current work is to integrate voice communication into the data communication infrastructure. Successful use of a SharedApp collaborative session requires each user to have an out-of-band voice connection with the other users. This is easily achieved through the use of voice bridging technologies available from most telecom network operators.[1]

The remainder of the paper is as follows: section 2 describes the architecture of SharedApp; section 3 details the implementation and experience building and using SharedApp in an X Windows environment; section 4 compares this work with previous work in the field; and section 5 summarizes the work and describes future directions.

1. This assumption (out-of-band voice communication) significantly simplifies the provision of a collaboration platform. If there is a need to more formally synchronize the audio content with the data content (e.g. to store potentially causally-related content for later replay), then it will be necessary to not only augment the current system with microphones and speakers, but also to battle with the network for sufficient guaranteed bandwidth to provide adequate voice quality.

2.0 The SharedApp Architecture

Any platform supporting collaboration must address the following technical issues:

- is the collaborative application centralized or replicated?
- is the application itself collaboration-aware?
- how are user interactions synchronized?

The following sections describe the SharedApp architecture with regards to these technical issues.

2.1 Centralized vs. Replicated Structure

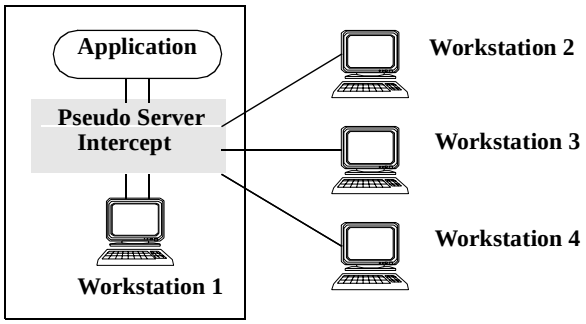
There are two possible structures for constructing a collaboration infrastructure:

1. in a centralized structure, there is only one instance of the shared application; input to the application is sent to the execution site, and the output of the application is sent to all of the collaborating displays; examples of this style of system are SharedX [3] and JVTOS's Shared Window System [1,2]
2. in a replicated structure, an instance of the shared application is executed locally on each user's workstation; user inputs to and outputs from the shared application only occur locally, with some coordination traffic occurring between the instances of the application; examples of this style of system are MMConf [8] and VConf [9]

As illustrated in Figure 1 on page 3, a common implementation technique for the centralized structure is to interpose a pseudo-window-server between an application and the display's window server. The pseudo-server receives window system calls from the application, and fans these out to the window servers for each of the connected workstations; it also receives the input events from each connected workstation. This implementation technique usually generates substantial network traffic due to the continual transmission of graphics primitives and bitmaps to the connected workstations. JVTOS [1,2] and SharedX [3] both use this technique.

Another drawback to the pseudo-server scheme is that only graphics calls directed to the pseudo-server can be shared. Most sophisticated graphics applications, like CAD tools, use direct hardware access (DHA) to maximize graphics performance. Use of DHA bypasses the

Figure 1. Centralized Intercept Structure



window system, rendering the pseudo-server technique useless for collaboration with DHA applications.

The replicated structure significantly reduces the network traffic and offers superior response time, but at the expense of synchronization complexity among multiple instances of the shared application. Lauwer's "Replicated Architectures for Shared Window Systems" [6] describes some of these synchronization problems in detail; they usually manifest themselves as input/output inconsistency among the multiple displays and event ordering difficulties. To resolve these problems, Lauwers required that applications be made collaboration-aware. Many other solutions to synchronization issues have been proposed [10-12,14].

Due to our requirement for operation in network environments with limited sustainable bandwidth, SharedApp employs the replicated structure, thus trading off bandwidth for processor cycles. This structure also permits DHA applications to be used collaboratively. In exchange for the flexibility provided by the replicated structure, SharedApp must solve the synchronization problems inherent in a replicated structure.

2.2 Application Encapsulation

Central to the design of the SharedApp architecture is the following assertion:

> Once a window-based application has been initiated and initialized to a particular state, all changes to the application are effected via events delivered by the window system.

For applications that satisfy this assertion, we can achieve a non-invasive, replicated, collaborative structure through the following steps:

1. create a collaboration session by initiating and initializing instances of the application to be shared on the multiple workstations involved in the session; and
2. capture events targeted at each of these application instances; multicast the captured events to each of the collaborating instances and replay the events to each collaborating instance.

In order to do this non-invasively, step 1 above is broken down into three constituent steps:

a. create a component that acts as the session controller
b. create an encapsulation component on each of the target workstations to be involved in the session
c. initiate and initialize an application instance on each of the target workstations

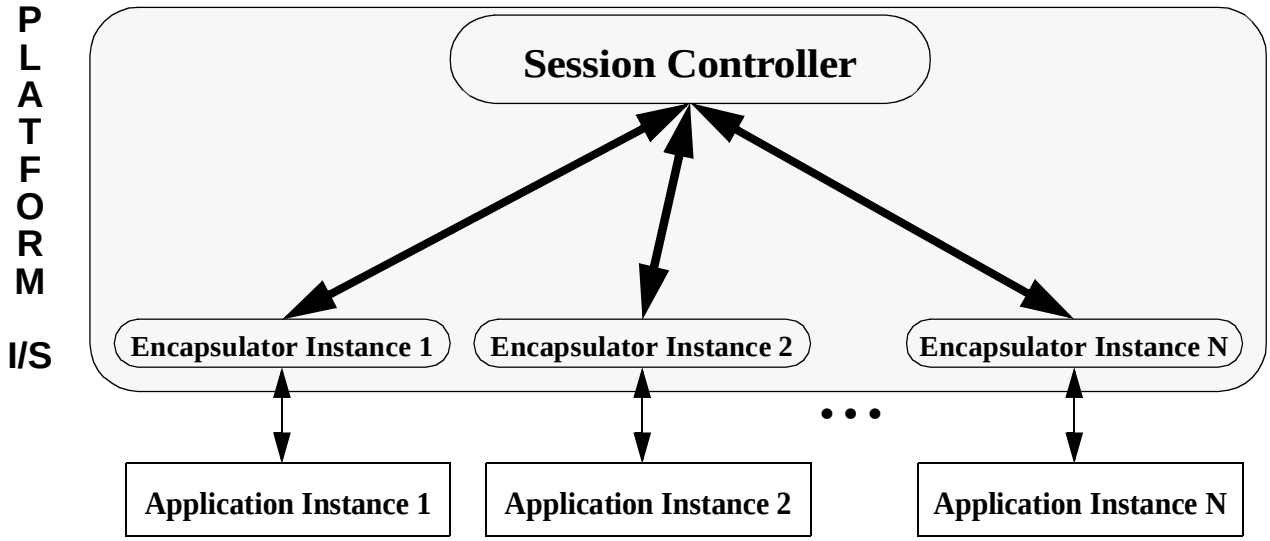
This component structure is shown in Figure 2 on page 4.

The session controller and application encapsulators (platform infrastructure) conspire to provide the collaborative work session. The individual application instances continue to be operated in single-user mode. As such, this structure satisfies the non-invasive requirement for the collaboration platform.

2.3 Synchronization

As alluded to in Section 2.2 on page 3, once this component structure is in place, the collaboration is achieved by capturing window system events targeted at an application instance's window, communicating those events to each of

Figure 2. Component structure for a SharedApp session



the encapsulators, and replaying those events to the application instances.

The synchronization requirement described in Section 1.3 on page 2 requires that the collaborators be synchronized in both place and time. Since we have exchanged the communication load of the centralized structure for concurrent execution in the replicated structure, we must ensure that each instance eventually is in the same state. In order to use the application collaboratively in real-time, we must also ensure that the replay of each event to the multiple instances is sufficiently synchronized in time. Absent either of these characteristics, the system will be unusable for real-time collaboration.

In order to meet the synchronization in place requirement, it is essential that each instance see the same sequence of events. This implies a total ordering on the events received from all of the instances. It also demands that the application instances be deterministic with respect to the event sequence - i.e. if multiple instances, all starting in the same initial state, process identical sequences of events, they will all end up in the same final state. This ordering can be achieved if the platform infrastructure implements some form of floor control, either implicit or explicit (from the point of view of the users).

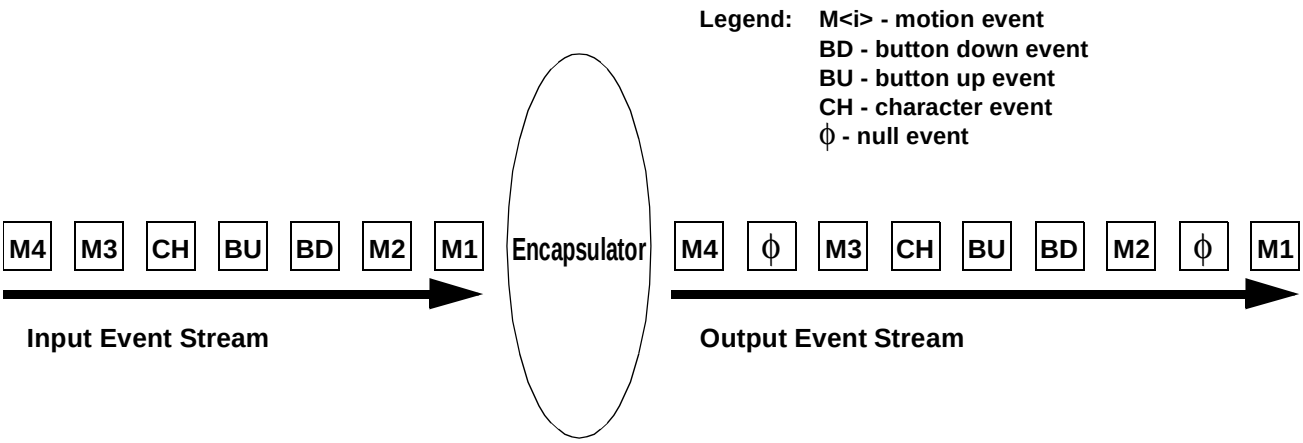
In order for the collaborators to see the same view at the same time, it is also necessary to synchronize the event replay at the different sites. There are a variety of mechanisms that can be used for such synchronization, varying from the extremely pessimistic (2-phase protocol for each event) to the extremely optimistic (totally open loop). Realistic systems will fall somewhere in the middle of this spectrum, using a 2-phase protocol to synchronize around critical events (in terms of the need for interactive synchronization) in the event sequence.

2.3.1 Event Capture

The normal activity when a user types a character or presses a mouse button on a window is the delivery of a window event, appropriate to the user action, to the application that owns the window [13]. That application is normally sitting in an event loop, retrieving each event in turn and performing the processing appropriate to the type of the event.

A critical capability required to enable the SharedApp platform is a non-invasive method for a third-party application to capture the events generated by the window system before they are delivered to the owning application. We have successfully achieved this capability for both X Windows-based applications and for Windows/NT-based applications [19-22]. Given the structure shown in Figure 2 on page 4, each Encapsulator Instance is wired to capture window system events destined for its corresponding Application Instance at session setup time.

Figure 3. Null event stuffing schematic



2.3.2 Event Encoding

Once the events are captured, different types of processing may be performed on the event data. Some examples are:

- while the network traffic resulting from the events is already relatively small compared to shipping graphics primitives and bitmaps, it may be necessary to further compress the event stream - e.g. motion events are generated by the window system by sampling at relatively short intervals of time; shipping each such motion event separately will generate many small packets of network traffic; with no loss of precision, one can compress the motion events gathered over a longer period of time, or until a non-motion event occurs, replacing the sequence of motion events by the last motion event in the sequence
- as stated in Section 2.3 on page 3, the application instances must be deterministic; during our experimentation, we have found that certain windows toolkits (e.g. Motif) take liberties with collapsing successive motion events on the event queue prior to delivering the next event to the application in its event loop; in order to disable this source of non-determinism, we have found it necessary to stuff null-events between successive motion events in the event stream; this null-event stuffing protocol is depicted in Figure 3 on page 5
- event packets generally contain the (x,y) coordinates where the event occurred, usually in terms of pixel number; in order to permit each user to customize his/her window sizes, the events that are distributed by the session controller have had the coordinates normalized; it is the responsibility of the encapsulator to map the normalized coordinates to the actual coordinates for its application instance

2.3.3 Event Mapping

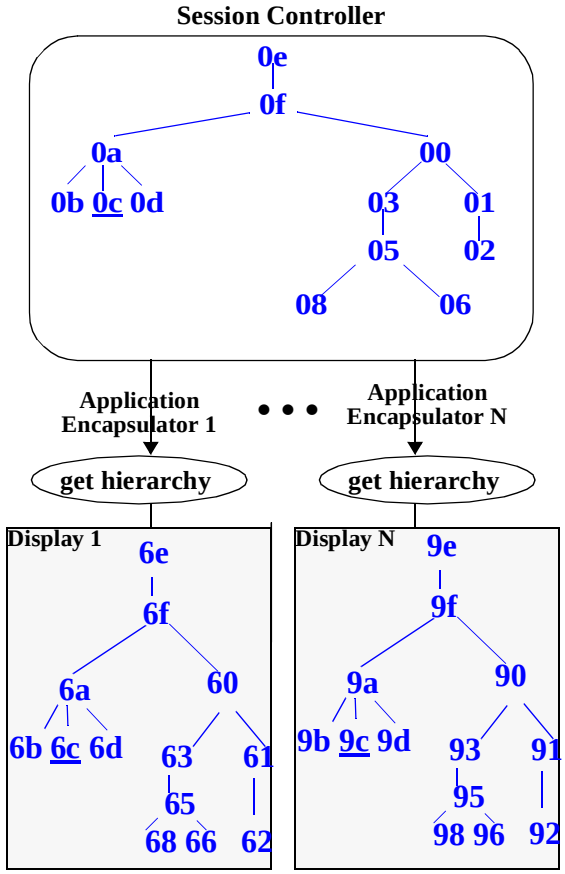
Once an event has been captured and communicated to all of the application encapsulators, we must be sure that the event is replayed into the correct window in each application instance.

Most window systems today support a hierarchical structure among the windows that populate a user's screen; this includes not only windows that the user would think of as windows, but also buttons, menus, and other screen artifacts that, in fact, are implemented as static or dynamic windows.

At session establishment time, the session controller communicates with each encapsulator to discern the window hierarchy for the application instance on that host. This permits the session controller to establish a canonical mapping for the shared application's static window hierarchy such that this event mapping can take place. This algorithm is shown schematically in Figure 4 on page 6.

The platform infrastructure components must also map dynamic windows that come into existence as a result of the actions of the application. Since the application instances are deterministic, we are guaranteed that each of the instances will create the same window in response to the same event stimulus.

Figure 4. Establishing canonical window hierarchy



2.3.4 Event Communication

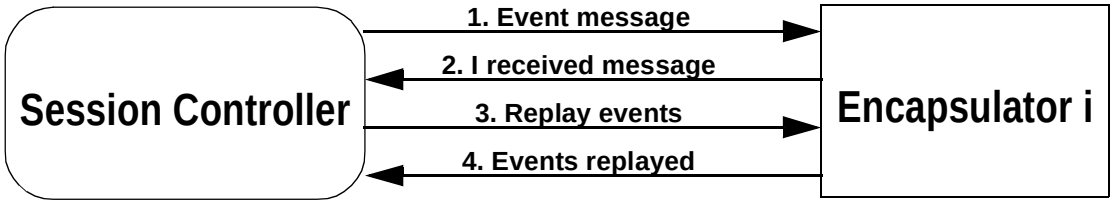
After capturing and encoding input events, the session controller tags (with the target window identifier) and communicates the events to each of the encapsulators; during this process, the controller defines the fixed order of events that all instances will see; it may also group the events into blocks or perform any other activities that may enhance the communication performance and synchronization (See Section 2.3.5 on page 6 for more details on synchronization). Each encapsulator forwards the received events to its application instance; the instances automatically trigger their own event handlers to execute received events. Events are processed just as they would be if the window events had been directly entered into the application windows by a user on that host.

The communication from an encapsulator to the session controller is unicast, while that from the session controller to the encapsulators is inherently multicast. Depending upon the size of the collaboration group, one can choose different mechanisms to achieve the multicast functionality. Due to $o(N^2)$ complexity in the synchronization algorithm, defined in Section 2.3.5 on page 6, we did not find it necessary to use an actual multicast mechanism, preferring to simply perform N unicast operations to the encapsulators.

2.3.5 Event Synchronization and Replay

As mentioned previously, a useful collaboration framework must be able to synchronize the activities of all of the instances in time. In addition to the communication delays that can be experienced in communicating the events to the encapsulators, the end workstations may have differing processing speed characteristics. The algorithm must be able to synchronize with respect to both of these sources of asynchrony.

Figure 5. 2-phase Protocol used by the Platform Infrastructure



The SharedApp platform has chosen to use a 2-phase protocol between the session controller and the encapsulators. This protocol can be likened to the usual 2-phase commit protocols used between a transaction monitor and the participants in a transaction.

The protocol between the session controller and the i^{th} encapsulator is show schematically in Figure 5 on page 7. The first phase consists of communicating the event packet to each encapsulator and waiting for each encapsulator to acknowledge receipt. Upon receiving all of the acknowledgments, the controller then directs each encapsulator to replay the event[s] contained in the event packet to its application instance. The encapsulator responds when it has replayed the event[s] to its application instance.

While the above protocol will synchronize with respect to network delays, it is unable to mask differences in processing speed. If the application which is being used collaboratively is a complex, 3D modeling tool, a single event can cause each application instance to initiate a substantial amount of processing. If there is a severe mismatch in the processing speeds of the collaborators, the slowest ones will get further and further behind if one of the faster machines is actually driving the collaboration (by providing events). What is needed here is a non-invasive mechanism by which we can determine that the application instance has completed its processing.

Unfortunately, such a non-invasive mechanism does not exist. But, we have found a capability that is almost as good - i.e. we can determine that the application instance has removed the last event that was placed on its event queue[1]. This permits us to synchronize over processing speed differences, albeit in an “off by one” manner.

If this sub-protocol is enabled for a session, then the encapsulator delays its “Events replayed” message (message 4 in Figure 5 on page 7) until it has determined that the last event in the event packet has been removed from the event queue by the application instance.

Note that the sub-protocol does not need to be enabled for a collaboration session in which processing speed is not a concern. In this case, the encapsulator sends its “Events replayed” message as soon as it has placed the last event from the event packet on the application instance’s event queue.

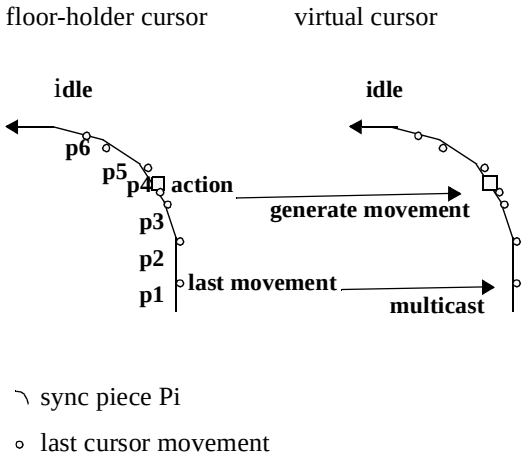
2.3.6 Floor Control

As mentioned in Section 2.3 on page 3, some form of floor control is required to guarantee a coherent, identical stream of events to each application instance. While several types of floor control algorithm are known [8], we have chosen an explicit floor control mechanism for SharedApp. Each application encapsulator provides a floor control window on each participant’s screen. The floor control provides buttons and dialogue areas to permit each user to participate in the floor control algorithm.

A user needs to acquire the floor before events that he/she is generating will be shared with the other instances in the collaboration. Input events from other participants are inhibited. The dialogue area in the floor control window indicates at all times which user currently holds the floor (or that the floor is not held by anyone, if that is the case).

1. No rocket science here - every window system supports both destructive and non-destructive calls to look at the next event on the queue. After placing the last event on the application instance’s event queue, the encapsulator simply performs a non-destructive get next event at a reasonable frequency until it discerns that the last event has been removed from the queue.

Figure 6. Virtual Cursor Movements



2.3.7 Virtual Cursor

During our experimentation with a SharedApp prototype, we discovered that non-floor-holders were unable to completely follow the floor-holder's logic if he/she performed a substantial amount of cursor movement. From these user studies, we determined that we needed to provide a shadow image of the current floor holder's cursor on the displays of the other collaborators. This virtual cursor improves visual perception of collaboration among participants since every cursor movement of the current floor holder's cursor is exactly replicated to the rest of the participants.

As described in Section 2.3.2 on page 5, the encapsulators may compress multiple motion events (those generated when the floor holder moves the cursor) based on collection time or the occurrence of a non-motion event. In order to support this virtual cursor capability, the current cursor position must be communicated with each motion event that is transmitted, such that the virtual cursor can be displayed on each of the collaborators's screens. This mechanism is shown schematically in Figure 6 on page 8.

3.0 The X Windows Implementation

While we have constructed versions of the platform for both X Windows and Windows/NT environments [15-17], we will concentrate on our experiences with the X Windows implementation here.

3.1 Architectural Refinement

Since the X Window system supports remote displays, the Session Controller component, itself, manages the floor control window and event capture from the Application Instance components from Figure 2 on page 4. The floor control window has the Motif look and feel.

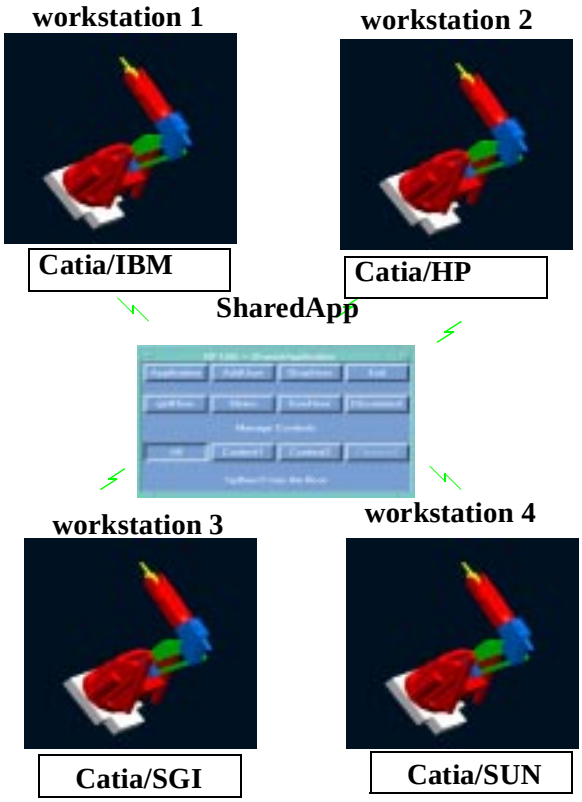
We found it necessary to perform motion event compression for realistic applications, null-event stuffing to override the non-deterministic event elimination performed by the Motif toolkit, and use normalized coordinates to permit users to personalize the sizes of their windows. The virtual cursor protocol is piggy-backed onto the motion event stream.

We use ICCCM communication support to simulate multicast between the Session Controller and each Encapsulator Instance. The 2-phase protocol and last event dispatch detection algorithm are used.

3.2 Application Examples

In order to prove the utility and non-invasiveness of the platform, we have experimented with a number of CAD applications (Pro/ENGINEER 3D, Catia, ICAD, SRC/IDEAS, ...) on a number of UNIX platforms (HP-UX, Solaris, AIX). While it is impossible to convey the collaborative nature of platform use via a static picture, Figure 7 on page 9 shows an example of the type of complex co-design projects that can be undertaken using the platform.

Figure 7. Virtual Team Design



A major automobile manufacturer has been using a version of the prototype with ICAD and SRC/IDEAS to facilitate engineering/manufacturing communication since 1996. Hewlett-Packard is in the process of producing a product from the prototype for general consumption.

3.3 Performance Characteristics

The principal performance measure for a collaborative platform can be summed up in a single question - i.e. does the platform manage the bandwidth available to provide the collaborative user with an interactive experience similar to a stand-alone user?

One can distinguish two sub-questions:

- can the typical network handle the communication traffic generated by the platform?
- do the various protocols introduced by the platform cause a degradation in the interactivity of the application?

Our most telling test of the bandwidth question occurred when we deployed the platform for the automotive manufacturer in 1996. The company's intranet consisted of 56-kbit/second dedicated lines between their engineering facilities and manufacturing facilities. They had previously attempted to use a centralized structure to provide the needed collaboration, but with no success due to the limited bandwidth. Upon installation of the SharedApp platform, within weeks the collaborative application was in daily use. "Using ICAD with SharedApp is as responsive as a telnet virtual terminal session." is representative of the testimonials from the customer. Since then, we have tested SharedApp over networks with as little as 8 kbits/second and found the interactivity provided by the application to be acceptable.

We were quite concerned that the 2-phase protocol would introduce intolerable delays from the interactivity point of view. Our experience has been quite to the contrary - unless you are attempting to use the mechanism over a very over-subscribed network, the communication proto-

cols in use all occur well-within the 0.5-2 second response time that users expect of interactive applications.

If there are severe processing speed mismatches among the workstations involved, and the users' collaboration employs a tool which does a substantial amount of processing, the last event dispatch detection algorithm will cause the collaborative session to run at the speed of the slowest processor. If a user's performance benchmark is relative to a fast processor in stand-alone mode, that user will obviously be disappointed by the performance when a slow processor is added to the collaborative mix. It is our experience that users quickly internalize this situation, and attempt to employ processors with similar processing capability when establishing their sessions.

4.0 Comparison with Other Work

Platforms that rely upon a centralized structure, such JTVOS[1] and SharedX[3], are non-invasive. Unfortunately, since they use the interception approach described in Section 2.1 on page 2, their use for collaboration in which there is a large amount of output will demand substantial communication bandwidth. It is also impossible to use these platforms to support collaboration with respect to Direct Hardware Access (DHA) applications.

Most other platforms that use the replicated approach are invasive - i.e. require that the application be modified in order to participate in a collaborative session. MMConf [8] is the least invasive, in that it only requires that the application be relinked with a substitute library; unfortunately, most users do not have the luxury to be able to perform this relinkage, nor is there sufficient adoption of any one of these platforms such that the major application vendors provide a collaboration-aware version of their applications.

The system most similar in spirit to SharedApp is the VConf system [6, 9]. In this work, Lantz and Lauwers attempted to integrate the Session Controller and the Application Encapsulator components directly into the V system window server. Without floor control, they encountered the expected problems with using their system collaboratively. Lauwers [9] proposed making the applications collaboration-aware to alleviate these synchronization problems. They also did not address the speed-mismatch problems when using processor-intensive applications.

Microsoft currently distributes a product called NetMeeting [18]. NetMeeting differs from SharedApp in the following ways:

- NetMeeting uses a centralized structure, in that one copy of the application executes at the initiator PC, with the participants sharing the initiator output displays.
- NetMeeting is based on sharing of the Windows API. Thus NetMeeting can not support collaboration using DHA applications.
- NetMeeting only runs on NT or Windows 95. SharedApp is designed for platform independence, although the implementation of the platform in an X Windows environment will be different than that for a Windows/NT environment. Note that the protocols will still be the same, so interoperation between platforms is possible.

5.0 Summary

SharedApp is a light weight, non-invasive application sharing platform that enables collaborative design using graphic intensive applications over low bandwidth networks. The technology is based on an event driven mechanism to share a reduced event set dynamically controlled by the current window state. SharedApp has been used to support collaborative CAD/CAM 3D modeling among multiple workstations. Its event-multicast design center permits it to be usable over a variety of network bandwidths (tested as low as 8 kbit/second).

This mechanism is the foundation on which to support virtual co-location and concurrent engineering strategies. The platform infrastructure supports complete synchronization and fast response time through replication without modifying applications. With a single user input source and two-phase protocol, SharedApp maintains a consistent view among the multiple application instances. With a 1-1 window hierarchy tree mapping and standard window system protocols (X Windows and Windows/NT), SharedApp is designed for heterogeneous processing; it is able to share applications across different platforms - e.g. HP, SUN, SGI, IBM. SharedApp is a simple and significant technology to support distributed, real-time, collaborative engineering.

6.0 References

[1] Thomas Gutekunst, Daniel Bauer, *et al*, "A Distributed and Policy-Free General Purpose Shared Window System," IEEE/ACM Transactions on Networking, Vol. 3, No. 1, Feb. 1995.

[2] Thomas Gutekunst and Bernhard Plattner, "Sharing Multimedia Applications Among Heterogeneous Workstations," Proceedings of the Second International Conference on Broadband Islands, 1993, Elsevier Science Publishers B. V.

[3] John R. Portherfield "Mixed Blessings" and "HP SharedX", HP Professional, Volume 5, Issue 9, September, 1991.

[4] Uwe Jasnoch, *et al*, "Shared 3D Environments within a Virtual Prototype Environment," Proceedings of WET ICE '96.

[5] M. Sobolewski, *et al*, "Functional Specifications for Collaboration Services," Proceedings of 3rd IEEE Workshop on Enabling Technologies: Infrastructure of Collaborative Enterprises, Apr. 1994.

[6] J. Chris Lauwers, Thomas A. Joseph, Keith A. Lantz, "Replicated Architectures for Shared Window Systems: A Critique," Communications of the ACM, 1990.

[7] Daniel Garfinkel, Randy Branson, "A Comparison of Application Sharing Architectures in the X Environment," Proceedings of Xhibition 91.

[8] Terrence Crowley, Paul Milazzo, Ellie Baker, Harry Forsdick, and Raymond Tomlinson, "MMConf: An Infrastructure for Building Shared Multimedia Applications," Proceedings of CSCW 90.

[9] K. A. Lantz, "An Experiment in Integrated Multimedia Conferencing," Proceedings of CSCW 86.

[10] S. R. Ahuja, J. R. Ensor, S. E. Lucco, "A Comparison of Application Sharing Mechanisms in Real-Time Desktop Conferencing Systems," p238-248, Communications of the ACM, 1990.

[11] C. Ellis, S.J.Gibbs, "Design and use of a group editor," Working Conference on Engineering for Human-Computer Interaction, 1989.

[12] Stephen Zabele, Steven I. Rohall, Ralph L. Vinciguerra, "High Performance Infrastructure for Visually-intensive CSCW Applications," Proceedings of CSCW 94.

[13] Communications of ACM, Special Section on "Graphical User Interfaces: The Next Generation", Apr. 1993, Vol. 36, No. 4.

[14] E. C. Cooper, "Replicated Distributed Programs", In Proc. 10th Symposium on Operating Systems. p63-78, ACM, Dec 1985.

[15] Sventek and Hao, "Collaborating Using Your Favorite 3D Application," Proceedings of the Third ISPE International Conference on Concurrent Engineering, Toronto, Ontario, Canada, August 1996.

[16] Hao, Lee, and Sventek, "A Light-Weight Application Sharing Infrastructure for Graphics Intensive Applications," Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5), Syracuse, NY, August 1996.

[17] Hao, Glajchen, and Sventek, "SmallSync: A Methodology for Diagnosis and Visualization of Distributed Processes on the Web," Proceedings of the IEEE Seventh International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Stanford University, CA, June 1998.

[18] http://www.microsoft.com/netmeeting/default.htm, the NetMeeting home page.

[19] U.S. Patent on "Method and Apparatus to Sense and Multicast Window Events to a Plurality of Existing Applications for Concurrent Execution," Hewlett-Packard, April 1998.

[20] U. S. Patent pending on "A Mechanism to Control and Use Window Events Among Applications in Concurrent Computing," Hewlett-Packard, Oct 1994.

[21] U. S. Patent pending on "A Mechanism to Synchronize 3D Motion Views Among a Plurality of Existing Applications in Concurrent Engineering," Hewlett-Packard, Oct 1995.

[22] U. S. Patent pending on "A Mechanism to share Cursor for Concurrent Execution and Consistent Graphical Views Among a Plurality of Existing Applications," Hewlett-Packard, Dec 1995.