Table of Contents

The Design Space of Collaboration Infrastructures

Prasun Dewan
Microsoft[1]

Abstract

Implementing collaborative applications is challenging because of the range of functions that must be offered to support collaboration and the fact that these functions must be provided in an efficient manner to both small and large collaborative groups. Therefore, it is important to provide infrastructures to automate the implementation of these functions. This paper presents a framework for describing, analyzing, and developing such infrastructures by identifying the issues that arise in their design, the approaches taken in existing systems to address them, and the pros and cons of competing approaches. Some of the issues considered are the application I/O layers that are shared, the sites on which a shared layer is replicated, the mechanisms for keeping replicated layers consistent, session management, access and concurrency control, firewall traversal, interoperability, and componentization. The framework is used to classify commercial standards/products such as SIP, T 120, NetMeeting, Web Services, XAF, Webex, PlaceWare, Groove, and PresenceAR; and several influential research systems. It is also used to identify several new directions for both commercial and research systems.

1 Introduction

While few people today dispute the importance of supporting collaborative applications, there is no general agreement on or even understanding of the infrastructures for supporting them and terminology used for describing them. For example, Groove is classified as a "peer-to-peer" system while PlaceWare is described as a "client-server" system, and it is assumed that the success of these two systems will be proportional to the success of arbitrary "peer to peer" and "client server" systems, such as Tapestry(Ben Y. Zhao)and the Web, respectively. A close look at the two systems, however, shows that Groove too has servers, and more important, architecture is only one of the many important aspects in the design and evaluation of a collaboration infrastructure.

Perhaps the major reason for this problem is that these systems are complex, the documentation on commercial systems and standards often does not provide motivation or separate the concepts from the obvious details, and comprehensive technical comparisons of these systems have not been made. This paper addresses this problem. The author spent several months wading through the documents available on commercial systems and standards and integrated this information with his knowledge of research systems to develop a framework for describing, analyzing, and developing collaboration infrastructures. The framework consists of a set of dimensions representing the issues that arise in the design of collaboration infrastructures, the approaches taken in existing systems to address them, the pros and cons of competing approaches, and holes in this dimensionalized space that have not been filled by existing commercial and research systems. Some of the issues considered are the application I/O layers that are shared, the sites on which a shared layer is replicated, the mechanisms for keeping replicated layers consistent, session management, access and concurrency control, firewall traversal, interoperability, and componentization. The framework is used to classify commercial standards/products such as SIP (Sinnreich and Johnston), T 120(DataBeam), NetMeeting, Web Services, XAF, Webex (Webex), PlaceWare(Placeware 2000), Groove(Groove), and PresenceAR (Reality)and several influential

[1] On sabbatical from UNC-Chapel Hill (dewan@cs.unc.edu)

research systems. It is also used to identify several new directions for both commercial and research systems.

The paper does not explore the user interfaces of collaborative applications, aspects of which are addressed in a companion paper(Dewan). A previous paper also addressed implementation of collaborative applications (Dewan 1993), but it did not cover any of the commercial systems surveyed here, or the research systems developed since the paper was published. We have tried to repeat as little as possible information in previous surveys.

The rest of the paper is organized as follows. It first defines collaborative applications and then presents a layered modeling of single-user applications which is next used as a basis for modeling the architecture of collaborative applications. This model is then used to present various infrastructure design issues and approaches. The paper concludes with a summary and new commercial and research directions.

2 Collaborative Applications

2.1 Definition

A straightforward definition of a collaborative application is one that interacts with multiple users. By this definition, however a multi-user operating system is a collaborative application, even though the purpose of the operating system is to give users the illusion that no one else is using the system. To highlight the features that distinguish collaborative applications from single-user ones, they have been characterized as applications that:

- make users aware they are part of a group.
- help people work together.
- provide a shared environment.
- support groups engaged in a common task.

These characterizations give a good intuitive feel for the purpose and nature of collaborative applications, but have subjective terms such as “common task” and “work together”. For the purposes of this article, we shall objectively define a collaborative application as one that presents a set of user interfaces that are coupled with each other, that is, each component user-interface can change as a result of the actions taken in one or more of the other component interfaces and/or can cause changes to one more of the other interfaces (Figure 1).

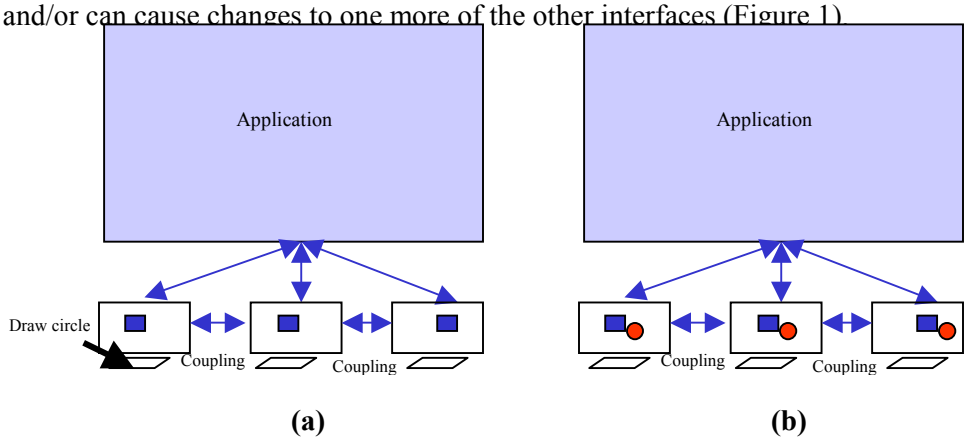


Figure 1 Coupled User Interfaces in a Collaborative Application: Draw command (a) creates circle on all displays (b)

Typically, a different user would interact with each of the component interfaces, though it is possible for the same user to interact with multiple interfaces on different computers (at possibly different times), or multiple users sharing a computer to interact with a single interface. Ideally, coupled user interfaces would help their users work together more productively than uncoupled ones, though there is no guarantee that this may happen as some of the users are free to use the coupling to "flame" to others or exhibit other anti-social behavior, deliberately or accidentally.

By our definition, traditional email and multi-user file systems are examples of collaborative applications as my action of sending email to you causes your email interface to display the message and my action of saving modifications to a shared file influences what you see the next time you open the file. However, we will focus here on the more modern collaborative applications such as instant messaging, gaming, whiteboards, and distributed presentations.

2.2 Synchronous vs. Asynchronous, Full vs. Partial Coupling

In Figure 1, the coupling between the users is synchronous in that the user who enters the input sees the output at the same time as other users (modulo networking displays). Asynchronous coupling would allow users to see the output at different times. Moreover, some of the user-interface state may be (eventually) shared with other users while the remaining is uncoupled. Figure 2, 3 and 4 take the example of the NetMeeting whiteboard to illustrate synchronous, asynchronous, and partial coupling, respectively.

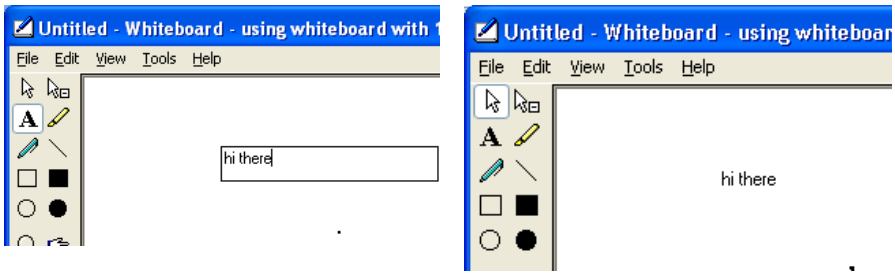


Figure 2 Synchronous Sharing: As the left user types, the right user sees the result

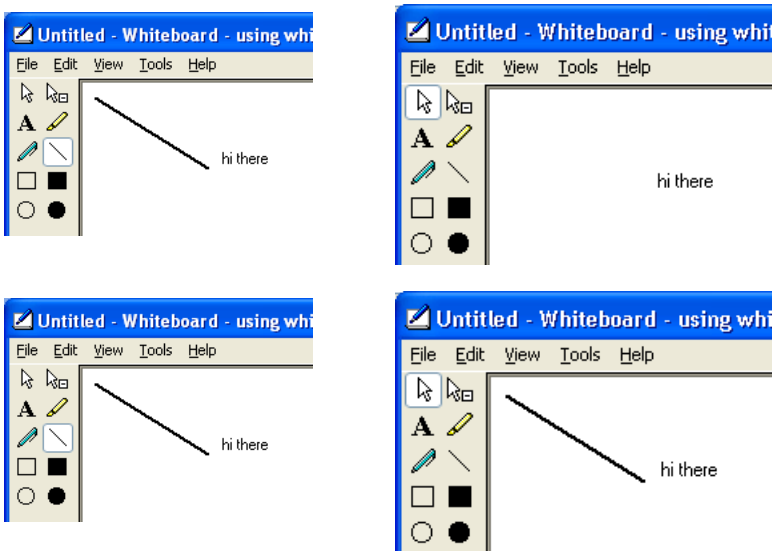


Figure 3 Asynchronous sharing: As the left user draws the new line, the right user gets no feedback (Top windows). When the line is completed, it is shown to the right user (bottom windows)

What part of the state should be shared with another user and when it should be shared is a user-interface design issue covered in depth in(Dewan 1993).We bring this issue here to show that an infrastructure should be flexible in how it answers these two questions. As we wee below, some infrastructures are more flexible in this respect than others.

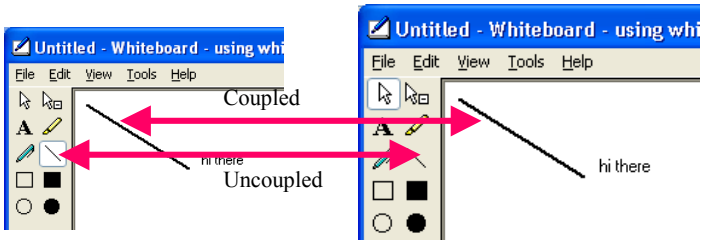


Figure 4 Partial coupling: The line is eventually shared but the tool selection is not.

3 Layered Architecture

As mentioned earlier, architecture is an important issue in the design of a collaboration infrastructure. To reason about it, we will use a layered model of application implementation, focusing first on layers supporting single-user interaction and then on those needed for collaboration. This discussion is a condensation of (Dewan 1998)

3.1 Single-user Layers

The idea of creating a layered abstraction on top of a hardware device is not new – for example the network device is managed by a set of communication layers (left stack, Figure 5). Here we focus on the set of layers that manage the I/O device such as the mouse, keyboard, and screen (right stack, Figure 5). As shown in the figure, an application may have multiple layer stacks managing different devices.

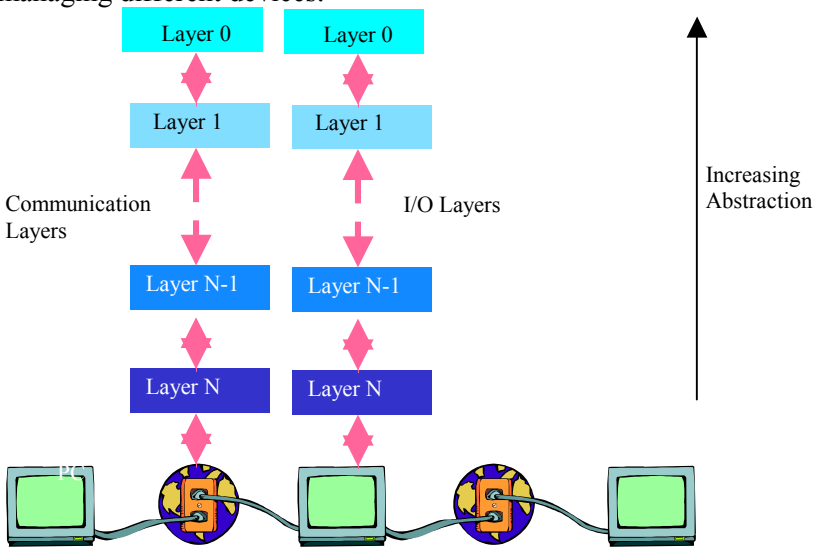


Figure 5 Just as a stack of layers manage the network device, multiple layers manage the I/O device

In general, input is abstracted by a set of layers before it reaches the application. Conversely, the output produced by an application is rendered by a set of layers before it is displayed to the user. In general, the top layer defines one or more abstractions whose presentations are edited by the

user. The process of creating an editable presentation of an abstraction is carried out in multiple stages by the different layers between the top layer and the hardware. Each of these layers renders an abstraction in the layer above into an interactor, which in turn serves as an abstraction for the layer below. An interactor consists of a transformation of the information in the abstraction that is closer to its screen presentation plus some additional "syntactic sugar," which has no representation in the abstraction and thus cannot be reverse transformed back into any aspect of the abstraction.

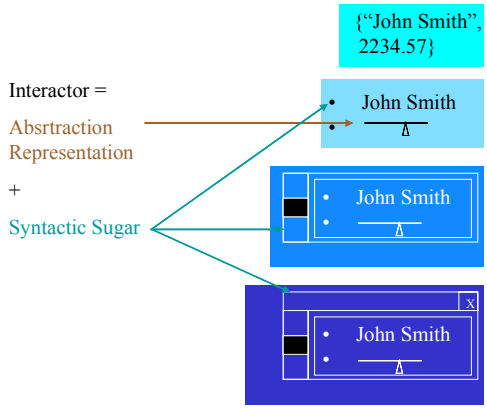


Figure 6 Creating an editable presentation of an abstraction via a series of transformations

Figure 6 illustrates this process. Suppose the top, "model" layer (Krasner and Pope August/September 1988) defines an abstraction consisting of a string and a number. The next, view layer, may transform these two values to an abstract text field and slider, respectively, and add bullets next to them as syntactic sugar. The subsequent, toolkit layer, may choose specific toolkit widgets for the abstractions it receives, and add a containing scrollable widget around them as syntactic sugar. Finally the window layer may transform the widgets into windows and add a window border around the top-level widget as syntactic sugar.

3.2 Shared layer and centralized and replicated architectures

To support collaborative interaction, we must pick some layer we want logically shared among the users. Once this layer is shared, all layers above it are automatically shared since they are abstractions of it. The lower layers, however, can diverge as they may transform (the abstractions in) the shared layer differently. The former layers are referred to as the program component, since they are nearer the semantics of the application, and the latter as the user-interface component, since they are nearer the I/O devices. This is consistent with the viewpoint that the part of the application that is user-dependent is the user interface and the remainder is the semantics or program component. Figure 7 illustrates this discussion. Since the user interface component may diverge, a separate instance of it is created for each user-interface of the application. The program component, on the other hand, which is logically shared, can be physically shared or replicated, as shown in Figures 8 and 9, respectively, which also show the infrastructure components enabling these two architectures.

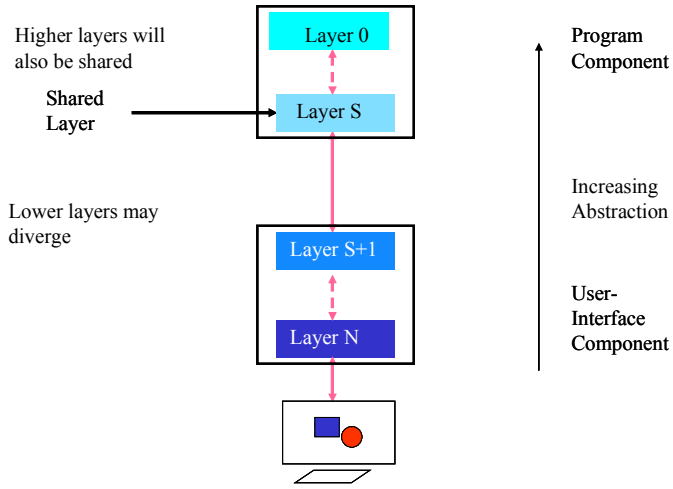


Figure 7 Choosing a shared layer divides the application into a program component, which is logically shared, and a user interface component, which may diverge.

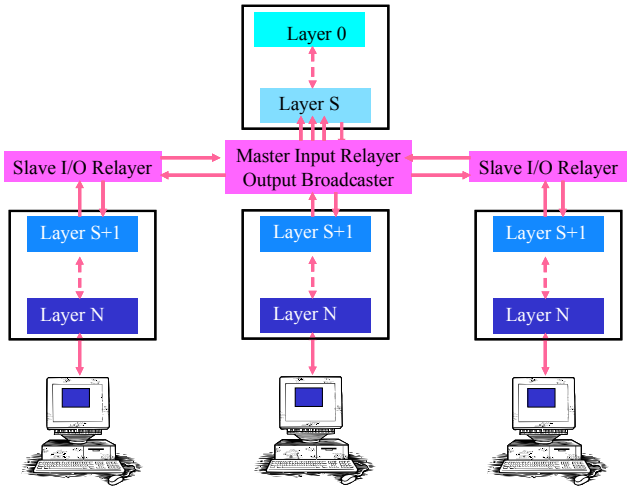


Figure 8 Centralized Architecture: Program component is physically shared.

In the centralized architecture, the computer executing the program component is called the master and the remaining sites are called slaves. An infrastructure module on the master receives input events from all user interface components and forwards them to the program component. Conversely, the module receives output from the program component and forwards it to all of the computers. An infrastructure module on the slave is responsible for relaying I/O between the local user interface and the master module.

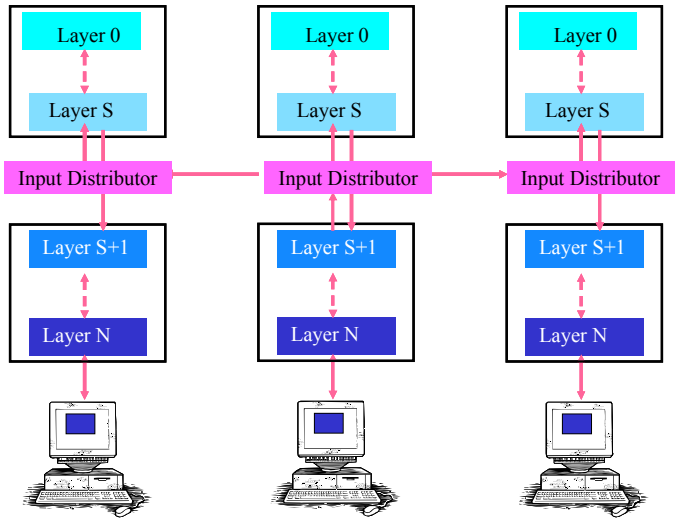


Figure 9 Replicated Architecture: Program component is physically replicated

The replicated (or peer to peer) architecture is more symmetric. Each site executes both the program component and the user interface component. An input event generated by a user-interface component is forwarded by infrastructure modules shown in the figure to all the program components to keep them consistent. Usually input is delivered to the local replica without (immediately) synchronizing with other sites, relying on concurrency control to prevent concurrency problems or merging to do late synchronization. Output from a program component, however, goes directly to the local user-interface component.

In Figure 8 and 9, we have assumed that input/output broadcasters are on user sites. In some systems such as PlaceWare and Webex, they are on special sites. These are dedicated computers provided by (a) users' organizations in the "server" model, and (b) an external organization providing (and charging for) the service in the "service". In the former case, user sites are connected to the broadcasting sites by a LAN link and in the latter case by a WAN link.

3.3 Shared layer and architecture in existing systems

The layer and architecture are two important dimensions for classifying collaborative systems, and use them for characterizing existing collaborative systems in Table 1.

System	Layer Shared	Architecture
NLS (Engelbart September 1975)	Screen	Centralized
VNC (Li, Stafford-Fraser et al. March 2000)	Screen	Centralized
XTV(Abdel-Wahab and Feit April 1991)	X Windows	Centralized
VConf /Dialogo(Lantz December 1986)	V Windows	Replicated
Rapport Centralized (Ahuja, Ensor et al. 1990)	X Windows	Centralized
Rapport Replicated (Ahuja, Ensor et al. 1990)	X Windows	Replicated
NetMeeting Application Sharing	T 120 Screen, Windows	Centralized
Webex Application Sharing (Webex)	Webex Windows	Centralized
PlaceWare Application Sharing(Placeware)	PlaceWare Windows	Centralized
GroupKit(Roseman and Greenberg 1996)	TK Toolkit	Replicated
Habanero (Chabert, Grossman et al. June 1998)	Java AWT Toolkit	Replicated
JCE (Abdel-Wahab, Kim et al. April 1999)	Java AWT Toolkit	Replicated
Rendezvous(Hill, Brinck et al. June 1994)	Model	Centralized

Suite(Dewan and Choudhary October 1992)	Model	Centralized
Groove(Groove)	Model	Replicated
NetMeeting Whiteboard	Model	Replicated
PlaceWare PPT(Placeware)	Model	Centralized
Webex PPT (Webex)	Model	Replicated
Groove PPT (Groove)	Model	Replicated

Table 1Shared Layer and Architecture of Existing Systems

Here we assume the following I/O layers: physical screen, window, widget, and model. As we will see later, a view layer may exist between the widget and model layer. As shown in the table, separate versions of Rapport were created to support the two architectures. As we will see later, it is possible to create a single implementation that supports both architectures and allows dynamic transitions among them. Some of these frameworks, such as T120, Webex and PlaceWare allow sharing of heterogeneous windows by supporting an abstract window model that maps to multiple concrete window models. We will discuss later some of the issues that arise in the design of interoperating heterogeneous collaborative systems. In Table 1 we have included not only collaborative infrastructures but also specific applications such as the NetMeeting whiteboard, and PlaceWare and Groove PowerPoint applications.

4 Evaluation Criteria

What are the consequences of choosing a specific shared layer and the architecture? There are several criteria we can consider to evaluate these decisions:

- Coupling flexibility: How much flexibility does the infrastructure provide in determining which part of the application state is shared and when it is shared?
- Concurrency: How much concurrency does it allow in the activities of the users of an application, and what correctness guarantees are provided regarding concurrent input?
- Access control: How much control does it allow in specifying access rights of the various users?
- Automation: How much of the sharing semantics is supported automatically by the infrastructure.
- Ease of learning: How easy to use and learn are the abstractions provided by the infrastructure?
- Reuse: Can the infrastructure support sharing of existing single-user software, possibly without recompiling?
- Interoperability: Can it support sharing among heterogeneous implementations of a layer?
- Performance: How well does it perform along the following dimensions?
    - Computation load and scalability: How much computation load is put on the sites interacting with the application? In particular, how well does it scale to sessions with a large number of users?
    - Network load: How much network bandwidth is required to support the sharing?
    - Feedback time: How long does it take for the users at various sites to get feedback to their input?
    - Feedthrough time: How long does it take for the results of a user's actions to reach other users?
    - Start time: How long does it take to start a user interface?
    - Completion time: How long it does it take for a collaborative task to complete?

5 Centralized vs. Replicated

5.1 Network load

Recall that in the replicated architecture, input is broadcast, while in the centralized architecture, output is broadcast. In general, sending input on the network consumes less bandwidth than sending output, as the former is created by humans and the latter by the much faster computers. For example, a single input event may result in a video stream being output. Thus, the replicated architecture can be expected to put less network load than the centralized architecture.

5.2 Feedback

In the replicated case, the feedback time is a function only of the time it takes to compute it on the local computer. In the centralized case, we must separately consider the local user at whose site the program is centralized and the remote users sharing this program from other sites. The feedback time for the local user, as in the centralized case, is a function of the time it takes to compute it on the central site. For the remote user, we must add to it the roundtrip communication time between the remote and hosting site, which in case of server or service broadcast, involves traversing an extra LAN or WAN link between the two sites. In the server or service distribution model, the roudtrip time between participating sites involves two roundtrip times, one from each of the site to the dedicated distributor machine. Thus, in the central case, the feedback time can be much higher than in the replicated case. Similarly, in the server or service distribution models, the feedback time can be much higher than in the normal (integrated processing and distribution) model.

Some of these differences can be seen when using existing commercial systems. Remote feedback time for window sharing is:

- noticeable when using NetMeeting's direct distribution model and centralized architecture.
- is intolerable when using PlaceWare's service-based distribution model and centralized architecture.

Similarly, feedback time for PowerPoint presentations is:

- not noticeable when using Webex's and Groove's replicated sharing of the model layer.
- is noticeable when using NetMeeting's centralized window sharing.

Of course we are using different implementations here, but in all of these cases, the overhead of sending I/O through the infrastructure seems to be pretty fast and thus the delays are most probably attributed to network communication. Moreover, in the PowerPoint comparison, we are using in one case the replicated model layer and in the other case the centralized window layer. However, the input events in a replicated window and model cases (key press and next slide command, respectively) can be expected to be roughly the same size and more important, fit within one packet.

The replicated architecture is specially important when users are trying to access a stream at an external site that has real-time requirements, as illustrated by the experience with building the collaborative video application of (Cadiz, Balachandran et al. 2000), which allows users to collaborative play, stop, rewind and fast forward a video stream stored in a server. Figures 10 shows its user interface and Figure 11 shows centralized and replicated implementations of this application.

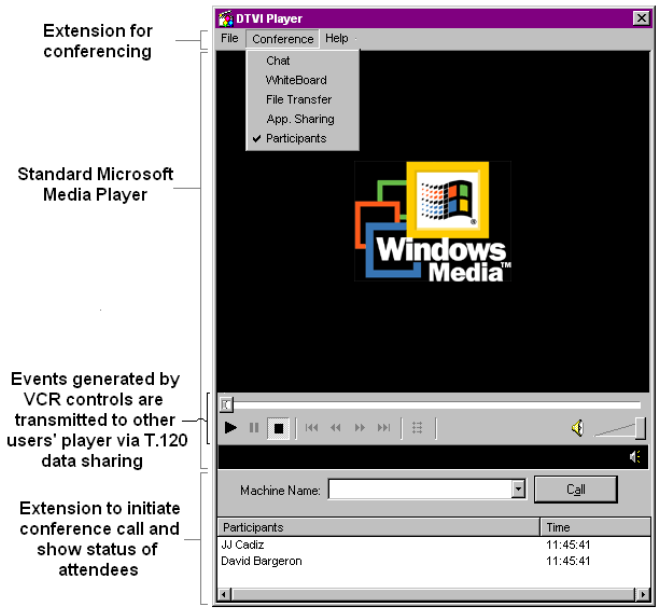


Figure 10 Collaborative Video Viewing

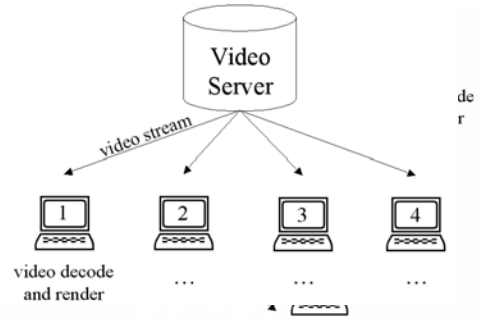


Figure 11 Centralized (left) and Replicated (right) implementations

Video from the server is transmitted directly to a user site in the replicated implementation and via the central user site in the centralized implementation. The additional transmission and processing done at the central site made the performance unacceptable in the central case. Therefore, the replicated implementation was used for this application.

For computationally intensive tasks such as a Checker's game, the relative computing power of the hosts on which the applications execute may be an issue. In the replicated case, the computation power of the local computer determines the feedback while in the centralized case only the computation power of the central host matters. Thus, when the slave sites have relatively low power (because they are PDAs for example), the central architecture can give better performance.

5.3 Feedthrough

This is the time it takes for a remote site to get results computed in response to local input. In the centralized case, it consists of one-way input communication time from local site to central host, plus computation time, plus one-way output communication time from central host to remote site. In the replicated case, it is local computation time plus one-way input communication time to remote site.

The feedthrough time is less significant than remote feedback time because it does not affect the active user's response. For example, feedthrough time when making power point presentations is not noticeable when using the replicated data model of WebEx or Groove but is noticeable though acceptable when using the centralized data model of PlaceWare. Similarly, feedthrough time for drawing operations executed on the hosting computer is noticeable but not unbearable when using PlaceWare's centralized window sharing with communication via PlaceWare servers.

It must be kept low, however, to reduce the divergence between local and remote displays leading to "can you see it now?" queries, which interrupt the natural flow of the collaboration and increase the task completion time

5.4 Task completion time

Task completion time depends on

- local feedback, assuming a hosting user inputs.
- in the centralized case, remote feedback, assuming the non-hosting users input, which may not be the case in many collaborations – especially presentations.
- feedthrough, if interdependencies exist in the task, that is, a user's actions depend on those of the others, which may not be the case in some collaborations – such as the idea generation phase of brainstorming (Stefik, Foster et al. January 1987).
- Sequence of inputs and the users who entered them.

(Chung and Dewan. 2001) show that in most cases the replicated architecture gives better completion time except when the task is computation intensive and the central computer is much faster than the others. As mentioned before, intuitively, when the computation powers are asymmetric (e.g. server/desktop, desktop/PDA configurations) and the task responsibility is asymmetric, the asymmetric centralized architecture performs better.

5.5 Load and scalability

Similarly, when the central computer is more powerful, the central architecture scales better as the task of broadcasting data to a large group can be computation intensive. (Of course, when a separate computer does the broadcast, both architectures scale equally well.) Scalability is important as the immediate target of many commercial systems is to allow presentations to the whole company, and someday they may even be aiming for national and international presentations. On the other hand, many useful collaborations involve less than 10 users, such as joint editing or a design review. To draw an analogy, most phone calls involve two users; very few are conference calls.

Therefore, the PresenceAR (Reality) goal is to support both the replicated and architectures, and dynamically transition to the central/replicated architecture as the conference gets large/small.

5.6 Correctness of concurrent input

This is not an issue in systems that support floor control, as only the floor holder can interact with the application. Other systems must ensure that the results of concurrent input events are correct. In particular, input events should appear to all of the users to have been processed in the same order. This is automatically ensured by a central architecture but not a replicated architecture, which processes local input immediately without synchronizing with other replicas.

Figure 12 shows what can go wrong in a replicated system. Two users are involved in a joint editing session and the current text is "abc". They enter the commands Insert d, 1 and Insert e, 2

concurrently. Their replicas process the local commands immediately, resulting in their local copies of the string becoming “dabc” and “aebc” respectively and then exchange these commands. After both replicas execute both commands, the two strings are different, “deabc” and “daebc”, respectively, because they executed the commands in different orders.

In the replicated architecture, concurrent commands will be executed in different orders by the replicas as long as local feedback is given without synchronizing with other sites. To correct this problem, a synchronization or merge step must be carried out by a merger module before a remote command is executed so that all replicas appear to the users to have the same sequence of commands. This module may transform the received remote command before executing it. Figure 13 shows an example of a merger for the above example. User 2’s input command, Insert e,2 is transformed to Insert e, 3 before it is executed by User 1’s replica, while User 1’s input is not transformed by User 2’s replica. As a result, after execution of the two commands, both users see the text “daebc”.

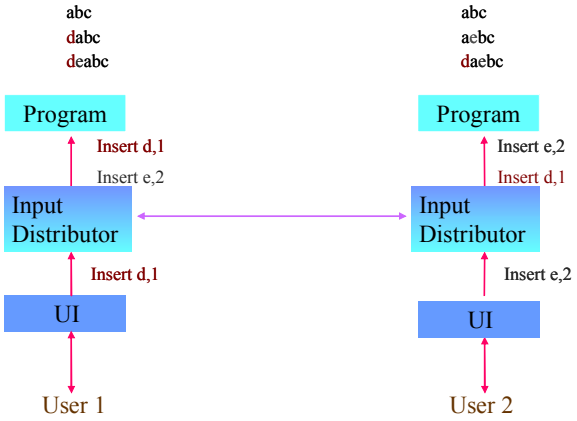


Figure 12 Synchronization Problems with Concurrent Input

In Figure 13, the merge module is itself replicated. Several algorithms have been developed for distributed merging, but they have been hard to get correct (Sun and Ellis Nov 1998). Therefore, some systems use support centralized merging (Munson and Dewan June 1997, Nichols, 1995 #946). It may seem incongruous to support centralized merging in a replicated system – do all of the replication advantages not disappear? In fact, feedback time is not affected, it is only feedthrough that traverses an extra communication link. Centralized merging is consistent with the service model of collaboration and is therefore implemented in PlaceWare.

Notice that in this example, the merger needed to know about text strings. In general, the more application semantics the merger has the better job it can do. This implies, as discussed in greater depth later, the higher the shared layer, the more application semantics are available to the merger. Infrastructures such as shared windows systems that support sharing of low-level layers do not provide merging, and simply serialize input through floor control.

Merging is also called optimistic concurrency control, though that is a slight misnomer, because unlike traditional concurrency control, it does not support serializable transactions. In fact, most merging systems have no notion of a user-controlled transaction. We will later look in depth at a comparison of merging with traditional forms of concurrency control.

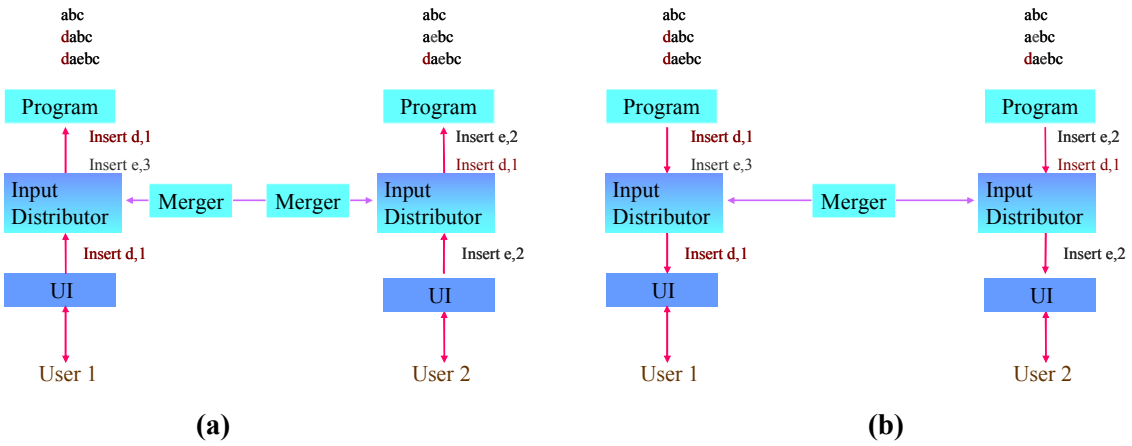


Figure 13 Replicated (a) and Central (b) Merging

5.7 Correctness and performance of replicated input processing

Correctness problems can also arise because of replicated input processing, as well as performance problems. Consider reading of an external centralized resource. As shown in Figure 14 (a), all replicas access the resource at the same time, creating a bottleneck at the central site. Writing such a resource is even worse as the same information gets written multiple times, one by each replica, as shown in Figure 14(b).

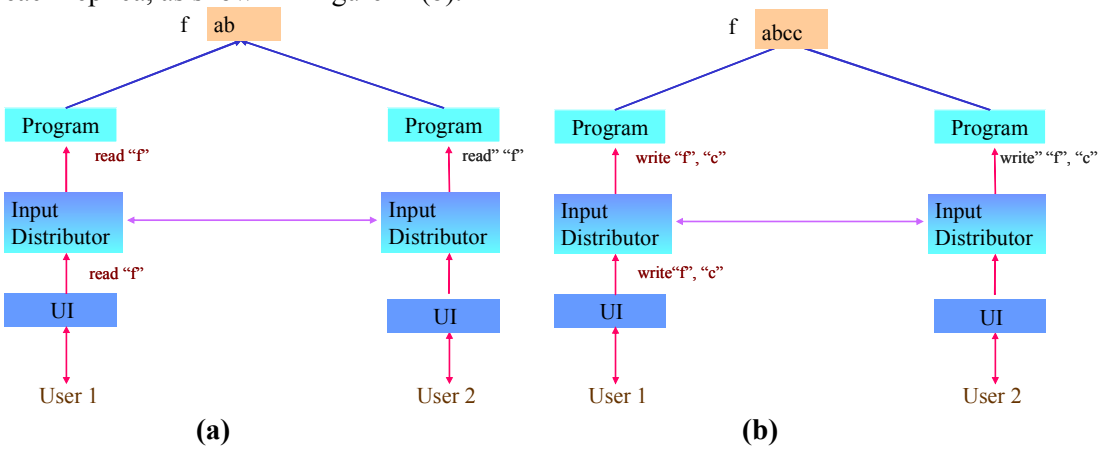


Figure 14 Bottleneck & Correctness Problems in Reading (a) and Writing (b) a Central Resource

The solution to this problem, adopted by Groove and Webex PowerPoint is to replicate external resources when possible (Figure 15). Webex PowerPoint completely replicates the PowerPoint file at user's site at the start of each collaborative session. Groove allows the replication to occur any time before the collaborative session and sends diffs to keep the replicas consistent if they are modified before the session. This approach increases the time required to start a new user session with a collaborative application, which depends on the network connection and the system. In one Webex session in which the author participated, replicating the larger PowerPoint file the author had (around 7MB in size) took less than a minute, which is a small fraction of the typical presentation time.

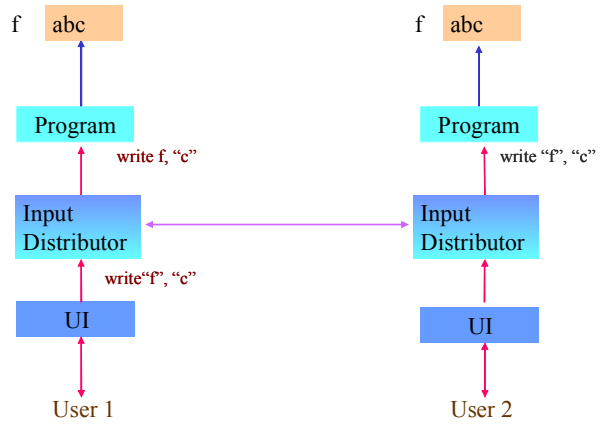


Figure 15 Replicating external resources to address bottleneck and correctness problems

However, replicating external resources is not sufficient to solve the problem shown in Figure 16, where a command to mail a message results in the message being mailed by each replica.

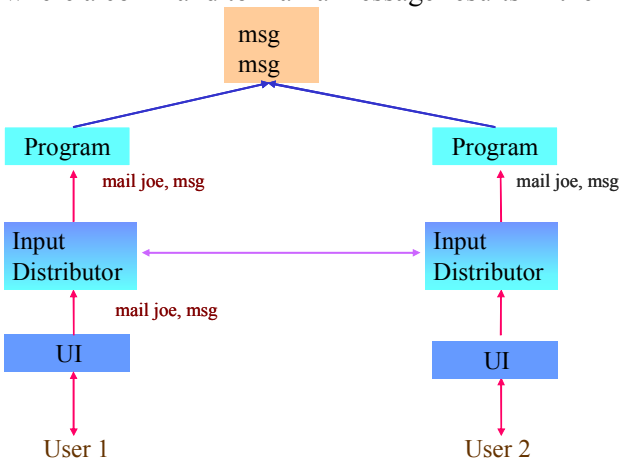


Figure 16 Correctness Problems with Executing a Non-Idempotent Operation

The problem here is that mail or writing to a centralized external resource is an non-idempotent operation, that is, an operation whose multiple executions are not equivalent to a single execution. By replicating the external resource, we changed the operation – accessing a replicated instead of a central resource - but it is not always possible to do so, as the mail example shows. To ensure that non-idempotent operations in general are executed only once, we need to break away from the pure replicated architecture.

Figure 17 shows one such approach, implemented in Groove. The program is divided into two components, one that executes idempotent operations, which is replicated, and another, called a "Bot," that executes non-idempotent operations, which is centralized.

Figure 19 shows an alternative approach, implemented in Suite. Again the program is divided into a replicated and a centralized component. The replicated component contains the subset of the functionality of the centralized component that is needed to drive the local user interface. Information is communicated between the user interface and central program component via the replicated program component. Thus, the replicated program component can be considered an extra user-interface layer in our layered model, except that it does not transform the abstraction in

the central component, and is simply an intermediate proxy layer. This two-level program approach provides more centralization than the Groove approach of an independent non-idempotent component – it forces even idempotent operations to go to a central component, thereby increasing the feedthrough times for such operations. On the other hand, it is consistent with the idea of a central merger and central broadcaster in that it does not decrease the performance in systems implementing these concepts.

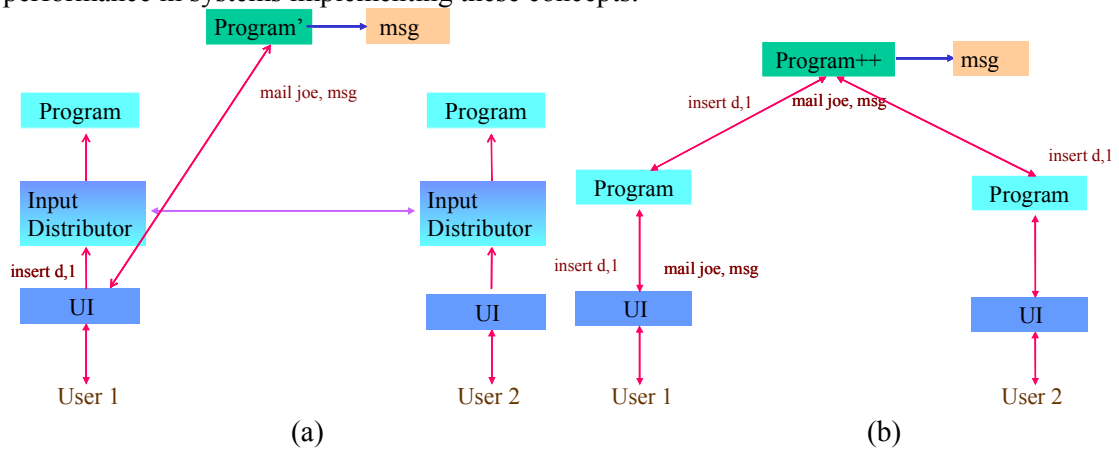


Figure 17 (a) Independent and (b) Two-level Program Component for Non-Idempotent Operations

We have seen above our first examples of a hybrid architecture combining elements of the central and replicated architecture. We will see below other ways in which the two architectures are combined.

6 Sharing Different Layers

Let us consider first some general implications of sharing layers at different levels and then the details of sharing layers typically found in applications today: screen, window, toolkit and model.

6.1 Sharing Low-level vs. High-level Layer

Sharing a layer nearer the model results in:

- greater view independence, as a larger number of layers are in the user-interface component, and thus a larger number of layers can diverge.
- finer-grained access and concurrency control, as the higher the sharing abstraction, the more it can be partitioned into logically meaningful units.
- less network load, as the higher the shared layer, the less the syntactic sugar in the interactors it creates, and usually, the more compact the representation of information in them.
- easier to solve replication problems as a higher layer has more semantic knowledge to solve them.

On the other hand, sharing a layer nearer the device results in:

- higher convergence in the displays of the users, which gives them more "referential transparency," that is, the ability to refer to the viewed objects in terms of their display. For example, referring to an object as the "green circle" has no meaning if different user interfaces colors objects differently.
- higher reusability and interoperability, as usually lower the layer, the more the chance that it is a standard one. For example, there are multiple toolkits built on top of the X window system, and there are multiple window systems built on top of an I/O device.

In either case, fixing the shared layer limits the sharing flexibility as users are constrained to share a specific abstraction – high or low. We will look later at systems that support multiple levels of sharing.

Let us now consider in more depth sharing of the specific abstractions.

6.2 Window Sharing

The window layer, unlike the screen layer we will discuss later, divides the display into separate windows. Based on X terminology, we will refer to the module that implements it as the window server (though in fact, it may be a part of the kernel as in Microsoft Windows) and the application layers above it as the window client. Window sharing is sharing of the window client layer.

6.2.1 Basis Centralized Window Sharing Architecture

Figure 19 shows the general central architecture shown earlier (Figure 8) instantiated for window sharing.

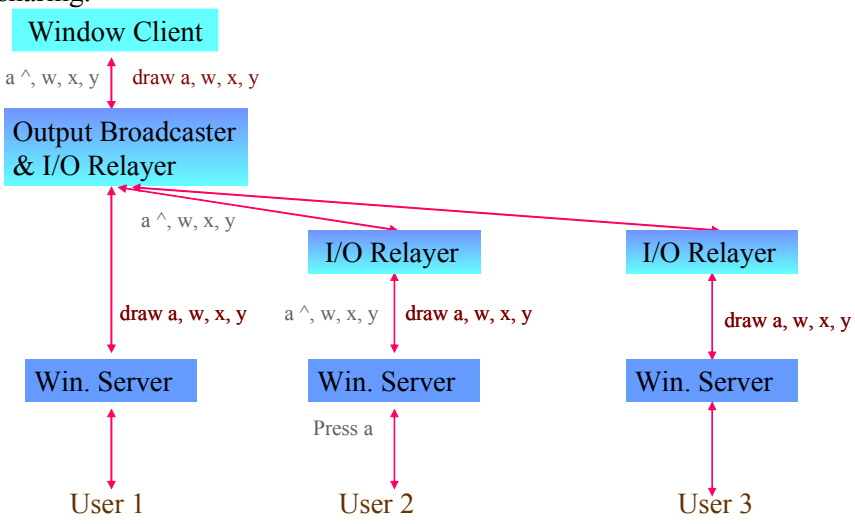


Figure 18 Basic Architecture for Centralized Window Sharing

Let us follow the event sequence shown in the figure to concretely understand how this architecture works. Suppose a user presses the key *a* while the pointing device is at position *x, y* of window *w*. In the single-user case, the window server would send information about this event to the local window client. However, in this architecture, this event is intercepted by the local I/O relayer, which forwards it to the infrastructure module on the master computer, which in turn forwards it to the central window client on that computer. The output operation executed by the client to draw the input character is then broadcast to all the window servers via the infrastructure modules.

6.2.2 Coupling Syntactic Sugar

In the example above, we considered window events that must be processed by the application. There are several events that can be processed by the window server without requiring application intervention. We will refer to these as "syntactic sugar" events as they modify date of interest to the user interface but not the program component. Figure 19 shows an example of such an event: the move window event generated in response to the user moving a window.

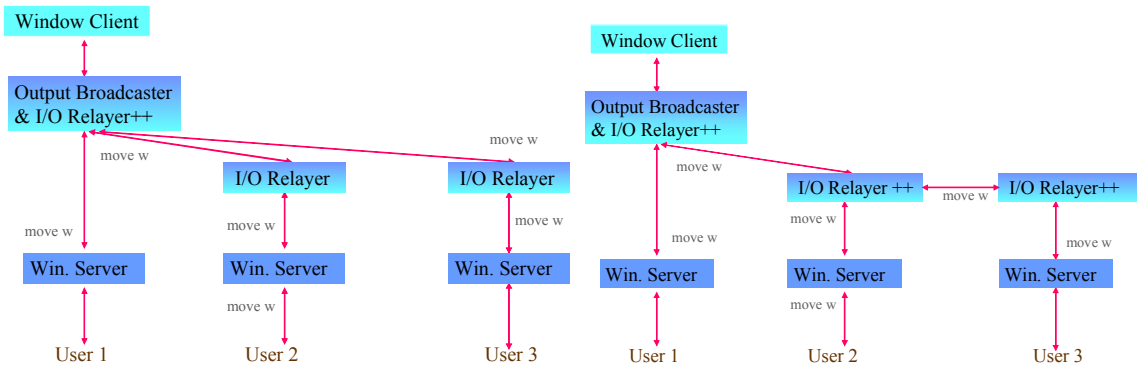


Figure 19 Centralized (a) and Distributed (b) Bradcoast of Syntactic Sugar Events

In Figure 19(a) a syntactic sugar event is broadcast to other window servers like a regular event, through the central computer, except that the event is not delivered to the central client. Figure 19(b) is a more efficient broadcast mechanism, where the event is broadcast directly to the other window servers without going through the central host, thereby giving the feedthrough of a replicated system. The infrastructure module in this approach, must now maintain links to all of the other sites, as shown in the figure. Thus we see in Figure 19(b) another way of combining aspects of the centralized and replicated architectures.

6.2.3 Broadcasting Peer vs. Input Events.

We see also in it an alternative to the mechanism presented in Figure 9 for keeping replicated data structures consistent in some layer at level S. In Figure 9, layer S-1 broadcasts an input event to all instances of the layer above. In Figure 19(b) layer S directly broadcasts synchronizing events to its peers. Figure 20 compares these approaches side by side. The peer communication approach is more flexible since the higher-level semantics in a layer can be used to ensure that some but not all data structures in it are synchronized. In the window case, the peer communication approach can be used to ensure that some but not all windows are synchronized, whereas the input broadcast approach would replicate the entire window layer. However, it is not always possible for an infrastructure to determine and intercept actions in a layer that need to be relayed to the peers. When we look at broadcast methods, we will how programmers can specify this information to the infrastructure.

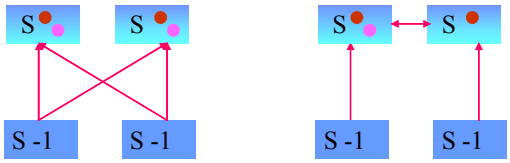


Figure 20 Input Broadcast vs. Peer Broadcast

6.2.4 Virtual Desktop

Let us consider now a semantic issue raised by Figure 19 – should a replicated window be positioned at the exact same position on all displays? Experience with such sharing shows that it leads to “window wars” as different users move it to different positions based on the non-shared windows on their displays. It is possible to not couple window positions, as shown in Figure 21, where the shared calculator window is at two different positions on the two screens. However this approach provides less referential transparency as, for instance, a user cannot refer to the “right window”.

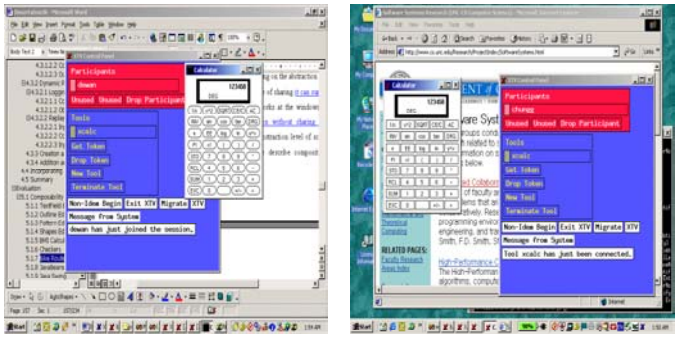


Figure 21 Minimal Window Coupling in XTV

A compromise approach, supported in T 120, is to position shared windows consistently in a virtual desktop. The virtual desktop is itself a window that can be moved position, iconified, and resized. This is the approach implemented by several commercial centralized shared window systems such as NetMeeting, PlaceWare, and Webex. In these systems, the virtual desktop window is a virtualization of the physical desktop of the computer hosting the centralized applications. The user of this computer, thus, does not see the virtual desktop. Figure 22 illustrates this approach. In this figure the physical desktop of the user on the left is displayed as a virtual desktop window on the screen of the user on the right, which contains only the two shared windows on the physical desktop.

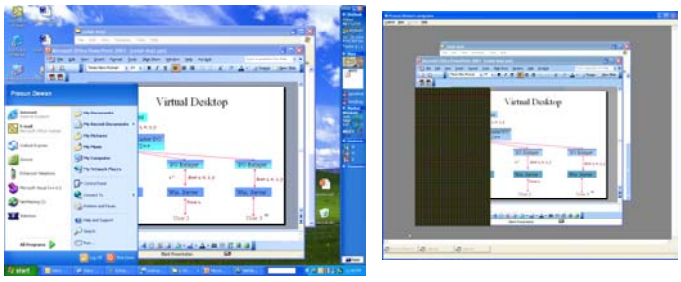


Figure 22 Maximal Window Sharing with Virtual Desktop in NetMeeting

6.2.5 Translating I/O

In a centralized system supporting a virtual desktop, I/O flow is not quite the one shown in Figure 18. The reason is that a window input event on a slave computer has two positions, one in its physical desktop and another in the virtual desktop. Similarly, the physical position of an output request from the host computer is different on each of the computers and depends on the physical positions of the virtual desktops on the machines. Therefore, in the T 120 framework, the master infrastructure module is sent the raw input event, which then uses the virtual desktop data structure (including the pointing device position) to translate it into the coordinates of its physical desktop, as shown in Figure 23 (a). Moreover, the master module translates the physical coordinates and window identifier of a drawing request it receives from the central application to virtual coordinates. Each slave module adds to them the identifier of the physical window displaying the virtual desktop before giving them to the local window server, as shown in Figure 23 (a). Thus, the infrastructure modules are not just I/O distributors - they are also interpreters of virtual desktops.

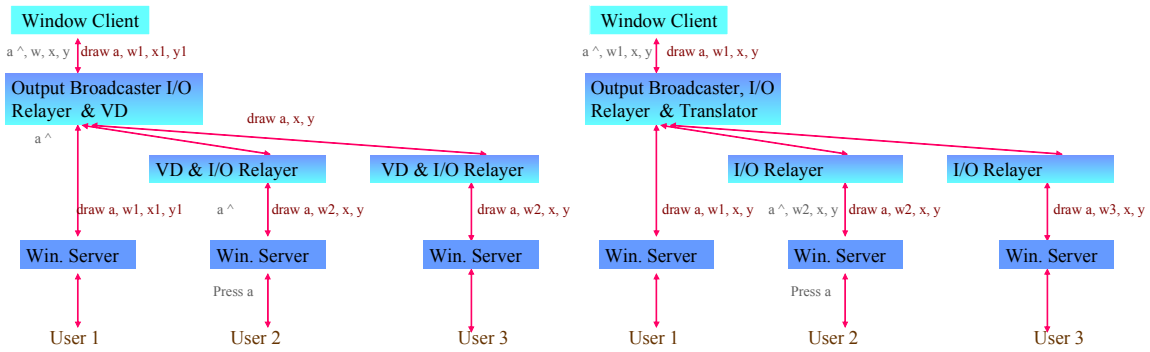


Figure 23 Basic I/O Distribution (a) with and (b) without Virtual Desktop

Figure 18 also does not describe the I/O distribution in systems such as XTV that do not support virtual desktops. The reason is that a shared application window has different identifiers in different window servers. Therefore, in XTV, the master infrastructure modules translates between the identifier on the master computer, which is the one the application uses, and those on the slave computers (Abdel-Wahab and Jeffay 1994), as illustrated in Figure 22 (b). The reason why the architecture of Figure 18 and the more abstract, layer-independent architecture of Figure 8 does not work is that it assumes I/O is host independent, which is not the case for today's window systems.

6.2.6 Coupled vs. Uncoupled Expose Regions

Consider now another semantic issue, raised by Figure 22. When the master computer occludes a shared window with a private window (left screen), that area is blacked out on the slave computers. While this ensures that the other users cannot see what the master user is not seeing, it is not clear it is desirable (James Begole, Mary Beth Rosson et al. 1999), as the example in the figure demonstrates. Therefore, other systems such as XTV do not black out occluded regions.

Figure 24 (a) shows how coupling of occluded regions is supported in T120 (which is implemented by NetMeeting). Occluding a window causes the window server to send to the application an "expose" event describing the region that is not covered by other windows. The T120 protocol requires that this information be sent to all other computers, which are then responsible for blacking out covered regions.

Figure 24 (b) shows how the alternate approach is implemented in XTV. The expose event is sent only to the application and not to other computers.

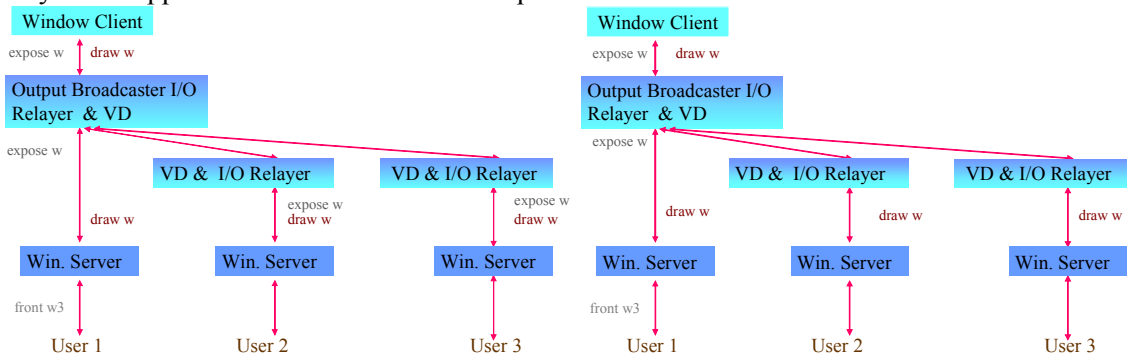


Figure 24 Coupled (a) vs. Uncoupled (b) Expose Events

Thus, more work is needed (broadcasting of expose events) to support coupled expose regions than uncoupled expose regions – which is contrary to the impression that the former is an inherent limitation of window sharing. However, the implementation of the latter described above does have the problem that if the regions covered on the host computer change, they will not be shown accurately to the other computers if the application draws only to the exposed regions of a window. It should be possible to overcome this problem if the master I/O distributor sends the union of exposed regions at all sites to the application, in which case, of course, more work would be needed to implement uncoupled regions.

6.2.7 Degrees of Window Coupling

As the discussion above implies, different window systems can support sharing of different properties of a window. A window system must support sharing of all window information that is processed by an application, which consists of the:

- window sizes
- window contents.

The sharing of the following "syntactic sugar" properties is optional:

- window positions
- stacking order of windows
- exposed regions

Both sets of properties could be shared with or without a virtual desktop. The T 120 protocol provides mechanisms to share all of these properties, and systems based on it such as NetMeeting (Figure 22) and Webex do indeed share them all. Some systems not based on T120, such as PlaceWare also provide maximal window sharing. XTV (Figure 21) is an example of system shares none of the optional properties.

6.2.8 Replicated Window Architecture

So far, we have considered only centralized window sharing architectures. The replicated architecture follows the general model given in Figure 9 modulo the issues raised here such as event translation, virtual translation, and degrees of window coupling. Figure 25 (a) shows how the basic architecture of Figure 9 may be adapted to support event translation in a system that does not support virtual desktops. Figure 25 (b) shows how the channels available for distributing input events can be used to distribute syntactic sugar events – the only difference is that the latter are not delivered to the program.

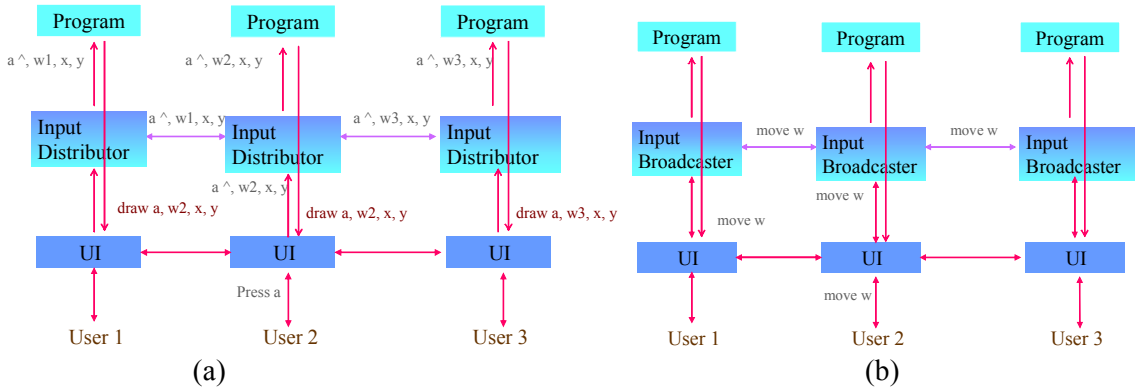


Figure 25 Translating input events (a) and distributing syntactic sugar events (b).

None of the current commercial systems support replicated window systems even though these systems are more efficient in many scenarios, as mentioned above. Moreover, as we will see later, it is easier to overcome firewall problems with this architecture as less data are exchanged among the distributed sites, which can be done over, for example, the channels used for sending instant messages. One reason is the replication problems mentioned earlier, which are harder to solve in a lower layer. However, experience with these systems has shown that these problems can be addressed for typical applications as long as floor control is assumed (Chung and Dewan. 2001; Chung and Dewan October 1996), which is the only concurrency control mechanism that can be used in a shared window system. *Thus, replicated window systems is a potential new commercial opportunity, something Senthil calls intelligent application sharing.*

6.3 Screen Sharing

As mentioned earlier, unlike the window layer, the screen layer treats the display as an un-partitioned region. In modern bitmapped displays, this is the layer that manages the framebuffer. Like window sharing, screen sharing is a popular infrastructure for collaboration. Here, instead of sharing the window client, we share the screen client, which typically is the window system and all of the applications that run on top of it – in essence the complete computer state. Thus, it is not possible, as in a window system, to share the windows of a subset of the applications that run on a computer. This is consistent with the general trend we pointed out earlier of lower level layers providing a coarser granularity of sharing. Screen sharing is the lowest level of sharing possible and thus provides the coarsest sharing granularity.

6.3.1 No Replication in Screen Sharing

Sharing the complete computer state in the replicated architecture means all computers are in the same state. For example, it implies that all icons are at the same exact (physical or virtual) position on each display. This can be ensured only if either:

- all computers are identical and from the point they are bought receive the same sequence of input, which rules out the use of the computers for activities other than collaboration.
- or, when a new user joins the conference, the entire computer state is downloaded from one of the existing conferees, which can take a very long time.

It may be possible to create creative solutions to address these problems. For example, one could create dedicated computers bought solely for the purpose of collaboration with a particular group of users to address the problem with the first alternative. However, to date, there has been no attempt to do so in the commercial or research world. Therefore, we will focus only on the centralized architecture for screen sharing.

6.3.2 Centralized Screen Sharing

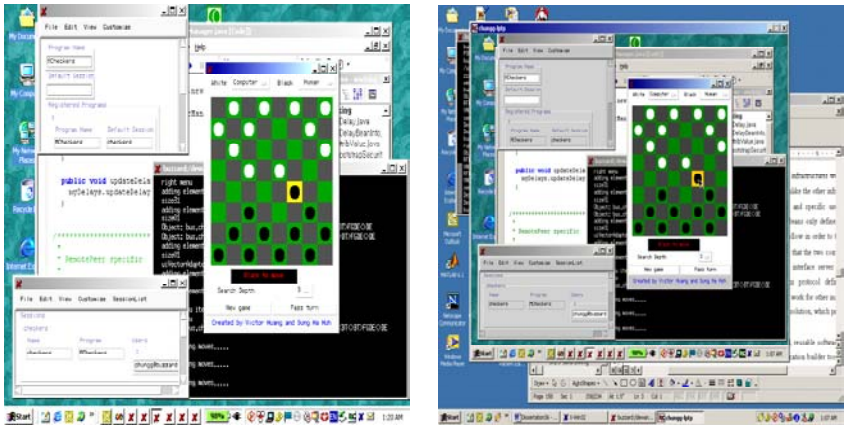


Figure 26 Centralized Sharing of Icons, Taskbars and other Screen Contents in VNC

As shown in the Figure 26 image of a VNC shared session, in centralized screen sharing systems, on all of the slave sites, a virtual desktop of the master computer is displayed. It is much like the virtual desktop we saw in the case of window sharing (Figure 22). The main difference is that icons, taskbars and all application windows of the master computer are displayed in the virtual desktop on the right.

Figure 27 illustrates how screen sharing is implemented using the example of VNC. When an application on the master computer updates the display, each slave computer receives pixmap rectangles describing the regions updated – which are essentially diffs between the screen contents before and after the operation. A pixmap rectangle consists of a matrix of pixels describing its contents and (x,y) coordinates describing its location. Conversely, when a slave user inputs a command, the key and mouse events comprising the command are relayed to the master computer.

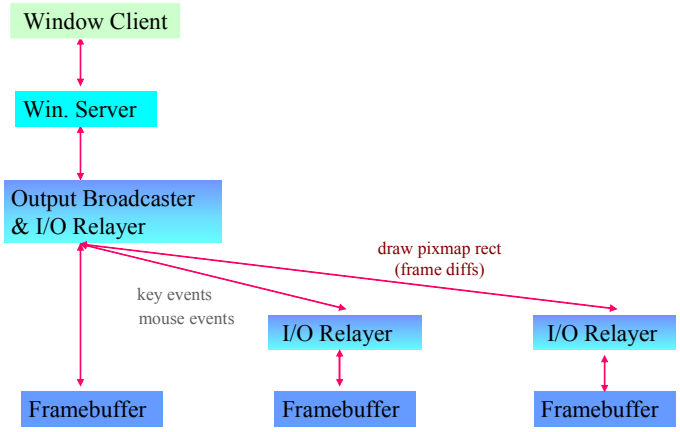


Figure 27 Centralized Screen Sharing in VNC

6.4 Sharing pixmaps vs. drawing operations

We have, thus, seen above two different ways for sharing screen updates – sharing drawing operations and sharing pixmaps. The window sharing systems we considered above use the former approach while VNC and other screen sharing systems use the latter. A screen sharing system must share pixmaps since it does not know about window operations. A window sharing

system, on the other hand, could share drawing operations or the pixmaps it generates to implement them.

6.4.1 Relative Advantages

It may seem that one should share drawing operations when possible since less data would be transmitted. Moreover, in a shared window system, it easier to obtain the data to be distributed - all we have to do introduce a proxy between the window client and server that intercepts the drawing requests. Obtaining the pixmap operations is more difficult as we must either

- do framebuffer diffs, or
- put hooks into the window server that retrieve the pixmaps it creates for the drawing operations, or
- do our own translation from window operations to pixmap changes.

On the other hand, there are several advantages of sharing pixmaps. To share pixmaps a single output operation – draw pixmap - is needed, whereas the number of drawing operations that need to be shared tends to be large. As a result, the I/O distributors must be modified as new operations are added. The single pixmap operation is also a standard one, while different window systems tend to support different sets of drawing operations. Thus, interoperability is more difficult in the drawing case.

Moreover, pixmap operations are stateless while drawing operations must be interpreted in the context of fonts, colormaps, and other state, which must, therefore, also be shared. This has at least three consequences:

- As in other stateless systems such as file systems, we may be able to use a more efficient transport protocol such as UDP to share pixmaps, whereas we cannot afford to loose a single drawing operation or execute it multiple times. It would be interesting to apply the work in stateless file systems to study the use of a more efficient transport protocol.
- Joining a new conference takes longer when drawing operations are shared because the state on which they depend must be distributed before sending them.
- The operations that distribute the state are also non-standard, which increases the complexity of the sharing infrastructure and decreases its interoperability.

Furthermore, multiple pixmap operations can be coalesced into one operation whereas coalescing of drawing operations is not possible. Coalescing increases feedthrough time and reduces the fidelity of interaction as each incremental change is not observed, but has the advantage that a computer with a slow computer or network connection can participate in the collaboration.

Finally, though drawing operations contain less data than the pixmaps they generate, the latter can be compressed, as we see later. Drawing operations can also be compressed, as we will see later.

6.4.2 Mixed Model

As T 120 shows, it is possible to combine the two sharing models by allowing communication of drawing operations or the corresponding (potentially coalesced) pixmaps– the choice is a policy matter depends on static and dynamic conditions. In particular, pixmaps are sent when:

- the receivers do not support the corresponding drawing operations
- the receivers are too computationally overloaded to process each of the corresponding drawing operations. In this case, these operations are coalesced into a smaller number of pixmaps.

- the network links to the receivers are congested and it is more efficient to send coalesced pixmaps than the original drawing operations.

The mixed model has the advantage that it reduces to the feedthrough and fidelity of pixmap model only when required. The disadvantage is that the infrastructure must provide the mechanisms for converting between drawing operations and pixmaps, and the policies that use these mechanisms.

6.4.3 Pixmap Compression

T 120 and VNC identify several ways in which pixmaps can be compressed:

- Combine updates to overlapping region into one update (T 120). In a framebuffer diffing system, this is automatically ensured. In other systems, an additional process must be used to compute the overlaps.
- When a pixmap contents already exists at the receiving site, send the coordinates instead of the contents (T120, VNC). This optimization is particularly useful when windows are moved or scrolled. The pixmaps contents may retrieved from the hardware framebuffer, if it is readable, or a separate cache.
- Send diffs with respect to previous rows of the pixmap (T 120).
- When a pixmap subrectangle consists of a single color, send the color instead of the contents of the entire subrectangle contents (VNC). Several drawing applications create a display rectangle by specifying a default background color and then use drawing operations to overwrite this color in parts of the rectangle. This optimization would allow the undrawn portions of the rectangle to be sent efficiently.

These optimizations increase the complexity and number of the operations communicated between the host and slave sites, thereby potentially conflicting with interoperability. Thus, they can be supported only if all participating sites understand them. They also reduce the statelessness of pixmap sharing, thereby requiring a more complex transport protocol. Finally, they increase the computation load on the communicating sites, which must compress and decompress the upated. We see later a comparison of the bandwidth gains with the computation load overhead in experiments.

6.4.4 Compressing Drawing Operations

T 120 also identifies a way in which drawing operations can be compressed. When an operand is repeated in a sequence of (successive) drawing requests, omit it in all but the first element of the sequence. This optimization may be useful for large repeated operands such as graphics contexts.

Compression of pixmap and drawing operations can be counterproductive in large bandwidth environments because they can unduly reduce feedthrough times because of the extra computation overhead, as we will see in the experiments reported later.

6.4.5 Pointer Coalescing

To further reduce the network traffic, T 120 supports the coalescing of multiple updates to the pointing device position, made by the user or the application, into a single update. This optimization decreases the fidelity of interaction as the user does not see each incremental change. Moreover, as we see later, experiments show that pointer updates produce very little network traffic. The main advantage of this optimization is that in a high latency environment, it reduces the jitter in the environment as fewer events are communicated (Kum and Dewan 2000). Whiteboards in PlaceWare, Webex, and NetMeeting implement a sharing policy that is consistent with pointer coalescing – instead of sending incremental movements and stretching of a graphical object, they send one update at the end of the movement or resize operation.

6.5 Flow Control Algorithms

We have seen above mechanisms to adapt the number and size of messages required to communicate I/O operations. Let us consider two dual policies for using these mechanisms.

6.5.1 Push-based Model

In T 120, these operations are pushed to the receiving sites. The end to end delay for communicating data is determined by looking at the application queue sizes at the receiving sites. The sender adapts data transmission based on the queue size of the slowest receiver. If this queue size is smaller than some threshold, then more messages and data can be sent, and if the size is larger than a greater threshold than fewer messages and data should be sent (using the various techniques outlined above). If the queue size is larger than an even greater threshold, then the slowest computer leaves the collaboration.

6.5.2 Pull-based Model

In VNC, each receiver pulls I/O operations from the site generating it. On each pull, it gets all changes to the framebuffer made since its previous pull. As a result, different receivers can operate at different rates – a slow receiver does not impact the performance of other conference participants. On the other hand, the infrastructure must maintain per-receiver state. Moreover, if multiple receivers share a network link to the sender, the same update is sent multiple times on that link. In the T 120 model, a single multicast message can be sent on that link.

6.5.3 Research Opportunity for Flow Control

It would be useful to explore an approach that caches arbitrary objects. In particular, it would be useful to replicate screen updates on each LAN and have each member of the LAN contact the local replica instead of the sending machine. This approach essentially requires a replicated architecture in which infrastructure components maintain application state. This idea has been explored for efficient latecomer accommodation in X applications (Chung, Dewan et al. 1998) and for dynamic architecture transitions of arbitrary applications (Chung and Dewan. 2001). In these systems, infrastructure components execute on user sites. The author's group has developed a simple extension of (Chung and Dewan. 2001) in which potentially replicated infrastructure components can maintain the state of remote, possibly replicated applications.

6.6 Performance of Distribution of Drawing and Pixmap Operations

In practice, what is the performance of the pull-based and push-based flow control algorithms, sharing of pixmap and drawing operations, and replicated and centralized window sharing architectures? These and other performance questions are partially answered by some of the experiments described here.

6.6.1 Remote access to window server

Nieh et al (Nieh, Yang et al. 2000) have conducted several experiments to measure the performance of remote window access in a single-user environment. Figure 28 shows the architecture used in these experiments. They compared different systems implementing this architecture. These systems include RDP (Win2K implementation) and VNC (Linux implementation). RDP is a Microsoft product. Based on these and other experiments (Wong and Seltzer 2000), it is believed to, like T 120, distribute both drawing and pixmap operations and

perform compression on pixmap operations. As mentioned earlier, VNC distributes pixmaps, does pixmap compression, and implements pull-based flow control.

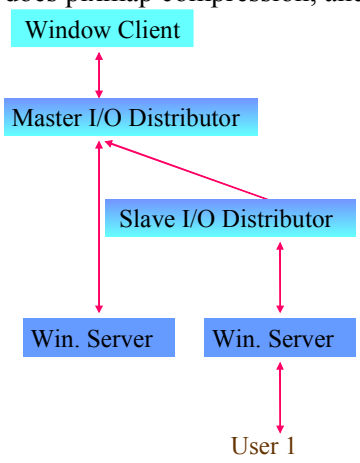


Figure 28 Remote Access Architecture

The architecture used in these experiments is essentially a two-person centralized window sharing architecture except that the master site does not have an active user. Assuming that the central window client does not become a bottleneck when multiple users interact with it, these experiments give us an idea of the performance seen by remote users of a centralized window system. In particular, they give us an idea of the relative performance of (a) distributing pixmap vs. drawing operations, (b) push-based vs. no flow control, and (c) VNC compression vs. RDP compression.

The experiments measured the feedback times and data transferred for various user actions including:

- inputting the letter "a"
- inputting a command that causes a red box to get filled.
- loading web pages that share images.
- viewing animations.

They found that when the network bandwidth is abundant (100 Mbps):

- the remote feedback time for
  - letter "a" is 60 ms in VNC and 200 ms in RDP.
  - red box fill is 100 ms in VNC and 220 ms in RDP.
- the data transferred for
  - letter "a" is 0.4 KB in VNC and 0.3KM in RDP
  - red box fill is 1.2KB in VNC and 0.5KB in RDP

As we see here, when the data transferred is less, the remote feedback time is worse. Their explanation is that the more effective compression increases the complexity of I/O distribution thereby increasing the remote feedback time. As we see from the data, the feedback time becomes greater than 100 ms, which is noticeable by humans. Their conclusion thus is that compression is counter productive when the network bandwidth is abundant.

In their web page experiments, they twice loaded 54 text and graphics web pages that shared common images (including a blue left column, white background, and PC magazine logo) to

allow for compression. After each page was loaded, it was scrolled down 50 pixels before the request for the next page was sent. They found that when the bandwidth is:

- between 4-100 Mbps, the load time in both systems is less than 50 seconds, which they considered tolerable.
- 100 Mbps
    - the load time is 24 seconds in VNC and 35 seconds in RDP.
    - the data transferred is 4MB in VNC and 12 MB in RDP (8 MB of data was sent to the web browser client on the master site)
- 128 Kbps
    - The load time is 24 seconds in VNC and 297 seconds in RDP.
    - The data transferred is 1 MB in VNC and 12 MB in RDP

The high bandwidth experiments show pixmap distribution (with compression) can compare well with model distribution – in the VNC case, converting the web pages to pixmaps increases the data size by only a factor of two. Moreover, it shows that pixmap distribution in VNC is at least as efficient as I/O distribution in RDP – since RDP distributed both drawing and pixmap operations, it is not possible to compare distribution of pixmap and drawing operations via this experiment.

The low-bandwidth experiments are more interesting. They show that as the available network bandwidth reduces, the load time in VNC remains constant while in RDP it degenerates to 297 seconds – six times the tolerable limit. On the other hand the data transferred in VNC becomes one fourth of what it was in the high bandwidth experiments while in RDP it remains constant. This shows the pull-based flow control of VNC at work – at lower bandwidths, the remote site pulls screen images at a lower rate, thereby losing all changes made since its last pull. Thus, neither system works well for a user having a low bandwidth connection to the application. The VNC approach does have the advantage that users with high bandwidth connections are not affected by the slow user, or put another way, the slow user can continue to participate in the collaboration, even though in an unsatisfactory way. However, it is not clear from this experiment that the VNC approach of allowing slow users to remain in the conference is better than the T 120 approach of ejecting them.

The animation experiments yielded similar results. In these experiments they measured the frame per second (fps) and data transferred at the remote site. A frame rate of 18 fps was considered good by users and 8fps tolerable. The animation clip used vector graphics. They found that when the bandwidth is:

- 100 Mbps
    - The frame rate is 18 fps in VNC and RDP
    - The data transferred is 2.5MB in VNC and 3MB in RDP.
- 512 Kbps
    - The frame rate is 15 fps in VNC and 8fps in RDP
    - The data transferred is 1.2MB in VNC and 2.2M in RDP
- 128 Kbps
    - The frame rate is 16 fps in VNC and 2 fps in RDP.
    - The data transferred is 0.3 MB in VNC and 2.2MB in RDP.

The experiments at 512 Kbps bandwidth again question the usefulness of the pull-based approach. RDP transfers the entire data at a tolerable frame rate whereas VNC loses more than half the data. (As these experiments show, RDP is also capable of losing some data.)

6.6.2 Bandwidth usage of pixmap and drawing operations

Animation experiments were also conducted by Wong and Seltzer (Wong and Seltzer 2000), which show that distribution of pixmaps can be more efficient than that of drawing operations. In experiments with RDP (the WinNT implementation), they found that the load on the network was:

1. 0.01 Mbps when animating a 468x60 pixel GIF banner.
2. 0.01 Mbps when animating a scrolling news ticker.
3. 1.60 Mbps when animating the banner and scrolling news ticker simultaneously.
4. 0.1 MBps when running the Bezier curve screen saver animation.

As can be seen from above, running the first two animations individually consumes (a) two orders of magnitude less bandwidth than running them concurrently, and (b) an order of magnitude less bandwidth than running the Bezier curve animation. Wong and Seltzer attribute this to a pixmap cache at the remote site. The first two animations are cyclic and once the whole cycle is cached, the master site need only transmit identifiers of the cached pixmaps. They conjecture that the cache is big enough for each of the two animations but not for both of them, each of which purges the data of the other using an LRU policy (A most recently used cache purging policy would perform better here!). Hence the concurrent animation takes far longer. Even though the Bezier curve animation is also cyclic, they conjecture it must distribute drawing operations, since it takes so much longer than the first two animations. Thus, in a cyclic animation, pixmap cache and compression strategies that make use of it can make pixmap distribution more efficient than distribution of drawing operations. On the other hand, assuming the concurrent animation is as complex as the Bezier animation, without a pixmap cache, distribution of drawing operations is more efficient than that of pixmaps without cache-based compression.

Wong and Seltzer also point out that experiment 3 above shows that the bandwidth consumption of animations can be substantial – in a 10MBps network, only 5 users can simultaneously run the animation; in a 100 Mbps network, only 50 users can be accommodated. As we see later, this has an important implication for collaboration that occurs among rooms of people – distribution of input and output should be decoupled in such collaborations.

Of course, not all interactions are bandwidth intensive as the other animation experiments show. Wong and Seltzer looked at the network load of a variety of interactions and found that it was:

- 6.2 Kbps when typing at 75 words per minute,
- 2 Kbps when randomly mousing. This number is low because cursor shapes are cached on the remote site. This low number shows that input filtering in T 120 does not have significant impact on network load.
- 1.17 Kbps when doing a depth-first selection from the Windows Start Menu.
- 39.82 Kbps when moving among Word toolbar menus.
- 48.82 Kbps when using Office 97 menu animations.
- 60 Kbps when continuously scrolling a Word document.

Thus, the network load of different interactive operations varies substantially, ranging from 2 KBps to 1600 Kbps in the experiments above.

6.6.3 Relative occurrence of Window Operations

The actual network load of an application depends on the relative occurrence of various operations. Danshkin and Hanrahan (Danskin and Hanrahan 94) performed studies to determine this factor in an X environment. They looked at usage of two 2D drawing programs, a postscript previewer, and an X 11 performance meter by 5 graduate students doing their daily work. The following is a list of operation classes sorted by how many bytes were transmitted on the network to execute them:

1. Image – many postscript previewers draw text using images.
2. Geometry – half of the geometry operations involved clearing rectangles to erase characters.
3. Text.
4. Window enter and leave.
5. Miscellaneous operations including mouse and window movement and font operations.

They also found that most I/O packets were small (less than 100 bytes) and that the TCP/IP header added a substantial overhead to them. Even the image packets were small – on an average 53 bytes in size consisting mostly of black and white rectangles encoding text. Finally, they found that creating the initial user interface generated a lot of network traffic and thus took a long time – on the average 20 seconds. This confirms the observation in (Chung and Dewan October 1996) and motivates starting application replicas before a collaborative session.

6.6.4 Network and Computation Load of Different User Classes

Of course, graduate students are not typical information workers targeted by industry. In another study involving Microsoft's Terminal Services (NEC 2000), three classes of workers accessing remote applications were studied:

- Knowledge workers such as marketing managers, who make their own work. They mostly use spreadsheets, mail, web browsers, and word processors, which they keep open all the time.
- Structured task workers such as claims processing personnel, who are given work by others. They mostly use mail and word, but use each application for a smaller time, closing and opening them frequently.
- Data entry workers such as typists and order entry personnel. They mostly use SQL and forms.

The study found that the average network load put by the:

- Knowledge worker was 1200 bps.
- Structured task worker was 1950 bps.
- Data entry worker was 495 bps.

It also determined how many workers of each class can be supported before there was 10 % degradation in the response of the master distributor running on a 450 MHz machine. They found the numbers to be:

- 70 knowledge workers
- 40 structured task workers.
- 320 data entry workers.

Thus, this study confirms the finding of the earlier study that on average remote access does not put substantial load on the system.

6.6.5 Regular vs. Bursty Traffic

Droms and Dyksen (Droms and Dyksen 1990) looked not only at the average but also the bursty network traffic when a systems programmer accessed remote X applications for 8 hours. They found that the average network load was 236 bps. However, there were many bursts greater than 40, 000 bps. In fact, the X Tex previewer (Xdvi) generated bursts greater than 60, 000 bps and an X window manager (Twm) and X screen capture program (Xwd) created bursts greater than 100, 000 bps. This load is comparable to the load put by some animations described above and can be an issue in centralized window or screen sharing.

6.6.6 Replicated vs. Centralized Network Load

As mentioned earlier, the replicated architecture can be expected to put less network load than the centralized architecture because each input event is expected to generate multiple output events. On the other hand, several input events such as mouse events can be discarded by an application, and output can be buffered. Ahuja et al (Ahuja, Ensor et al. 1990) performed experiments to compare the two architectures when using a couple of drawing programs. They found that:

- When output was not buffered, there were 6 times as many output events as input events.
- When output was buffered, there were 3.6 times as many output events as input events.
- An output event was about the same size as an input event – about 25 bytes (not including network headers presumably).

As indicated by the experiments above, there are several other kinds of applications such as postscript previewers and animation programs that can generate much larger output data. Thus the replicated architecture can indeed be expected to generate much less traffic than the centralized architecture.

6.7 Widget Sharing

So far, in this section, we have focused on screen and window sharing – perhaps the most popular forms of sharing today. Let us now consider sharing of higher-level layers.

In most systems, the layer immediately above the window layer is the toolkit layer, offering the abstractions of "widgets" – the name given for text fields, scrollbars, sliders, and other toolkit objects. Figure 29 shows the replicated architecture for sharing these abstractions. Instead of sharing device-level input events such as key presses and releases, this architecture shares higher-level input events such as text operations.

There are several implementations of the replicated architecture for the toolkit layer including GroupKit, JCE, and Habanero. However, there in only one system known to that supports the centralized architecture for it (Chung and Dewan. 2001). A reason for the unpopularity of the centralized architecture is that unlike the window layer, in all systems known to us, the toolkit layer is not a network layer, that is, its client cannot be on a separate host. As a result, it is harder to intercept the I/O operations that flow between a toolkit and the layer above. Fortunately, most toolkits announce input operations to any interested module through a publish/subscribe mechanism, which can be used to implement the replicated architecture. However, an equivalent mechanism does not exist for output operations. In (Chung and Dewan. 2001), a special debugging capability provided by the Swing toolkit was used to intercept toolkit calls made by the toolkit client.

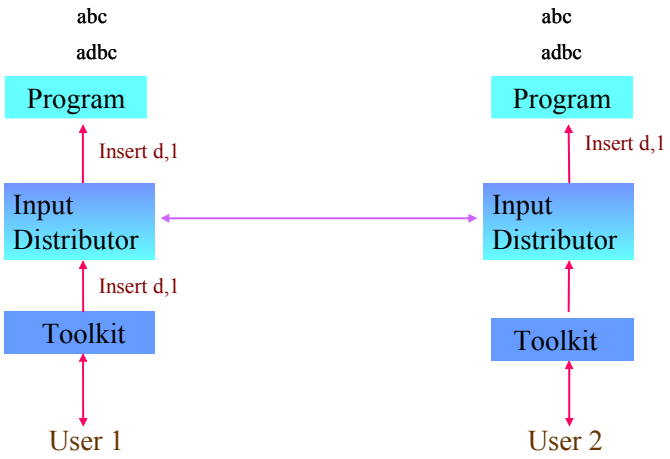


Figure 29 Sharing String Operations in a Replicated Toolkit

Figure 30 illustrates the user interface of toolkit sharing using the example of the Swing toolkit and the sharing support provided in (Chung and Dewan. 2001). The screens of the two users are implemented on platforms using different implementations of the toolkit – as a result the "look and feel" of the windows is different. For example, the slider on the left is triangular and the one on the right is rectangular. Moreover, the sizes of the windows are different, as it is widgets in the windows and not the whole windows themselves that are coupled.

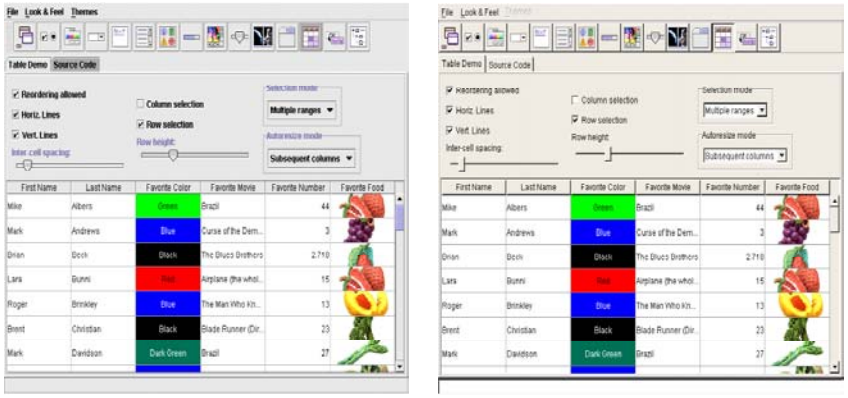


Figure 30 User Interface for Shared Swing

In the above example, user windows differ only in their look and feel. It is also possible to provide more divergence among the windows. For example, users can independently scroll a shared text widget, and if the widget is collaboration-aware, it could provide a special "multiuser scrollbar" showing the positions of the scrollbars of all of the other users, as shown in Figure 31.

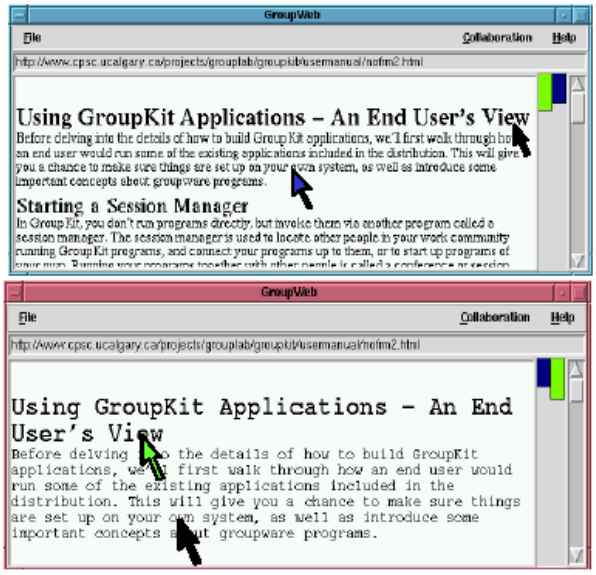


Figure 31 Multiuser scrollbar and semantic telepointer in GroupKit

The figure also illustrated a subtle issue with window divergence – telepointers cannot be synchronized by coupling their positions within the window. Instead, they must be synchronized by ensuring they point at the same contents.

While several research implementations of a shared toolkit exist and have been used, there is no commercial one (at least widely) available today. Such an implementation would be an intermediate point between window and model sharing. Like model sharing it can allow

- independent window sizing and scrolling
- concurrent editing of different widgets in a window
- and merging of concurrent changes to a replicated text widget.

Like window sharing, it:

- does not require changes to the application programming model,
- can provide sharing of existing applications.

Some of the research toolkits such as Habanero and GroupKit have been fairly successful. *It would be useful to explore, for instance, the development of a similar toolkit on top of Microsoft's User or Avalon toolkits.*

6.8 Sharing the Model Layer

Toolkit sharing allows only for limited window divergence. It cannot, for instance, allow one user to see a pie chart view and another to see the bar chart view of some data (Figure 32). In general it cannot support collaborations in which the same model object must be bound to different widgets, as this binding task is not done by a toolkit. The objects that do this task are referred to as the view and controller in the MVC terminology and we will refer to them collectively as the view (Figure 33).

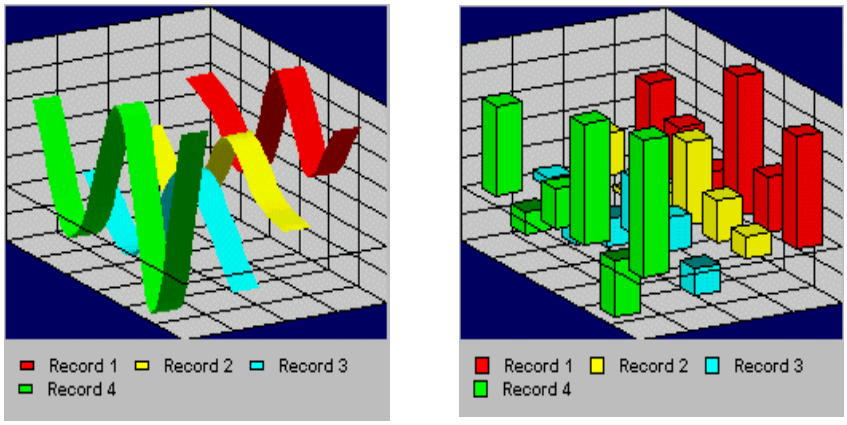


Figure 32 Multiple Views of a Shared Model

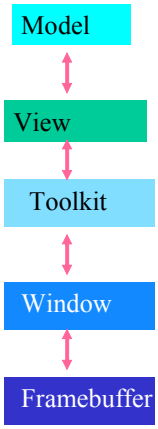


Figure 33 Views: Binding Models to Widgets

The maximum divergence between the user interfaces displaying a shared model is to share only the model layer, allowing the view and the layers below it to be replicated. Unfortunately, sharing the model layer is more of an issue than the other layers because there is no standard model-view protocol. The MVC architecture leaves undefined the elements of this communication. Without such a protocol, it is not possible to build modules that automatically distribute I/O.

Several approaches have been used to address this problem:

- Provide communication facilities of varying abstractions for manual implementation of the I/O distribution task.
- Define a general I/O model for model view communication and hope it gains widespread use.
- Provide special types that implement mechanisms for replicating their instances.
- Mix the above these abstractions to various degrees.

We discuss below these approaches in depth.

6.8.1 Unstructured Channel-based Communication

This approach is used in T 120 to support sharing of the model layer. Modules at distributed sites can send unstructured byte streams to each other. Facilities are also provided to define (hierarchy of) multicast channels. The multicast facility is integrated with the session management facility to automatically define multicast channels addressing the nodes participating in a session. These abstractions can be used by programmers of collaborative infrastructures and applications to

implement I/O relayers and broadcasters. In particular, the T 120 window sharing infrastructure and the T 120 Whiteboard use these abstractions.

6.8.2 Remote Procedure Call

An alternative to the above approach is to support communication of structured types provided by some programming language. Such communication is supported by the abstraction of a remote procedure call, which may be synchronous or asynchronous. Ideally remote and local procedure call should have identical syntax and semantics, but in practice this does not happen. Many collaboration infrastructures and applications have been built on top of Java's implementation of RPC – RMI.

The basic RPC abstraction has also been integrated with session management to support multicast to subsets of session participants [Greenberg and Marwood, Dewan & Choudhary '92], more specifically, subsets of processes registered with the infrastructure that run at the sites participating in the session. These include:

- (registered) processes of all users in the session
- processes of users other than the caller
- processes of specific user
- specific process
- processes of user-defined group of users.

It would be interesting to explore adding all of these extensions to Indigo, the standard layer being developed at Microsoft to support structured communication.

6.8.3 General Model-View Communication Protocol

As mentioned above, another approach is to define a standard model-view communication model and then provide facilities to automatically distribute the I/O defined by this protocol to support collaboration. This protocol can be defined in terms of objects defined by the model or the view. View-defined objects such as pie charts can be arbitrary – limited only by the imagination of user-interface designers. Model objects on the other hand tend to be defined by standard programming language types, at least in a non object-based programming language such as C.

Therefore, the following I/O model has been defined in (Dewan October 1990) for C:

- Output: Model sends copies of its elements that are to be displayed by the view as well as updates to them.
- Input: View sends updates to displayed model elements made by the user.

This model has been extended in (Dewan and Choudhary November 1991) to support a centralized collaboration architecture in which:

- Output Broadcast: Output messages sent by the model are broadcast to all views.
- Input relay: Multiple views are allowed to send input messages to the model.
- Input coupling: Input messages sent to the model are also automatically broadcast to all other views. Thus, this model supports both peer and input event broadcast.

Figure 34 illustrates this architecture. As the figure also shows, the architecture essentially allows synchronization of selected model objects in the model and its views. The I/O messages download model objects in the views and keep these replicas consistent.

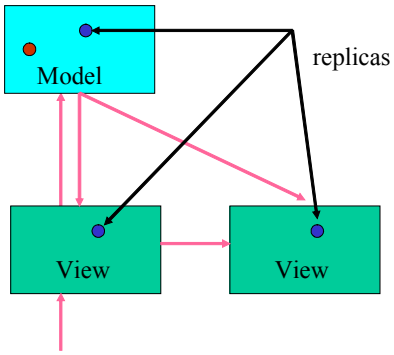


Figure 34 I/O between model and views in a centralized collaboration architecture

6.8.4 Replicated Types

It is possible to support the replication, not in an I/O model, but in the implementation of the type itself. This approach can support both the centralized (PlaceWare) and replicated (GroupKit, Groove) architectures (Figure 35). In the centralized architecture, instances of replicated types in the centralized model and its view are kept consistent, while in the replicated architecture, instances of the types in peer models are kept consistent. In the replicated architecture, replicated types or some non standard model view I/O protocol can be used to keep model and view data consistent. As shown in the figure, it is possible to synchronize a subset of objects in the model layer.

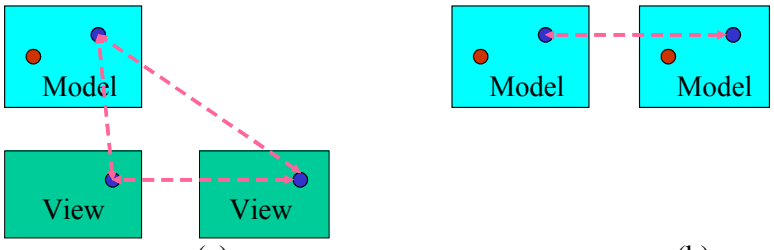


Figure 35 Using Replicated Types to Create Semi-Centralized (a) and Replicated (b) Architectures

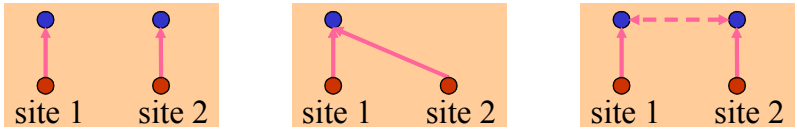
The architecture on the left is not truly central as peer views keep replicas. It is essentially the semi-centralized two-level program architecture of Figure 17(a). The model replicas in the view can be together considered a subset of the model functionality. This architecture may synchronize the view replicas directly, as in the figure above, or indirectly via the model replica as in Sync.

The replicated types can be extensions of standard non-replicated types including object-oriented types (Groove) and non object-oriented types (GroupKit). They can define a variety of objects including primitive objects such as strings and integers, records of simple objects (Sync(Munson and Dewan June 1997), Groove), dynamic sequences (Suite, Sync, Groove), hashtables (Groove, Sync), and combinations of the above objects (Sync, Groove).

6.8.5 Replication vs. Passing by Value and Reference

Replicating an object distributes it dynamically to another site. It is important to distinguish it from other, more popular forms of object distribution (Figure 36):

- Copy: When a remote procedure call is made, in several systems such as Java and .NET, it is possible to pass an object parameter by value to the remote site, where a copy of the object is sent to the remote site. Changes made to the original object are not reflected in the copy.
- Reference: An alternative to the above approach is to pass the object parameter by reference, that is, send a pointer to the object to the remote site, which can be used by the remote site to invoke methods in the local object. Thus, both sites see the same object state, but method invocation is a costly operation for the remote site.
- Replicating: Replicating the object is mix of the two approaches in that as in passing by value, a copy of the object is sent to the remote site, and in passing by reference, both sites see the same object state, as changes in one copy are replicated to the other. As we have seen before, it is possible for the two copies to diverge for a period of time. When this happens, a special merging mechanism can be invoked to make them consistent.



(a) (b) (c)

Figure 36 (a) Copy vs. (b) Reference vs. (c) Replication

Replicated types is one several approaches we have considered in this section for model sharing. Let us compare them.

6.8.6 Replication vs. Communication Mechanisms

Replicated types free programmers from having to use any kind of communication mechanism to support model sharing, thereby offering much higher automation. Thus, creating collaborative application is a much easier with replicated types.

All of the above applications were built from scratch. It is not clear replicated objects are as useful when creating collaborative applications from existing single-user ones. The reason is that non-replicated types in the program must be converted to replicated types, which can be non trivial as it may require re-partitioning of the application to separate shared from unshared models. For this reason, it is not clear, for example, that any of the Office apps can be made collaborative by using replicated types. These applications essentially share a tree structure, but extracting this structure from existing code is very difficult. For this reason, the XAF project is building an equivalent set of applications with an explicit tree structure. *It would be useful to support this tree as a replicated type.*

6.8.7 Unstructured channel vs. Other Mechanisms

Of all the communication mechanisms presented here, unstructured channels offer the least automation, requiring programmers to:

- manually serialize and deserialize objects.
- multiplex and demultiplex operation invocations over a channel

They do have the advantage that they are:

- language-independent, and thus allow a collaborative program to be composed of modules written in different languages..
- easy to learn as they are a simple abstraction and do not require the use of special compilers.

They may be particularly useful for adding collaborative features to existing single-user programs with simple data structures. The collaborative video viewing example presented earlier (Figure 10) is such a case. The authors used the T 120 primitives to multicast video command names to other replicas, and in private communication said that learning and using these primitives was trivial.

6.8.8 RPC vs. Other Mechanisms

The RPC abstraction is an intermediate point between replicated types and unstructured communication. It is higher level than unstructured communication as programmers do not have to worry about serlialization, deserialization, multiplexing, and demultiplexing. It is less flexible and harder to learn than unstructured communication because it is associated with a particular language, compiler and procedure invocation syntax and semantics.

It is lower level than replicated types as programmers must manually communicate information to shared objects. Programmer using a replicated type simply changes instances of the type without worrying about propagating the changes to others. On the other hand, it is easier to learn than replicated types because a new set of type names and associated sharing semantics do not have to be learnt. Moreover, it is more suitable for extending existing programs as existing single-user types do not have to be converted to replicated types. It is also more flexible than replicated types as programmer have explicit control over the exact remote feedback to a local command, and the architecture used to implement the application. For example, it can be used to display changes to a shared item in one (programmer-chosen) color at a remote site and another color at the local site or display the changes synchronously to one user and asynchronously to another. Similarly, it can be used to implement the centralized architecture for some applications and replicated architecture for another. The replicated type systems discussed so far do not give this flexibility.

Thus, an RPC system is useful when:

- it is available for the target application programming language.
- the overhead of using unstructured communication is high compared to the overhead of learning the RPC system.
- appropriate replicated types are not available or the overhead of learning them is high compared to the overhead of using the RPC system.

6.8.9 RPC vs. M-RPC

As we saw above, RPC comes in two flavors – regular RPC and session-based multicast RPC (M-RPC). M- RPC is an intermediate point (in ease of learning, ease of use, and flexibility) between regular RPC and replicated types. Moreover, it can be automatically mapped to an underlying multicast channel abstraction by the system. In a program that uses a point to point communication mechanism to manually distribute data, it is not possible for a system to automatically know to which sites the same data have to be sent. Thus, M-RPC should be used over RPC when possible.

6.8.10 Combining Approaches

Thus, each of the mechanisms we have seen above for model sharing has its unique advantages. Therefore, it is useful to explore how one could combine their advantages.

1. Coexistence: A simple approach for doing so is to provide a single system with multiple mechanisms. For example, GroupKit and Suite provide RPC and replicated types.
2. Migratory path: A more ambitious approach is to layer the mechanisms, providing frameworks that allow a lower-level mechanism to be used to create a higher-level one.

For instance Sync allows an RPC mechanism to be used to build a replicated type. The advantage of this approach is that the effort involved in using the lower-level mechanism is now reused each time an instance of the replicated type is instantiated. To illustrate, assume we need to replicate a set of file directories. The RPC mechanism can be used to create a replicated hashtable type, which is now available to not only support shared file directories but also for an address book and other hash tables. It would be useful to provide migratory paths not yet explored– *in particular the use of a multicast unstructured communication channel to support multicast-RPC*. A problem with this approach is that the layering frameworks can be hard to learn.

3. Different abstractions: An even more ambitious approach is to identify different abstractions that mix the advantages of the above abstractions.

6.9 Multi-Layer Sharing

6.9.1 Broadcast Methods

One example of the third approach is the abstraction of broadcast methods {, which was invented in Colab (Stefik, Foster et al. January 1987). A class can declare one or more of its methods as broadcast, as shown below using Java syntax:

```
public class Outline {
  String  getTitle();
  broadcast void setTitle(String title);
  Section getSection(int i);
  int   getSectionCount();
  broadcast void setSection(int i,Section s);
  broadcast void insertSection(int i,Section s);
  broadcast void removeSection(int i);
}
```

Figure 37 Broadcast methods

An explicit invocation of the method in one replica triggers an implicit invocation of it on all of the other replicas. Thus a broadcast method is a like a multicast RPC invocation on all replicas in the session. The difference is that it is associated with a class – tagging certain methods of the class as broadcast makes the class a replicated type. No special framework is needed to create replicated types from RPC. Thus, we have the flexibility of RPC and the automation of replicated types in one abstraction!

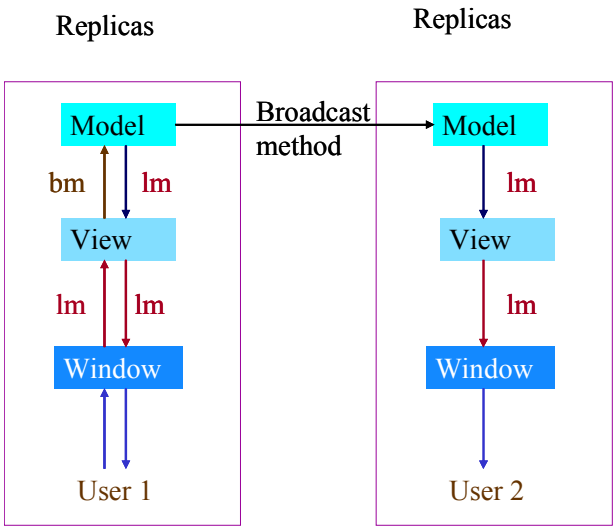


Figure 38 Sharing the Model Layer

Figure 37 shows how broadcast methods were used in Colab to share the model layer. (Here we ignore the toolkit layer). When a user invokes a command, the window layer invokes a local method on the view layer. The view layer, on the other hand, invokes a broadcast method on the local model, which in turn triggers the invocation of the same method on all other replicas of the model. Each of these replicas invokes a local method on its view, which in turn invokes a local method on its interactor and so on until the user gets the feedback.

As the above discussion shows, it is possible to use broadcast methods to share objects in any layer – not just the model layer. Objects occur in every layer – so why not use all the model sharing abstractions we have seen above to share arbitrary layers. It is more difficult to use replicated types to share user-interface abstractions since current systems were targeted at model sharing and do not provide replicated types describing UI abstractions such as windows and text fields. But the more flexible mechanisms - the communication mechanisms and broadcast methods – provide arbitrary abstraction flexibility, which determines the range of objects that can be shared using the mechanism. Thus, they can be used to share UI layers.

Returning to broadcast methods, in Figure 37, broadcast methods were put in the model layer. We could have also put them in the classes in other layers.

Broadcast methods, thus, provide a very useful intermediate point between communication and sharing mechanisms. However, they have their drawbacks:

- Limited sharing flexibility: Since they are implicitly invoked on remote sites, they do not offer as much sharing flexibility as the communication abstractions. The programmer has no control over the remote feedback to a user command, exact set of sites that get the feedback, and when the feedback is given.
- Language-based: They require language support and thus cannot be added as a library to existing languages. *It would be useful to add them to C# as an abstraction above multicast methods.*
- Multi-layer objects: As with replicated types, broadcast methods are difficult to use when an object implements multiple layers and only one of these layers is to be shared. They are easier to reason about when the complete object is shared.
- Error-prone selection: Selecting broadcast methods require much care as a broadcast method must not call other broadcast methods – otherwise the latter will be invoked

multiple times in the remote replicas. For example, the code in Figure 39 is erroneous as the insertAbstract broadcast method calls the insertSection broadcast method. It is easy to make such an error.

```
public class Outline {
  …
  broadcast void insertSection(int i,Section s);
  broadcast void insertAbstract (Section s) {
        insertSection  s);
    }
}
```

Figure 39 Erroneous use of broadcast methods

6.9.2 Property-based Sharing

Consider the last problem illustrated by the code above. It arises because the infrastructure treats an object as a set of methods with the goal of keeping its state consistent with its replicas. As it does not know the relationship between methods and its state, it requires programmer intervention.

If the infrastructure could automatically determine how to retrieve and set the state of an object, then it could also automatically provide the synchronization support. However, the only state of an object that can be automatically determined (by a language-based approach) is the internal or physical state of an object. However, we don't want this state to be synchronized since an object, specially a UI abstraction, may have host dependent state in it. Therefore, we want its external or logical properties - the state available to other objects - to be synchronized

In general, it is impossible to know the mapping between an object's methods and its external state. However, if it is written according to consistent programming conventions, then it is possible to do so(Roussev, Dewan et al. 2000). Consider again the class of Figure 37, this time without the broadcast keywords. Based on the Java Beans conventions, we can deduce (a) the property, Title, of type String, (b) that the following "getter" method:

```
String getTitle()
```

returns the value of the property, and (c) that the following "setter" method:

```
void setTitle(String title)
```

sets the property to its parameter.

The following specification formalizes the JavaBeans conventions for extracting properties from method signatures:

```
getter = <PropType>   get<PropName>()
setter = void set<PropName>(<PropType>)
```

Here, words within angle brackets are pattern variables that are unified (in a Prolog sense) in a search of the method signatures. The result of a successful unification with respect to the above rules is a set of atomic properties, each of which is associated with a name, type, getter method and setter method.

Similarly, we can define our own conventions for determining that the above class also defines a, non-atomic or structured, property – a sequence of sections, associated with methods to insert, remove, lookup, set, and count the elements of the sequence:

insert = void insert<PropName> (int, <ElemType>)
remove = void remove<PropName> (int)
lookup = <ElemType> get<PropName>(int)
set = void set<PropName> (int, <ElemType>)
count = int get<PropName>Count()

Given such rules for various types of properties, one can build a sharing infrastructure that synchronizes them automatically.

As with the broadcast method approach, programmer input is needed to perform the synchronization – here in form of such specifications. However, this approach is higher-level and more flexible as the specifications:

- are given once for a set of naming conventions, each of which may apply to several classes, rather than once for each class.
- do not indicate what is to be shared, only the structure of an object. The sharing infrastructure can decide, based on programmer-specified parameters, what is to be shared and when.

Figure 40 shows that the what and when of sharing is decided by an external parameterized coupler, which is attached to the replicated layers to be synchronized. Examples of synchronization parameters are implemented in Suite's coupler (Dewan and Choudhary April 1991; Dewan and Choudhary March 1995)

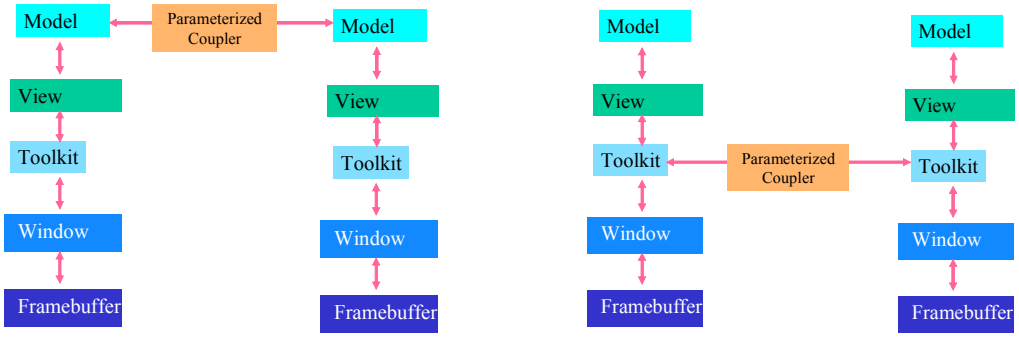


Figure 40 A Single Property-based External Module Coupling Different Layers

Property-based sharing via a single, external coupler has been demonstrated for sharing existing models and toolkits (Roussev, Dewan et al. 2000)

It would be useful to support property based sharing of C# objects for situations where broadcast methods are not appropriate.

Property-based sharing is not without its problems – it depends on a consistent set of conventions being followed. Moreover, it requires computation reflection - the ability to dynamically determine and invoke the methods of an object. Not all systems provide this facility.

6.9.3 Style-sheet based Sharing

Another approach to multi-layer sharing, supported in Suite, is to assume that:

- how objects in the top layer mapped to other layers is determined by "style sheet attributes" . For example, an integer may be mapped to a slider by one attribute and to the blue blackground by another. A slider may itself be mapped to a Swing slider widget (instead of an AWT) slider widget.

- the sharing infrastructure sits below the top layer and can create and bind all of the layers below, solicit events from all of these layers, and make requests to all of these layers

Then the programmer (or user) simply specifies which style sheet attributes are to be shared and the infrastructure enables the sharing by appropriately trapping events and making requests in various layers.

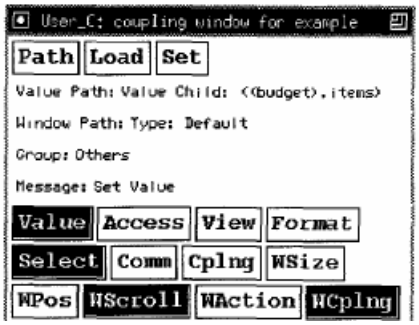


Figure 41 Specifying Shared Style Sheet Attributes

Figure 41, taken from (Dewan and Choudhary March 1995), illustrates this approach. Here the user has specified (among other things) that the value of a model and the scrollbar position of the window in which it is displayed are to be shared, but not its view (e.g. barchart vs pie chart) or the format (e.g. color, font) of the widgets used to create the view. The infrastructure traps events and makes requests in multiple layers to ensure this sharing.

From the point of view of a programmer or user, this approach is the easiest to use. The problems are that it assumes:

- a monolithic infrastructure that knows about all possible application layers.
- standard style sheet attributes – thus, like property-based sharing, this approach cannot be used for existing programs.

6.9.4 Translator-based Adaptive Architecture

Yet another approach to multi-layer sharing, addressing these problems, is to support the sharing of an abstract I/O protocol to which specific I/O protocols are mapped by programmer-defined translators (Chung and Dewan. 2001). For example the abstract protocol could be defined by the following two methods:

void input (Object newInput)
void output (Object newOutput)

For each I/O protocol, a translator module can map the specific operations defined by the protocol into these two abstract methods. A protocol-independent module distributes these abstract operations to the other sites, which are responsible for translating them back to the specific operations, as shown in Figure 42.

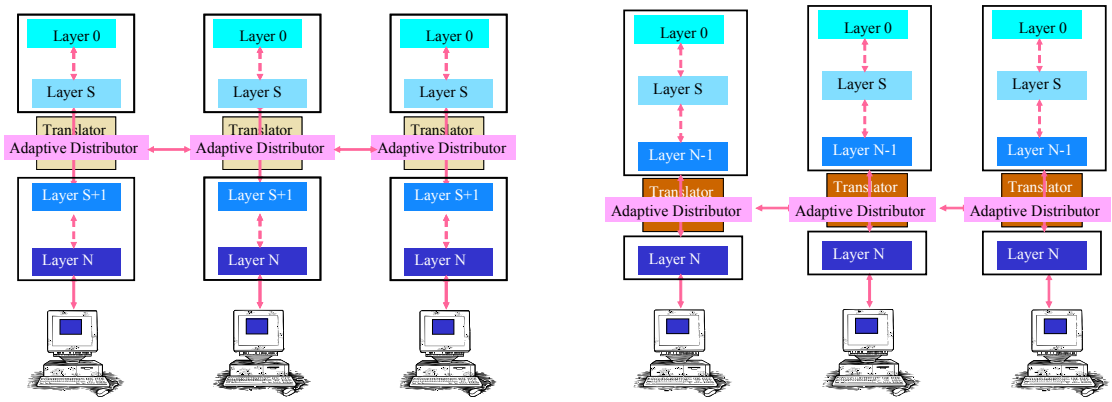


Figure 42 Multi-Layer Sharing using a Single Distributor

In fact, the distributor module supports both the centralized and replicated architectures, distributing input in one case and output in the other, and can even allow dynamic transitions between them. It even supports a hybrid architecture wherein a collaborative application session consists of multiple program replicas, each of which serves multiple user-interface components.

The idea of translator-based distribution was first supported by T 120, which defines an abstract protocol for the window and screen layer. This protocol is more complex than the simple two-method protocol defined above for performance reasons. In fact, the protocol defined by Chung is also more complex than the simple one presented above, also for performance reasons, but is layer independent.

The idea of dynamic architecture transitions is supported for model sharing by PresenceAR. The goal of this product is to support the replicated architecture when the number of collaborators is low and the centralized architecture when it is large, as a special server machine can open more connections than a regular user computer. There can be other dynamic factors such as the network bandwidths and computing powers of the current set of collaborators influencing the choice of the architecture.

The translator-based adaptive distributor has been used to support sharing of a large variety of layers including the VNC framebuffer, X windows, Swing toolkit, and various models. It has been used to demonstrate the usefulness of dynamic architecture transitions in simple experiments.

This framework has the problem that translators must be written – the complexity of which is a function of the complexity of the I/O protocol to be translated. It would be useful to explore an integration of the property-based and translator-based frameworks by automatically translating property changes to operations in the abstract I/O protocol. The integrated framework would provide the automation of the property-based approach and the adaptive architecture of the translator-based approach. It would also be useful to explore support for dynamic changes to the shared layer, not currently supported by the translator-based framework. An even more ambitious project is to allow a user to share different layers with different sets of users (Figure 43)– for example a higher-level (and hence less data-intensive) layer with users connected by a slow network and a lower-level layer with users connected by a fast network. It requires the ability to operate multiple sharing stacks simultaneously.

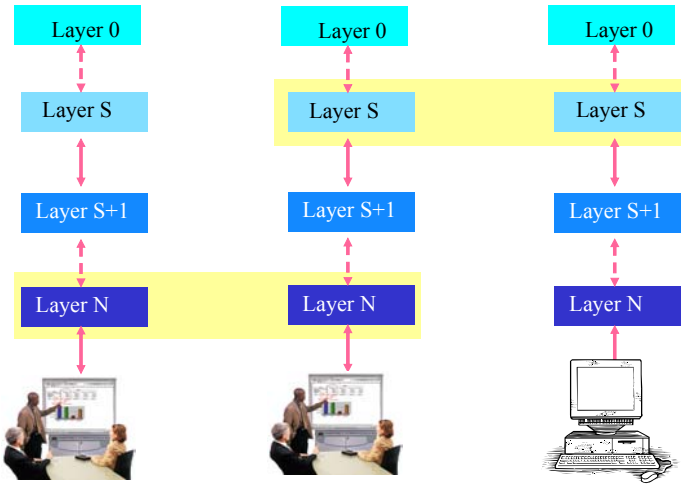


Figure 43 Operating multiple sharing stacks simultaneously

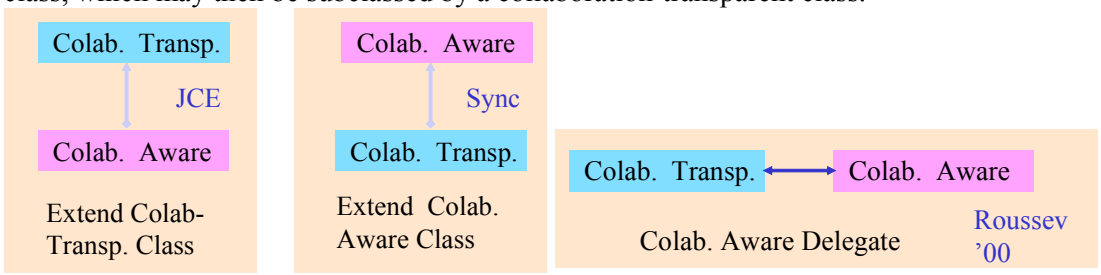
6.10 Making a Layer Collaboration Aware

Creating a translator, directly distributing native events, or using some other means to support sharing of the abstractions in a layer requires adding collaboration-awareness to that layer. How does this code interface with the collaboration-transparent code in the layer?

In some systems such as Suite there is no well defined interface between the two, and the two kinds of code are mixed in arbitrary ways. In other systems there is a better organization.One approach, adopted in systems supporting replicated types, is use the inheritance relationship among the two kinds of code, in two complementary ways (Figure 44 (a) and (b)):

- The collaboration-aware code can be made subclasses of the collaboration-transparent code. As a result, it can intercept the relevant actions of the collaboration-transparent code, invoke sharing functionality (such as multicasting the action to other sites), and then call the superclass implementations of the actions. This approach is supported in JCE to add sharing functionality to AWT toolkit classes.
- Conversely, collaboration-transparent code can be made subclasses of collaboration aware code. Actions added by the collaboration-transparent code are written in terms of the actions defined by the collaboration-aware code, and thus are shared. This approach is supported in Sync, which provides a set of (collaboration-aware) replicated types such as a replicated sequence, which can be used directly or subclassed by the application programmer.

The first approach is appropriate when a fixed set of existing collaboration-transparent classes need to be shared, while the dual approach is appropriate when a fixed set of shared abstractions need to be provided to create collaborative programs. Of course, both approaches can be used together – an existing collaboration-transparent class can be subclassed by a collaboration-aware class, which may then be subclassed by a collaboration-transparent class.



(a) **(b)** **(c)**

Figure 44 Inheritance and Delegation Links Among Collaboration Transparent and Aware Code

The dual of inheritance is delegation, wherein IS-A links are replaced by HAS-A links. This approach is used in the property-based approach described above. Collaboration-aware system classes register themselves as event listeners of collaboration-transparent classes and invoke actions on them. In general, delegation links are more flexible than inheritance links, for at least two reasons:

- They can occur among distributed classes.
- They can be configured at application start and even changed at runtime. Thus, it is possible to change the sharing semantics by linking a different collaboration-aware class to a collaboration-transparent class.

The main disadvantage of delegation links is that they have to be configured by the programmer. It is easy to ameliorate this situation by providing code that creates some popular links equivalent to those defined in the inheritance approach.

A special kind of a collaboration-aware delegate is a proxy that sits between two communicating objects, typically in two different I/O layers, intercepting the messages between the two objects and sharing them with peer objects on other sites. The proxy may sit between:

- two different processes, as in XTV, wherein a pseudo X server sits between the X clients and the regular X server, appearing to the clients to be an X server, forwarding traffic to both the local X server and remote X servers (Figure 45(a).
- two objects, as in COLA, wherein all methods are invoked on an object via its adapter, which provides sharing and access control functionality (Figure 45(b).

The former approach requires no changes to existing code, while the latter provides finer-grained modularity, allowing the adapter for each object to be created independently, but requires changes to application code unless compiler support can be used to replace links to regular objects with adapter objects.

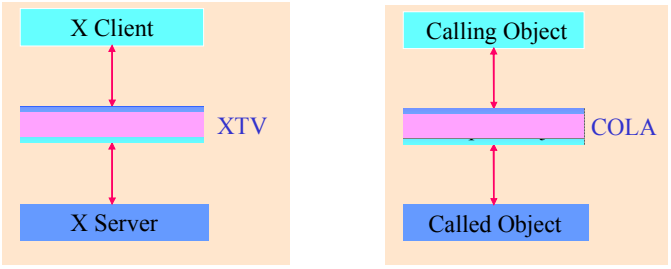


Figure 45 Proxies between communicating processes/objects

6.11 Linking Objects

A final issue in the layered architecture is how objects in corresponding layers on different sites are connected to each other for synchronization purposes. Under session management, we discuss how the layers themselves are linked. Figure 46 shows the difference between layer and object correspondence.

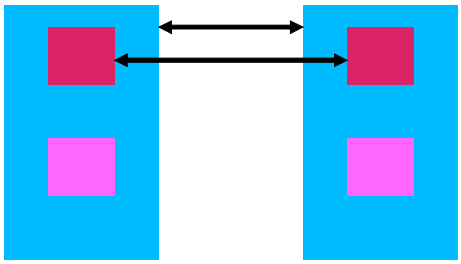


Figure 46 Linking objects in linked layers

As we see in the figure, only a subset of the objects in corresponding layers may be linked.

The linking problem has two aspects:

- Correspondence: How do we identify the peers of an object?
- Channels: How are communication channels among these peers established?

We will consider here only the first aspect. Under session management we consider the second aspect, since the channels are usually established among the containing layers rather than the peer objects themselves.

6.11.1 Manual

The most general approach, of course, is to assume the application programs establish the correspondence among peer objects. This is the approach implemented in Chung's framework, where the peer translators are expected establish the correspondence.

We describe below some of the approaches for doing the correspondence automatically.

6.11.2 Central copy

In some centralized systems such as Suite each replica downloads its objects from a central store. All copies of a downloaded object can, thus, be automatically linked to each other.

6.11.3 Identical programs

Some replicated systems such as Colab assume that the different sites run identical, deterministic programs. In such systems, objects at the same virtual address in each program can be linked to each other. (Recall that in Colab, linking objects does not automatically imply that the objects are fully synchronized – broadcast methods in the object determine the synchronization semantics.)

6.11.4 Instantiation number

A weaker requirement than the above is to assume that corresponding objects at different sites are instantiated in the same order. Thus, we link, the Nth instantiated object at each site to the Nth instantiated object at each of the other sites. This approach is taken in XTV and also in several translators implemented using Chung's framework. Here we assume that the infrastructure can determine the instantiation order, typically by intercepting it.

6.11.5 External description

Another approach, used in Groove and Sync, which support sharing of only the model layer, is to require the programmer to provide, for each application, an external description describing the models and the views to be linked to them. Based on this description, the system instantiates and connects models at each site to local views and corresponding models instantiated by it at other

sites. This is a variation of the above approach in that instead of just intercepting instantiations, the infrastructure does the instantiation.

6.11.6 GIDs

A semi-automatic approach, implemented in T 120 and (Roussev, Dewan et al. 2000), is to assume that each object has a program-assigned ID (within the containing layer) that is globally unique, that is, is shared by all of its replicas. This ID can then be used to link the replicas to each other. All of the above automatic schemes can be considered variations of this scheme wherein the GIDs are implicitly generated by the system based on virtual addresses, instantiation number, and external descriptions.

7 Session Management

As mentioned above, we concerned ourselves only with how correspondence is established among objects of corresponding layers at different sites. We did not address how correspondence is established among the layers themselves, which falls under session management.

7.1 Conference Operations

Session management defines the abstraction of a conference (also called a session, place, and space), with the following basic operations:

- Add application: adds an application (instance) to the conference and makes all existing conferees start an interactive session with the application.
- Delete application session: Closes all interactive sessions with the application and removes the application from the conference.
- Join user: makes the user a conferee and starts interactive sessions with all applications added to the conference.
- Leave user: Stops all interactive sessions of the user and removes him/her from the conference.
- Query: Gives information about the current applications and users in the conference.

Thus, the following sequence of operations on Conference 1:

- Add App1
- Add App2
- Join User 1
- Join User 2
- Add App2

results in User 1 and User 2 together using instances of App1, App2, and App3 (Figure 47).

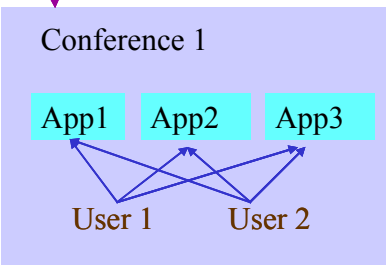


Figure 47 Conference abstraction

It is possible to support several advanced features in session management:

- Join/leave subset of (possibly queried) applications in the conference (T 120).
- Allow one user to eject another (T 120).
- Transfer users from one conference to another (T 120).
- Replace a conferee with another referred by the former (SIP).
- Timed conferences:
  - Set conference duration (T 120, PlaceWare)
  - Query duration left (T 120)
  - Modify duration (T 120)
- Schedule the add operations and allow this schedule to be modified (PlaceWare).
- Keep interaction log of actions and allow this log to be queried (PlaceWare).
- Automatically delete temporary conference (T 120) or passivate persistent conference (PlaceWare, Suite, Groove) when there are no longer any active users. Here, passivating persistent conference means that the running program components are stopped, giving them a chance to save their state on persistent storage.
- Automatically activate passivated conference when first user joins (PlaceWare, Suite, Groove). Activating a passivated conference implies starting the (central or local) component to which the user-interface of the first user connects.

7.2 Linking Layers in Centralized and Replicated Systems

In the discussion above, we never defined the exact semantics of adding an application instance to a conference and starting an interactive session with the instance. They depend on the layered architecture the session management creates.

In a centralized system, adding an application instance loads and starts the program component at the site invoking the add operation (XTV and Suite), or an arbitrary site (T120). Starting an interactive session between the application and a user involves loading and starting the UI component at the user's site and attaching it to the program component (via input and output distributors) as shown in Figure 48 (a).

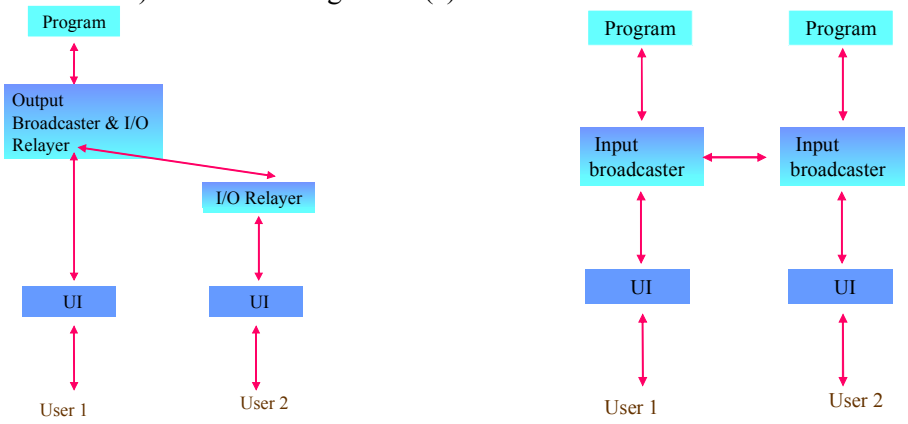


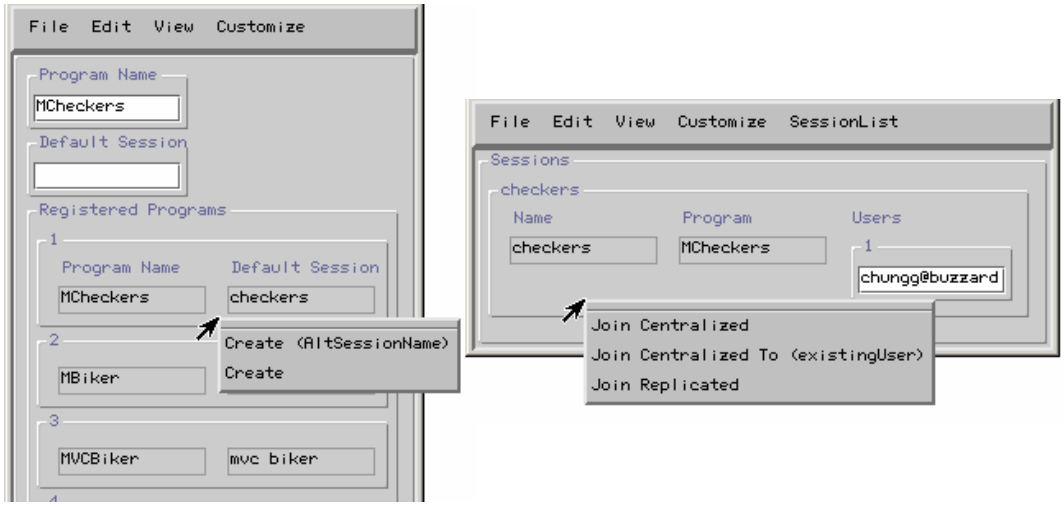
Figure 48 Joining Layers in Centralized (a) and Replicated (b) Systems

In a replicated system, adding an application instance loads and starts the program component at the site invoking the operation. Starting an interactive session with the application from that site loads and starts the UI component at that site and links it to the program component. Starting an interactive session from another site loads the program and UI components at that site and links them to each other and the program and UI components at other sites in the conference, creating the architecture of Figure 48 (b).

Here we assume that there is a static division of the application layers into program and UI components, that is, the shared layers (program) are predefined. This is consistent with current systems, none of which allows the shared layers to be determined at conference start or later times. *Configuring the program component dynamically via session management is a matter for future research.*

7.3 Architecture-Independent Session Management

In fact, most session management systems are hardwired to either the centralized or replicated architecture. An exception is the one created in (Chung and Dewan. 2001), which supports dynamic architecture adaptations. This is illustrated by the user-interfaces of Figure 49.



(a) creating a new session (b) joining an existing session

Figure 49 Session management supporting multiple architectures

Figure 49(a) shows how a new conference (called session) is created – a user selects a program registered with the session manager and creates a new named conference from it. Figure 49(b) shows how a user joins a session – while joining the user specifies whether his user interface is to be connected to a new program replica or an existing program component of a particular user.

7.4 Application-Session Manager Coordination

In order for session management to dynamically create the centralized and replicated architecture, it must know about the attachment points – the UI and program components. Some system are bound to a particular UI. For example, the XTV system is bound to the X server. Similarly, Suite is bound to the Suite user-interface generator. In such systems, only the program component must be specified to the system. In other systems both of these components need to be registered with the system. For instance, in Groove and Sync they are models and views, which are instantiated and connected by the system to create a replicated architecture.

All of the systems above make some assumptions about the programs and UI components. T 120 and Chung's farmework are designed for arbitrary program and UI components. They assume at each site some module that can instantiate the program and UI components and sit as a proxy between them. They simply instantiate and connect instances of this module at different sites to make the layered architecture. For example, in T 120, this module is called an APE (Application

Protocol Entity). When a user at that site joins a conference, the local APE is instantiated and given information that allows it to be connected to peer APEs in the conference. On instantiation, an APE, in turn can instantiate and/or connect to other modules to create the desired application sharing architecture. For example, the window sharing APE connects to the local window server and other APEs to create a centralized window sharing architecture. The Whiteboard APE, on the other hand, instantiates a complete application replica and connects to other APEs to form a replicated architecture.

Once logical connections among the attachment points have been established, corresponding physical channels must be created. These can be created/used by the session manager or the applications.T 120 provides multicast and unicast channels for establishing these connections. Some of these channels are statically created by the system to represent standard sharing sessions such as a whiteboarding session, while other are dynamically created by the application protocol entities.

7.5 Explicit vs. Implicit Session Management

In the user-interface of Figure 49, a user explicitly created and joined a conference by invoking special collaboration commands for this purpose. Other conference operations also require such operations.

It is possible to invoke these operations implicitly as side effects of some non-collaborative actions.

Consider applications that are editors of some persistent artifact. A user starts the editing of an artifact by executing an open command that takes the artifact identity as an argument, and terminates the editing session, by invoking the editor close command to stop interaction. As shown in Suite and Intermezzo(Edwards October 1994), in a collaborative environment, the open and close commands can be overloaded as follows to invoke conference operations as side effects:

- If a user opens an artifact that is not currently being edited by another user, then a new conference is created, the editor of the artifact is added to that the conference, and the user is joined to the conference.
- If a user opens an artifact that is currently opened by some other user, then the user is added to the conference associated with the artifact.
- Closing the artifact removes the user from the associated conference.
- When the last user closes the artifact, the conference is deleted.

Another approach to implicit session management is supported by aura-based virtual environments such as MASSIVE(Greenhalgh and Benford September 1995), where join and leave commands are implicitly executed as a result of navigating actions. In these systems, users and the applications are represented by icons in the virtual environments. User's can navigate in the environment, and their icons (or avatars) represent their position in it. Each application in the environment is part of its own conference. When users' icons come "near" an application's icons, they automatically join the conference associated with the application. Conversely, when they go "far" from the application icon, they automatically leave the conference. "Near" and "far" above are defined by user and application specific parameters.

Implicit session management reduces the effort required to collaborate with others, as no special commands have to be learned to do so. More importantly, it allows for unplanned collaboration

sessions that arise from two users choosing to edit the same object at the same time. There are three main disadvantages:

- Users have no control over whether they join the conference. They can be given the option of joining, thereby doing semi-implicit joins.
- In all of the current systems, a conference is restricted to one application session. No ways have been found to add multiple applications implicitly to the same session.
- Implicit session management is, so far, restricted to artifact- or aura- based applications, and thus is not a general form of session management.

7.6 Invitation-based vs. Autonomous Join

In explicit session management, joins can be invitation-based or autonomous.

In invitation-based joins, a user joins a conference in response to an explicit invitation from an existing conferee. Thus, the join operation itself is a collaborative action, involving input from the inviting and invited persons (Figure 50(a)). In autonomous session management, on the other hand, it is a single-user action carried out by the joining user, who needs some way to identify the conference to be joined. NetMeeting and Groove are examples of systems supporting invitation-based joins, while PlaceWare and Webex are examples of those supporting autonomous joins.

Figure 50 Invitation-based vs. Autonomous Join

Invitation-based joins have several advantages over autonomous joins:

- Anonymous conferences: A conference need not be named, as the conference identity is not specified as part of the join operation. Since naming is a tricky issue in a distributed system, this can be a substantial advantage.
- Implicit access control: No special mechanism is needed to control access to the conference – a user can join only in response to an invitation from an existing user. In the autonomous approach, such a mechanism is needed to ensure that unauthorized users do not join the conference, which can be a substantial overhead. If the conference is recurring, then the overhead is amortized over all of its occurrences.
- Implicit discovery: An invitation is an implicit notification to the invited person about the existence of the conference. In the autonomous approach, a separate mechanism is necessary for the user to discover the conference. Of course, in a scheduled conference, even in the invitation-based approach, the invitees must be notified in advance of the join. However, in an unscheduled, opportunistic conference, such discovery is not necessary.

The autonomous approach, of course, has its own advantages:

- Less per invitee overhead: An existing user does not have to take the responsibility for and action of inviting another user. This can be substantial advantage if the number of participants is large, and the conference is recurring.
- Less message traffic: It requires half as many messages as autonomous join, as shown in Figure 50. Again, this can be a substantial advantage if the number of participants is large.
- Supports mobility: The locations of the participants do not have to be known by others, whereas in the invitation-based approach, the inviter must have a means for directing a message to the invited person's computer. The SIP and Groove invitation-based session managers address this problem by requiring each potential conference participant to register his current device with the system. However, this approach puts an additional burden on the user and raises privacy issues.
- Special mechanisms in the underlying session management and network infrastructure can be used to locate the invitee, but they do not work in all situations.
- Less firewall issues: The autonomous approach requires the ability to "call out" of the joiner's site, whereas the invitation-based approach requires the ability to "call in" to the site. Most firewalls accept calling out but not than calling in. Thus, in the invitation-based approach, any user behind such a firewall cannot join the conference. Of course, in the autonomous approach, the call out is a call in to some other site. The other site, however, can be outside a firewall, if the conference is hosted on a server (as in PlaceWare and Webex) or if that site is the computer of a user outside the firewall. We will see later fixes to this firewall problem in the invitation-based approach.

Each of the two approaches is useful in some situations. For example, invitation-based join is useful for a small ad-hoc conference in which the mobility and firewall issues do not arise, while autonomous join is suitable for large or recurring conferences. Therefore, the T 120 protocol supports both kinds of joins. *It would be useful for NetMeeting/PlaceWare to be extended to support both kinds of joins, by, for instance, integrating Windows messenger and PlaceWare session management, which has probably been done?*

7.7 Session Management APIs

As we have seen above, there are multiple session management policies (and user interfaces) possible, which are appropriate for different situations. A closed system such as Suite is bound to one set of policies and user interfaces, while an open system provides default policies and external APIs to override/replace these to various extents.

7.7.1 2-Person Invites

SIP is perhaps the simplest of these systems in terms of the external interface, so let us consider it first to illustrate what a session management API and associated architecture might look like. Figure 51 shows the basic architecture and operations supported by SIP.

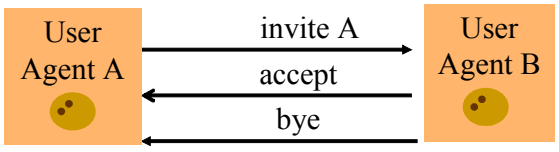


Figure 51 2-Person Invite API

Each site participating in the conference runs a module called a user agent, which talks to other user agents (through proxies, for firewall and other reasons). A user agent can invoke an invite operation on another agent, which can then accept it to start a conference between the two sites. Either user can execute the bye operation to terminate the conference.

SIP is restricted to invitation-based two-person conferences. However, SIP programmers are free to choose other aspects of session management such as whether the operations above are invoked explicitly or implicitly and the user interfaces to invoke explicit operations.

How do we remove the restrictions of invitation-based joins and two-person conferences? Let us incrementally answer this question by considering a series of other APIs, which are inspired by the designs of GroupKit and T 120.

7.7.2 2-Person Autonomous Joins

Figure 52 (a) shows a variation of the SIP architecture and API, which supports autonomous instead of invitation-based joins. As in SIP, the conference is restricted to two persons, and the conference initiator controls who joins the conference.

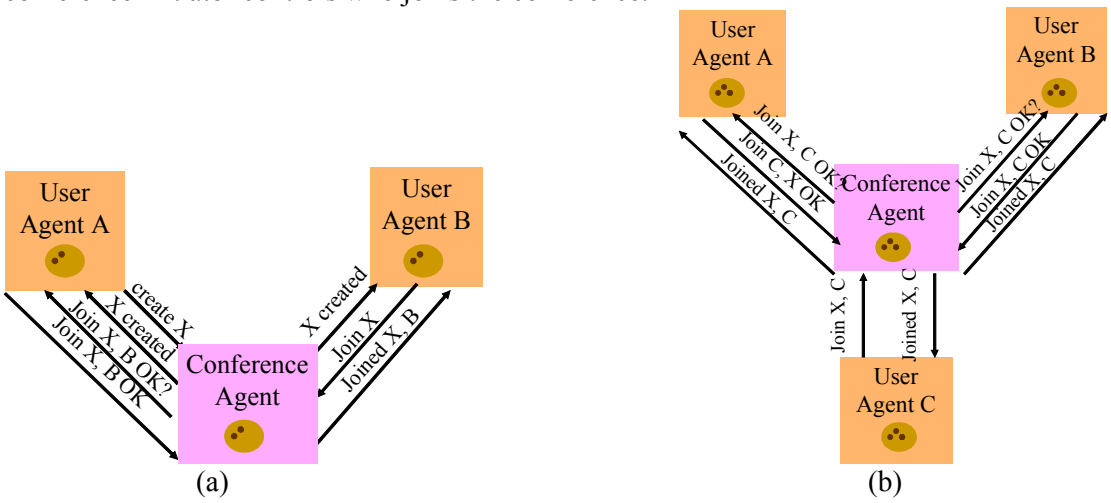


Figure 52 (a) 2-Person and (b) N-person Autonomous Join

This picture has an extra module, called the conference agent, which is at a well-known site. All potential conferees register with it. Instead of inviting another user, the conference initiator asks a conference agent to create a named conference. After completing this operation successfully, the agent informs all registered user agents about the new conference. On receiving this notification, some user agent can autonomously request the conference agent to join the conference. The conference agent checks with the existing conference initiator if this join is acceptable. Upon receiving an affirmative answer from the initiator, it informs both sites that the join has completed. As in the previous case, a bye operation can be executed to terminate the conference. Even the create operation may require an OK from other sites, as in GroupKit, to ensure that scarce resources at the agent are properly used.

This architecture and associated API are perhaps not very useful on their own, but form a basis for a more useful and general mechanism.

7.7.3 N-Person Autonomous Joins

Consider an N-party extension of the above protocol, in which the create operation is as before. Figure 52(b) shows how N users are accommodated. When a new user joins the conference, all existing conferees are asked for if the join should be allowed. On receiving affirmative answers from all of them, the user is joined, and everyone is told about this event. T 120 and GroupKit both provide such a join.

A special operation must now be provided for a particular user to leave the conference without terminating it. In T120, the invocation of this operation does not send an event to all conferees, because if the leaver wishes to inform them, it can explicitly send them messages. Similarly, a special delete operation is now needed. In T120, a special, LastUserLeft event is sent to the conference creator so that he can delete it.

7.7.4 N-Person Autonomous and Invite-based Joins

Figure 53 shows how the above mechanism can be extended to support invitation-based joins.

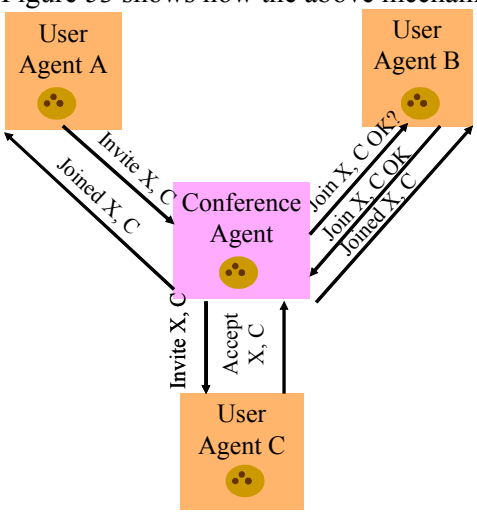


Figure 53 Adding Invitations to Figure 52(b)

New operations are provided to send and accept an invitation. The invite operation behaves as above in that everyone (except the inviter) is asked if this join is agreeable. On receiving affirmative answers from everyone asked, the conference manager does not inform all sites about a successful join, as before. Instead, it sends the invitation to the invitee, who can then accept it, which then causes the successful join event to be sent to everyone else.

An open mechanism similar to the one above has been used in GroupKit to implement three different session management policies:

- Open registration: Anyone can invite, and conference persists after the last user
- Centrally facilitated: Only a special convener can invite.
- Room-based session management: Anyone can join autonomously by entering a room representing the conference.

The mechanism above has performance problems as each access check and notification message is sent to each participant. In T 120 only the notification message must be sent to all sites, but even this broadcast can be expensive, making a conference join an expensive operation. It should be possible to use a publish/subscribe scheme to advertise which control and notification messages a site is interested in. *In fact, it would be useful to extend SIP, using it's*

publish/subscribe facility (not discussed here), to implement such a protocol. Such an implementation would solve the mobility problems of N-party session management systems.

7.7.5 Session-Aware Clients

In the discussion above, we assumed that the message exchanges occurred among modules of session management. Clients of session management (which could be other components of the sharing infrastructure or the application) may also be interested in some of these events to:

- display roster information.
- create communication channels among the involved sites as in T 120.
- exchange capabilities among the various notes, as in T 120 (discussed later)

There are two possible approaches for a client to subscribe to these events:

- It could subscribe directly from the conference agent, like user agents.
- It could subscribe them from the user agent (T 120).

The former approach can result in multiple messages being sent to the same node unless a multicast channel is allocated intelligently for the purpose. The latter approach results in some increased latency as messages must go through an extra process, and, more important, results in the user agent implementing the conference agent interface.

8 Access Control

8.1 Improving Session Access Control

The schemes above protect the join the operation by allowing only authorized users to join a conference. It is useful to also protect other session operations. Check and notification events similar to those associated with join can be added to ensure that only authorized users are allowed to create or delete a conference, add applications, and query current users.

These events allow conference participants to decide interactively who should be allowed to join at the expense of extra work during the conference. It is possible to extend the interactive approach to support traditional access control mechanisms involving passwords (T 120 and PlaceWare) , access control lists (PlaceWare.), and roles. Moreover, it is possible to protect a group of related conferences together, that is, define a single password or ACL for the group. This mechanism is taken in PlaceWare, in which the servlets are associated with a “place”. The place is protected in that it is associated with an ACL, which is inherited by all conferences consisting of instances of these servlets.

It may also be possible to use other, novel access-control mechanisms designed specifically for conferences. T 120 allows a conference to be dynamically “locked” to prevent additional users from joining. PlaceWare provides a more general facility, allowing limits to be set on the number of users allowed to join, which can be changed dynamically. It can also prevent users from joining after a certain time.

In a TechFest and Trustworthy Computing presentation, we demonstrated advanced access control, which mixed many of the above access control features mentioned above (Dewan, Grudin et al. 2003).

8.2 Session vs. Application Access Control

We have talked so far about session access control, that is, checking of session operations such as join. In contrast, application access control checks application operations such as draw circle in a

whiteboard. Session access control indirectly provides application access control in that those who cannot join a session cannot invoke any operations on the applications in the session. Thus, access control of the join operation can be considered a coarse-grained form of application access control wherein a user is allowed to execute all or none of the application operations. However, the join operation may be protected for considerations other than security – for performance or economic reasons only a subset of the users authorized to execute application operations may be allowed to join.

In some situations, finer-grained application access control may be needed. For example, in a presentation, the presenters may have privileges to change the slide which an audience member may not, as in PlaceWare and Webex. Similarly, in NetMeeting, PlaceWare and Webex, a telepointer position is editable only by its owner. Thus, in these situations, application access control is needed in addition to session access control.

8.3 Access Control and Shared Layer

The higher the layer, the more fine-grained the application access control possible as user inputs and application outputs can be more finely distinguished. For example, in a screen sharing system, all input is classified into one group – keyboard input – and the system can allow a user to either provide input or not. Similarly, the system can allow a user to either see all output or none. Thus, a useful access control policy we can build is one in which a user is allowed to either:

- Write screen: see all output but provide no input, or
- Read Screen: see all output and provide no input

These rights are sufficient in many situations, for instance, to ensure the PlaceWare policy of allowing only presenters to provide input.

In a window sharing system, on the other hand, input and output to different windows can be distinguished. Thus, it is possible to define the read and write rights for each window. However, we know of no window system that provides such rights, though they have been proposed in research. *It is important to add this feature to NetMeeting and PlaceWare, specially to support Office digital rights, for example, to ensure that an Outlook window showing a "do not forward" mail message is not shared with others.*

A model sharing system can provide even finer-granularity as it can distinguish among reads and writes of different objects displayed in the same window. Such granularity is exercised, for instance, in Webex, which allows audience members to change their private views of the slides but not the shared view, though both views share a single window.

8.4 Operation-Specific Access Control Mechanisms

To provide arbitrary application-specific control, cooperation from the application is needed, which can take many forms. For example, in Suite and Groove, each operation can query the system for the user who provided the input that triggered the operation, and decide whether it should execute the associated functionality.

Not all systems providing model sharing require applications to have access awareness. Suite extends the notion of "generic" file rights to a set of "generic" tree-based collaboration rights(Shen and Dewan November 1992), and provides automatic enforcement (and specification) of these rights. However, this approach only works when the system knows about and can intercept all application operations. Thus, it does not apply to abstract data types defining their own operations. *It would be useful to explore this approach in the context of XAF.*

8.5 Access Administration

Who should use the (interactive or traditional) access mechanisms to enforce session and application access control? The simple approach is to make the conference creator the access administrator (PlaceWare and T 120). It is possible to use more advanced approaches involving joint ownership and delegation (Dewan and Shen Nov 1998).

9 Concurrency Control

9.1 Access vs. Concurrency Control

Access control determines whether a user is authorized to execute an operation whereas concurrency control determines whether an authorized user's actions conflict with others, and schedules conflicting operations. They are confused with each other because both schemes can prevent a user from executing an operation. In fact, even the user interfaces for the two kinds of control can be identical. Consider a system such as T 120 window sharing that supports floor control, that is, allows only one user to provide input, and provides mechanisms for a mediator to give control to other users. In such a system, the floor may not be given to a user eager to interact because he is not next in line according to the concurrency control schedule implemented by the mediator, or because he is not authorized to use the application.

9.2 Concurrency Control and Shared Layer

Disallowing concurrency is a very simple minded approach to concurrency control – ideally we would like a high a degree of concurrency in the system to reduce the task time. However, the degree of concurrency achievable is a function of the layer being shared – the higher the layer, the more the possible concurrency because the more the information with the system to:

- distinguish among different classes of user input.
- map input to atomic operations.

A screen sharing system must assume all concurrent input is conflicting and must therefore disable concurrency. Even in an application in which no operations conflict, it must still disable concurrency, for a more subtle reason: it does not know the input events comprising an atomic application operation. It must disable concurrency to prevent input events from different users being intermixed.

A window sharing system could make the assumption that input in different windows are non-conflicting, and therefore, allow different users to concurrently interact with different windows. However, the commands to change windows would still have to be executed sequentially. It is perhaps to avoid two forms of concurrency control that none of the window sharing systems allows concurrent interaction. *This is a commercial opportunity as lack of concurrency is one of the biggest complaints against existing window sharing systems.*

In some applications, concurrent input to the same window can be non-conflicting. For example, two users could be concurrently editing different paragraphs or modifying different figures. Model-based sharing, with some application cooperation, can provide this degree of concurrency (Munson and Dewan November 1996).

9.3 Pessimistic vs. Optimistic

The goal of concurrency control is ensure serializability – concurrent processing of the input of users should be equivalent to sequential processing of it in some order. There are two general approaches to implementing serializability:

- Pessimistic: Users are prevented from entering conflicting (that is non-serializable) input. This approach implies locking. Floor control is an approach to pessimistic concurrency control that provides a single, coarse-grained lock controlling all input. It is possible to put finer-grained locks on windows and objects in windows. A system might provide mechanisms to support variable-grained locks. For example, the T 120 multicast layer provides the general abstraction of a lock, called a token, that could be put on arbitrary data structures. The T 120 window sharing subsystem uses these to support coarse-grained floor control, while the T 120 whiteboard uses them to put fine-grained locks on rectangles, circles and other objects it supports.
- Optimisitic: Instead of preventing conflicts, this approach fixes conflicts after the fact, by aborting the actions of one or more of the users whose actions led to a non-serializable interaction. For example, if two users made conflicting edits to the same paper, then one of the user's edits would be aborted. This approach implies replication of the shared data, local unsynchronized changes to it, and consistency checking when the changes are synchronized.

9.4 Merging

The optimistic approach is acceptable in conventional database systems since the concurrent activities are programmed. Aborted actions can later be redone later by re-executing the program that issued them. In collaborative systems, on the other hand, the actions are issued by users, and thus, aborting them destroys the work put in by the user who issued them. Imagine aborting all of a user's edits to a paper!

Therefore a variation of the optimistic approach is used in these systems, wherein, conflicting actions are merged rather than aborted. The result of the merge process is not guaranteed to be serializable– however, it may be acceptable to the users, depending on the specific actions. For example, if one user edited the expected earnings field to "50" and another (concurrently) to "60", and the merge process yielded the average, "55", then this may be acceptable to the users, even though it is not equivalent to any serial execution of the actions. Nonetheless, merging can yield strange results – consider concurrent dragging of the same shape in a Whiteboard. There does not seem to be a reasonable merge of these actions. If you try the PlaceWare Whiteboard, which takes the merge approach, you will see that the merge semantics implemented for these actions is unintuitive.

9.5 Application-Specific Merging

Unlike the pessimistic and optimistic approaches, the merge approach, to yield reasonable semantics, must be application specific. There has been substantial work in working out the merge semantics of text editing, based on an operation transformation approach, wherein the operations issued by users are transformed to implement merging, illustrated in Figure 13. *Recently this work has been extended to tree operations, which will be published in ECSCW 03, and should be applicable to Word and XAF.*

For more general objects, application-specific code, or a merge procedure, is required. Such a procedure can be associated with a file, as in Coda (Kistler and Satyanaraynan February 1992), a relation or table, as in Bayou (Terry, Theimer et al. 1995)and Groove, or an object, as in Rover (Joseph, deLespinasse et al. 1995), IceCube(Shapiro, Fessant et al. 2001). IceCube is a Microsoft Cambidge project. *One of the Bayou inventers, Douglas Terry, has joined Microsoft Bay Area, and implemented this idea for WinFS,. The environment he has created seems to have some of the functionality of Groove – at the file system rather application level.*

An intermediate approach between predefined merging and merge procedures is implemented in Sync. An object is decomposed is merged according to a merge matrix, which is a declarative mechanism for specifying merge semantics. Default values for the merge matrix are provided by the system, which can be overridden by the user. Though not as general as merge procedures, this approach is easier to use, and has been able to accommodate all of the merge policies implemented so far. *It would be useful to implement this mechanism for C# objects.*

9.6 Synchronous vs. Asynchronous Merge

Merging may be done synchronously or asynchronously. Synchronous merging requires simple but efficient merge algorithms and architectures as time between input and conflict detection is small and an erroneous merge destroys relatively less work. In asynchronous merge, the opposite is true. Centralized merge procedures and merge matrices were designed for asynchronous merges, while replicated operation transformation was designed for synchronous merges. As computers get faster, the distinction between the two is dwindling. For instance, Sync, designed for offline merging, has been successfully used to do synchronous merging of text without noticeable delays, but not drag operations in Whiteboards.

9.7 Locking vs. Merging

Given that traditional optimistic concurrency control is a non-starter for interactive applications, it is useful to compare locking with merging.

Merging, as mentioned before, requires replication, and thus inherits all of the associated problems. Therefore it is used mainly in situations where such problems do not exist. For instance, one of the problems is multiple execution of non-idempotent operations. Pure text editing does not have this problem and therefore merging has focused on this scenario.Moreover, merging requires sharing of higher-level operations as it is more difficult to do a reasonable merge of low-level input. Therefore, no window sharing system supports merging.Furthermore, some non-serializable actions cannot be automatically merged, and thus require human intervention.Finally, merging increases the feedthrough time as updates at remote sites have to implement the merge algorithm.

On the other hand, merging has important advantages over locking. More concurrency is possible as users are not blocked because of locks. Moroever, merging allows disconnected (asynchronous) interaction while locking requires connection to other sites to determine lock status. Finally, and perhaps most important, as operations are executed, no (possibly remote) locks have to be checked, which decreases feedback time.

The lock checking time depends on the how locking is implemented. In a centralized locking implementation, all lock information is kept at a central site. Thus, checking lock status requires a roundtrip delay. In a distributed implementation such as T 120, lock status is communicated to all sites (increasing message traffic). Thus a lock check requires no network delay. A lock transfer, however, does involve a roundtrip delay.

9.8 Optimistic Locking

Yet another implementation (Greenberg and Marwood October 1994) keeps locks at a central site, but allows user actions to proceed while the lock check is taking place, making the optimistic assumption that the check will succeed. In case the check fails, these actions are aborted, as in optimistic concurrency control. It is likely that few user actions will actually take place during the checking process. Thus, the cost of an abort is small.

9.9 Floor Control Policies

Another issue in locking is the granularity of locks. As mentioned before, a popular form of locking is coarse-grained floor control, where only one user is allowed to input at any time. There are several variations of floor control possible:

- Host only (T 120) : In a centralized application sharing system, only the user hosting the applications is allowed to input.
- Mediated (T 120): Any one can request the floor, but one or more of the other users has to agree. Moreover, the current floor handler can pass control to another if the latter accepts.
- Facilitated (PlaceWare): This is a special case of the mediated policy wherein a distinguished user, the facilitator or convener, passes floor and approves floor requests.
- Unconditional grant (T 120): Anyone can take the floor autonomously. This is also a special case of mediated wherein no one has to approve a floor control request.

As indicated above, T 120 supports multiple policies. It allows the policy choice to be made collaboratively using the capability negotiation mechanism described later.

9.10 Variable-grained Locking

As mentioned before, T 120 also supports fine-grained locking, providing the general abstraction of a token, with the following operations:

- Allocate/de-allocate token.
- Grab token exclusively/non-exclusively.
- Release token.
- Request/give token.

As mentioned before, this abstraction can be used to implement floor control or associate locks with fine-grained data structures such as Whiteboard objects. However, programmers are responsible for making this association using the above operations.

A higher-level approach to variable-grained locking is supported in (Munson and Dewan November 1996). Objects are broken into properties, which are further decomposed into sub-objects until atomic objects are reached. Users can associate these properties with inheritable lock tables that support variable-grained hierarchical locking. This mechanism can be built on top of the T 120 mechanism.

These abstractions are relevant to Office 12 scenarios, which involve, for instance, fine-grained, implicit locking of Word paragraphs.

10 Interoperability

A collaborative system involves distributed sites cooperating with each other to exchange information. In order to do so, they need to agree on the protocols they use for communication. Interoperability issues arise when they cannot make assumptions about the exact protocols other sites will follow. These issues will not arise if all sites are uniform. However, assuming uniformity is not practical in collaboration systems as one non-conforming site can prevent adoption of collaboration technology. For example, today, Groove runs only on PCs, which seems like a good business decision given that 80% (check this) of the computers today are PCs. However, this means that in a collaboration involving 5 random sites, one of the sites is likely not be a PC, and thus prevent the use of Groove in the collaboration.

Protocol decisions involve, for example:

- What codecs are to be used in audio/video communication?
- Which drawing operations can be invoked on remote sites?
- How many bits are in a pixel?
- What is the size of the virtual desktop?
- How many pixmaps and colormaps can be cached at remote sites?

The general approach to addressing interoperability is to:

- Define one or more "standard" protocols (for some collaboration aspect) that the collaborating sites follow directly or to which the specific protocols supported by the sites can be translated.
- When there are multiple standard protocol for a specific collaboration aspect, provide facilities for selecting one of them.

10.1 Standard Protocols and Shared Layer

In general, the lower the shared layer, the easier it is to agree to a protocol for communicating about objects in the layer. The reason is that lower the layer, the less complicated the kind of objects it defines, and thus less variation in the implementation of these objects. To illustrate, compare the screen, window and model layers. As we saw in section 6.3, the screen layer can be defined by three simple operations: an output operation to display a rectangles, an input operation to communicate key events, and another to communicate mouse events. Interoparation among different implementations of the framebuffer is relatively straightforward, as the success of VNC shows.

On the other hand, in a model layer, arbitrary programmer-defined types can be used, which in turn, can be implemented using predefined types provided by different programming languages. Interoperation among different types defining the same user-level object has been found to be an extremely different problem.

An intermediate layer, such as the window layer, offers an interemediate degree of interoperation. It is far more complicated than the framebuffer layer, as the size of the documentation on the X window system illustrates. As a result, there is far less uniformity. For example, WinCE and Windows, two products of the same company, are not identical. Interoperation of different window systems is possible but complicated, as the size of the documentation of the T 120 interoperable window sharing document illustrates.

It may even be possible to interoperate the model layer. Web services are increasingly being considered a standard data model, but require the implementer of each type to provide the translation between the type and the equivalent web service. The property-based approach, mentioned above, can be used to do the translation automatically.

10.2 Choosing a protocol

As mentioned above, more than one standard protocol may exist for carrying out some aspect of collaboration. How is a specific protocol chosen?

10.2.1 Enumeration & Selection

SIP demonstrates a simple approach for two-way collaboration. One party proposes a series of alternative protocols for some collaboration aspect (called a media) and the other party chooses one of them. For example, one party may send the following messages:

M= audio 4006 RTP/AVP 0 4
A = rtpmap: 0 PCMU/8000
A = rtpmap: 4 GSM/8000

giving two alternativesfor the audio media. The other party can now respond with:

M= audio 4006 RTP/AVP 0 4
A = rtpmap: 4 GSM/8000

picking one of the alternatives.

10.2.2 Multi-Round Negotiation

The above approach works for multiple parties if each pair of parties is free to choose its own protocol but not when a common protocol must be used by all. One can imagine extending it so that each responding site also enumerates the possible alternatives, and then the original site picks one of the common protocols and communicates this information to all of the others. This requires 3 rounds of messages: enumeration, responses, notification.

10.2.3 Level-based Single Round Negotiation

A more efficient mechanism, implemented in T 120, requires only one round of messages if we make the following assumption:

> Alternative protocols can be ordered into levels where support for protocol at level l indicates support for all protocols at levels less than l.

Now each site simply broadcasts the level, and each site determines the minimum supported level and picks the protocol at that level.

10.2.4 Capability Negotiation

We have assumed above that identities of protocols must be communicated to determine the exact protocols that are used. Sometimes these protocols must be indirectly determined by the capabilities of the collaborating sites, and it is thus important to exchange the capabilities, from which the protocols can be derived. For example, consider a window sharing system. In such a system, each site can indicate the exact set of graphics operations it supports. Based on this information, each site knows which of these operations it can invoke on other sites.

We can extend the above level-based algorithm to support capability negotiation by allowing each level to be a capability set, with increasing levels indicating larger sets. For example, T 120 creates increasing sets of drawing operations, and associates each of these sets with a level. We can further extend this idea by allowing the levels to be values of specific parameters that again determine the protocol used. For example, a T 120 level can be the desired size of the virtual desktop or the number of bits in a pixel. Now, instead of always doing a min of the received level, each of the above sites, can choose:

- The minimum of the received levels. If the level is a capability set, this essentially ANDs together the advertised capabilities.
- The maximum of the received levels. If the level denotes a capability set, this essentially ORs together the advertised capabilities.
- Something else based on the number and identity of each site supporting a particular level.

The exact algorithm does not matter as long as all sites statically agree on the same one for the collaboration aspect being negotiated. For example, in T 120, negotiations of the set of drawing

operations and the number of bits in a pixel use the minimum value, while negotiation of the virtual desktop size uses the maximum value.

10.2.5 User-Interface Policy Negotiation

This approach can be used to also negotiate user-interface policies, which constitute human-human protocols. For example, in T 120, to determine if the floor should be granted to a user requesting it, each user can indicate, always, never, or ask floor handler, where always < ask floor handler < never. Each site chooses the minimum of these values to support the least permissive control. Similarly, to determine if the scrollbars of users should be coupled, each user can say no, or yes, where yes < no. Again, choosing the minimum value supports the least permissive sharing.

10.2.6 Negotiation among versions

The approach can also be used to negotiate among multiple versions of the same product, as not all collaborators can be expected to have the same version. For example, for a powerpoint presentation, each site can indicate if it has a PDF viewer, a PowerPoint viewer, or a full PowerPoint application, with PDF viewer < PowerPoint viewer < Full App, and choose the minimum of these values. Similarly, each site running Office can indicate if it has Office 10, 11, or 12, with Office 10 < Office 11 < Office 12, again choosing the minimum.

10.3 Translating down to negotiated values

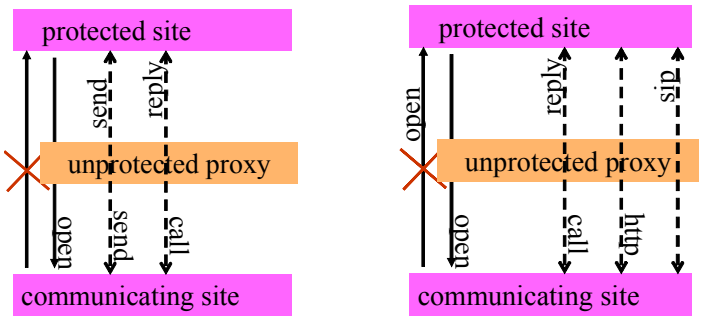
Once a common set of capabilities has been negotiated, a site with a richer capability set may need to translate down to the common set. For example, if the negotiated set of drawing operations includes “draw pixmap” but not “draw ellipse”, and an application issues the latter operation, then a shared window system must translate it to the former before sending it to a remote site. Similarly, a system supporting fine-grained locks communicating with a system supporting only floor control would need to translate lock requests to floor requests, and vice versa (Dewan and Sharma Sep 1999).

11 Firewalls

In our discussion so far, we have assumed that all collaboration sites connected to each other via the internet can in fact communicate with each other, opening connections to other sites and sending messages to them. To prevent denial of service attacks, many organizations protect sites from others by putting different degrees of “firewalls” around them.

11.1 Basic Firewall

The most permissive firewall allows a protected site to open connections to other sites but not the reverse (Figure 54 (a)). Traffic can flow in both directions, in the form of message passing or remote procedure call by the protected site, as shown in the figure. The firewall ensures that a site does not receive unsolicited traffic.



(a) Basic Firewall **(b) Protocol-based Firewall**

Figure 54 Different Degrees of Firewalls

To prevent opening of connections to a protected site, the IP address of the site is not exported outside the firewall. However, to receive messages from other sites, some way of addressing the protected site must be found. The solution is to connect to the outside world through a proxy that sits outside the firewall in the DMZ (de militarized zone) of an organuzation, which multiplexes and demultiplexes traffic in the two directions, as shown in the figure. Routing messages through a proxy makes it easy to break connections to the rest of world in case something goes wrong.

11.2 Restricted Protocol

The reason why something can go wrong is that no restrictions are put on the sites with which a protected site communicates. One common restriction is to limit protocols used to the ones trusted or deemed absolutely necessary by the organization. Examples of such protocols are HTTP and SIP, as shown in Figure 54 (b). It is possible to use some of these protocols to build higher-level and possibly more general protocols, such as RPC. For instance, HTTP has been used to build the SOAP RPC protocol. This goes against the idea of restricting protocols. Therefore, some firewalls try to look at the exact traffic being communicated to prevent more general protocols. However, we will not consider such firewalls here.

11.3 Firewalls and Service Access

Traditional distributed applications tend to be “service access” applications, wherein some user site accesses the capabilities of another site providing some service to the user. Figure 55 shows how such applications can be facilitated in the presence of firewalls. The latter, server, site is placed outside the firewall. The user sites, however, can be behind firewalls. An application on the user site can use an unrestricted or restricted protocol based on the nature of the firewall around it. In fact, if it uses SOAP, this adaptation would be done automatically (I hope).

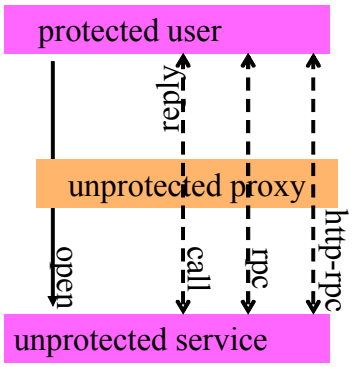


Figure 55 Accessing distributed services through firewalls

11.4 Forwarder

It is more challenging to support collaboration through firewalls because two protected user sites may need to communicate with each other. The solution, shown in Figure 56, is to use an unprotected intermediary, which forwards traffic in both directions. A protected site opens connection to the forwarder site when it is available for collaboration.

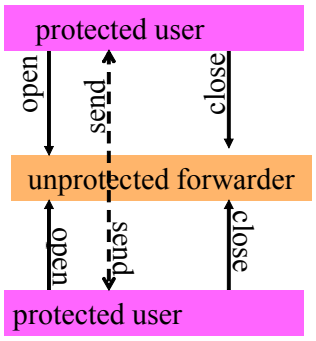


Figure 56 Collaborating via a forwarder

This approach works well if unrestricted sends are allowed and used. What if the applications wish to use RPC or if only restricted protocols are allowed by the firewalls?

11.5 Sending data to protected site over one-way RPC

If a protected site is allowed to only communicate using restricted protocols, then it must "cheat" as in the service connection case, and build the actual protocol it uses over these restricted protocols. It is of course not always possible to build an arbitrary protocol over another. In particular, it is difficult to initiate a data transfer to a protected site over HTTP. It is possible for the protected site to initiate a data transfer to itself by getting the data. But there is no direct support to send unsolicited data to the site. Such a facility is important in collaborative applications, since the various sites work independently and do not know when other sites may have data for them. This problem occurs in general whenever the restricted protocol supports one-way RPC interaction – the protected site can issue calls but not service them.

This problem can be addressed in various ways:

1. Polling: the protected site can periodically poll other sites for data. This solution works for any one-way call model, but allows only semi-synchronous collaboration or collaboration in which data arrive at regular intervals – such as in video or audio.
2. Blocked calls: HTTP allows a site to post a get information request that blocks if the information is not available. Such a request may be automatically timed out after a certain period. After each data receipt and timeout, the site can post such a request. This approach can be used in any one-way call model that allows the caller to block if the callee is not ready to service it.
3. Use separate notification protocol: It may be possible to send small amounts of data to a protected site through an alternative firewall-safe protocol such as SIP. Such a protocol must be provided to support invite-based joins. In this case, the protected site may be sent a notification through this protocol about the availability of new data, and it can then

invoke a regular call (using the regular firewall-safe protocol) to fetch the data – essentially following the MVC model for communicating updates. If the amount of data transferred is small, then the complete data transfer may occur through the alternative protocol. *To make such transfer easy to program, it would be useful to build RPC over the protocol – in particular, it would be useful to build RPC over SIP.*

11.6 Firewall unaware communication

Ideally, the application and even other components of the infrastructure should be unaware of the nature and even presence of a firewall. This can be done by building firwall traversal into the communication abstraction. In Groove, it is built in the replicated types abstraction. It would be useful to provide a firewall-unaware byte-stream channel.

At UNC, we have built firewall traversal based on the MVC-style communication abstraction. We assume that whenever any information must be sent to a protected site, a well-defined interface is used to send the notification. The interface has been implemented on top of various communication abstractions:

- RMI: we assume here that the protected site implements RMI invocations, that is, there is no firwall.
- Web Service (SOAP): Like the above case except that we assume the protected site can implement web service to receive notifications,
- SIP: Notifications received through Java's SIP.
- Blocked SOAP calls: The protected site receives notifications in blocked SOAP calls (via blocked gets of HTTP underneath).
- Polling: The protected site polls a SOAP service for the data.

The exact transport can be configured externally. We are currently comparing the performance of these various implementations described above.

11.7 Firewall traversal & latency

Layering over a restricted protocol can add overhead, which is a more significant problem for interactive collaborative applications than for service access applications. Communicating through forwarders further adds to the latency. This is the reason why, in Groove:

1. first a direct connection between the user sites is tried,
2. if that fails, because one of them is protected, then a forwarder-based connection directly implementing the desired protocol is used,
3. and if that fails, because the protected site is restricted to certain protocols, then a forwarder-based connection indirectly implementing the desired protocol over the restricted protocol is used.

Thus, Groove can also operate as a service model, and not just as a peer-to-peer system, as is commonly believed.

The Groove algorithm can result in an asymmetric communication pattern – the communication from a protected site would go directly to an unprotected site, but communication to it may go through a forwarder and over a restricted protocol..

PlaceWare also tries the latter two alternatives. Its service model requires it to use a forwarder, which adds to the latency of interaction, which is noticeable by the end user.

11.8 Forwarders & congestion control

A more subtle problem with forwarders is congestion control, which involves reducing message production when the end to end latency crosses a certain threshold. Such congestion control is built into the implementation of TCP/IP, the underlying communication protocol for most communication abstractions presented here. However, TCP/IP congestion control does not work in the presence of a forwarder because two TCP/IP connections exist between communicating sites, one from the source site to the forwarder, and one from the forwarder to the destination site. These two links may experience different congestions, making the TCP/IP congestion control algorithm ineffective (Figure 57).

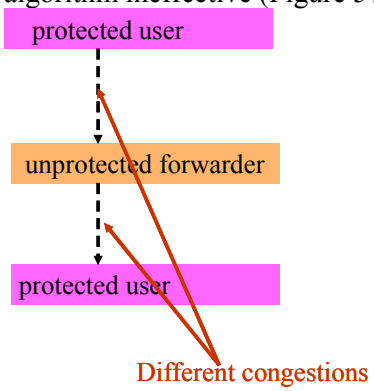


Figure 57 Forwarders break TCP/IP congestion control

For example, there may be no congestion to the forwarder (which is likely to be on the same LAN as the source site), but there may be heavy congestion to the destination site. TCP/IP does not know the relationship between the two links, and thus, cannot tell the source site to reduce the traffic.

T 120 window sharing addresses this problem by requiring the I/O distributors to implement their own, application-specific, congestion control algorithm. It is not clear PlaceWare, Windows Messenger and other products address this problem, which I believe is noticeable in the current implementation of Windows messenger.

So far, we have considered the negative effects forwarders have on performance. We consider below the positive impact possible.

11.9 Forwarder + Multicaster

A forwarder can multicast messages on behalf of the sending site, as shown in figure 58 (a). This approach is more modular in that it allows application processing and I/O distribution to be separated on different machines. It reduces the message traffic to the forwarder, which is an issue if the link to the forwarder is congested. Moreover, it reduces the computation load on the protected site, which is an issue if the number of destination sites is large.

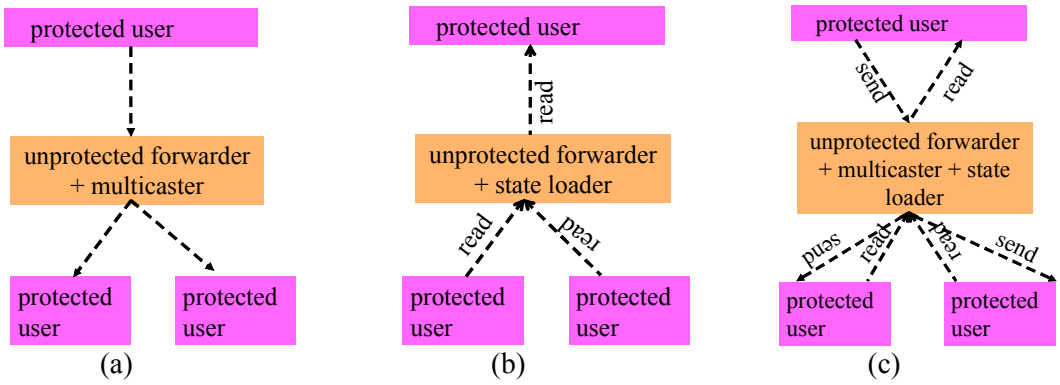


Figure 58 Forwader (a) + multicaster , (b) + state loader, (c) + multicaster + state loader

The idea of a separate multicaster is useful even when there are no firewalls. In Groove, if a protected site is connected to the Interner via a slow link, then a multicasting forwarder is used to deliver its messages, even if it is not behind a firewall. Similarly, T 120, which is not designed for firewalls, provides special machines for multicasting. In fact, it provides a hierarchy of such machines to reduce the latency and the chance that duplicate copies of a message are sent on a physical link. In particular, it supports the dumb-bell architecture wherein each organization goes to its own multicaster rather than a central multicaster, as in PlaceWare.

11.10 Forwarder + State Loader

The multicaster approach is designed for the push model of data communication. Figure 18(b) shows the dual approach for the pull model. Here the data read by the forwarder is cached so that subsequent reads of it do not cause a message to the unprotected site. This approach has the advantage of the pull model in that each site can consume data at its own rate. It has the disadvantage of the pull model – two messages are needed to transfer data from a producer to a consumer rather than one, thereby increasing the latency and traffic. As in the multicast case, one can have a hierarchy of state loader to reduce duplicates.

11.11 Forwarder + State Loader + Multicaster

Naturally, it is possible to add both multicasting and state loading capability to a forwarder, as shown in Figure 18 (c).

12 Composability

As mentioned above, the idea of a multicaster/state loader is independent of a forwarder, though it is convenient to integrate the two, which brings us to the general question of composability.

As we have seen in this paper, a collaboration infrastructure must offer a variety of services such as session management, connection of program and UI layers, I/O distribution, access and concurrency control, and firewall traversal. We have also seen that there are multiple ways to offer each of these services. Hence it would be desirable to implement each of these services in separate composable modules. In practice, doing so if difficult because the various services must work with each other and defining the interfaces between them is difficult.

12.1 T120 Components

T 120 is perhaps the most ambitious effort at defining such interfaces. Some of these are identified below together with the functionality they provide access to:

- Multipoint: facilities to dynamically configure a hierarchy of multicast servers, send messages using this hierarchy, and create and use locks (called tokens). Used by all of the other layers.
- Generic conference control: session management and capability negotiation.
- Node controller: the interface of a site that can participate in a conference.
- Application protocol entity: the general interface of an application that can be added to a session. It is used by session management and uses multicast and generic conference control.
- Window-based sharing: facilities to provide centralized sharing of windows. It uses generic conference control, multicast and application protocol entity.
- Whiteboard sharing: facilities to provide replicated and/or centralized sharing of windows. It Uues generic conference control, multicast and application protocol entity.

Figure 59, taken from T 120 edocuiments depicts some aspects of the layering implied above, graphically:

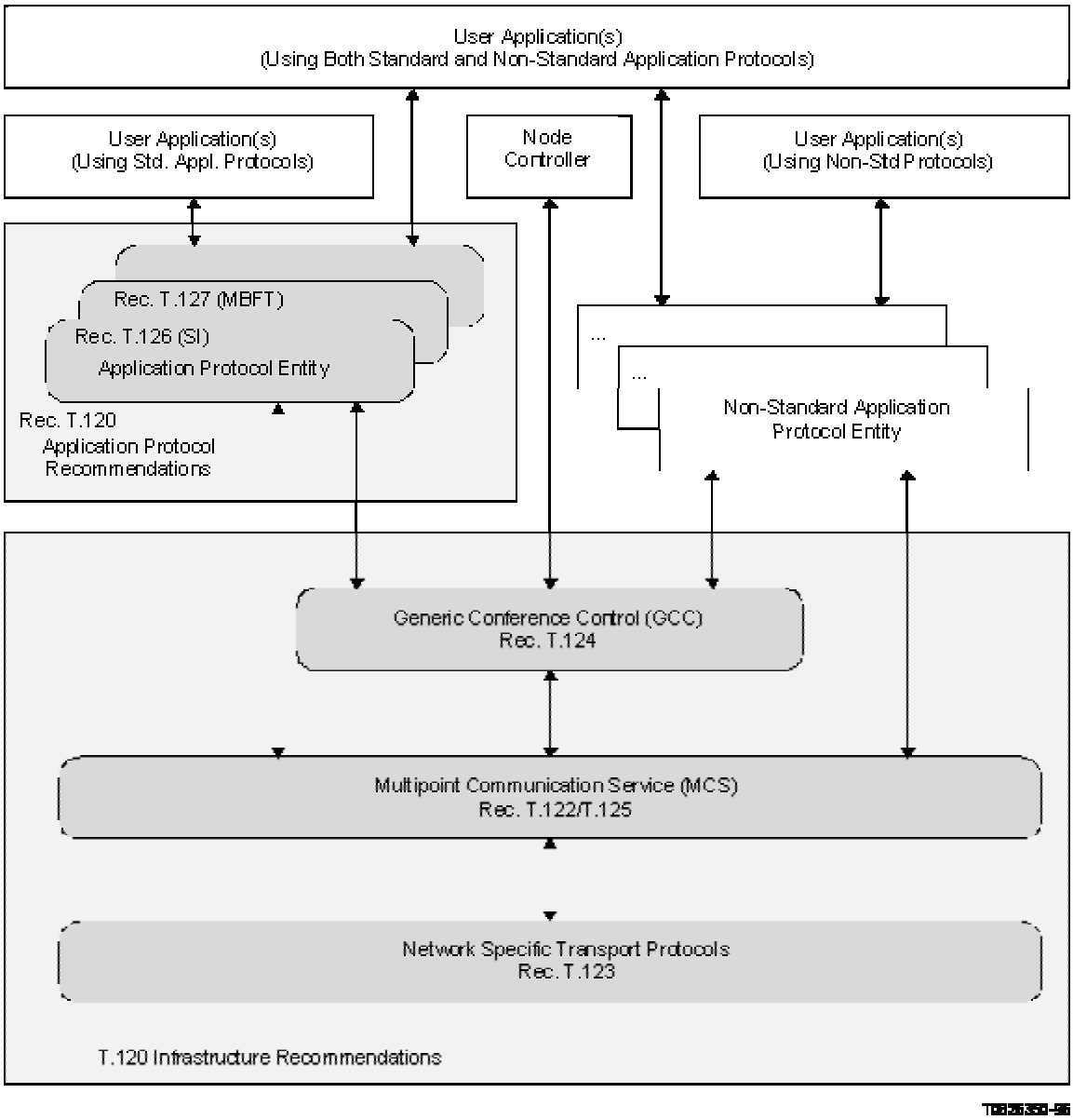


Figure 59 T 120 Components

12.2 Composability vs. Integration

There are several advantages of creating separate, stand-alone infrastructure components:

- Use what you need: Programmers can use just the components they need, without the overhead of learning, paying for, or installing software they do not need. For example, as mentioned earlier, in the collaborative video project, the programmers only used the multipoint component of the T 120 framework.
- Substitutability: It is possible to easily change part of the infrastructure functionality without changing the implementation of the remainder. For instance, it should be easy to replace the implementation of the T 120 multipoint layer without affecting the remaining components.
- Orthogonality: Componentizing forces one to not link together orthogonal aspects, allowing combinations of functionality that may not be possible in an integrated system. For example, by separating communication and sharing facilities in T 120, it is possible to use the communication facilities for both the centralized and replicated architecture. In systems such as Suite not providing this separation, the communication facilities are tied to a specific architecture. Similarly, by not tying the communication and sharing facilities to a particular programming language, applications written in multiple programs can be accommodated.

On the other hand, designing the entire infrastructure functionality in an integrated way has the advantage that the programmer has to do less work as components do not have to be connected to each other. To illustrate, consider the tasks T 120 and Suite programmers have to do to create a new collaborative application. In T 120, the programmer has to:

- Create a multicast channel for the application and join it.
- Create an application protocol entity representing the application to T 120 that invokes the actual application components on various sites and connects them.

Creating a Suite application is much easier as the equivalents of these two steps are automated since the infrastructure assumes a specific architecture (centralized) and language (C). The programmer simply creates a C programs and connects it to a generic “dialogue manager”. The infrastructure uses this connection information to essentially create a multipoint connection between them. A separate application protocol entity instantiating the C program/dialogue manager is not needed.

Of course, it is possible to get the best of both worlds by creating components, and also connectors that link the components in typical ways. For example, it should be possible to create the Suite model on top of the T 120 model by:

- Creating an application protocol entity for C programs.
- Ensuring that when the C program is instantiated, a new multicast channel is created and the program automatically joins it, and when a dialogue manager connects to the C program, it also automatically joins the channel.

12.3 Improving Components

There are no layers in T 120 corresponding to the RPC and replicated type abstractions. *Thus, it would be useful to extend the model to support these abstractions, perhaps based on Web Service and C#.*

Yet another problem with T 120 is that input and output sharing is integrated, thereby not accomodating input-only sites. Such sites would be useful in collaborations involving multiple users sharing a single large screen display (Figure 60). Here, it would be useful if the users could use their PDAs to input data, such as answers to poll questions. *Thus, separating the input and output sharing in PlaceWare is another possibility that should be explored.*

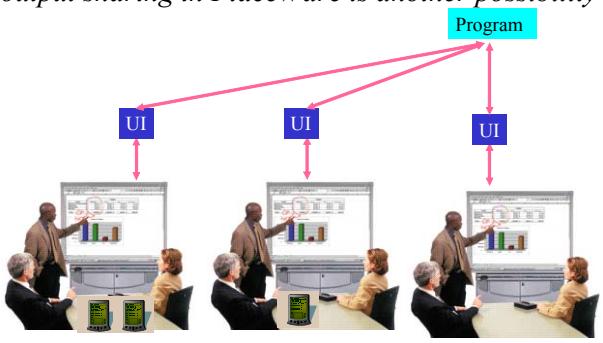


Figure 60 Sharing PDA input

13 Conclusions

The designer of a collaboration infrastructure must worry about several issues:

- Application architecture : dividing application layers into program and UI components and connecting these components into centralized, replicated, or hybrid architectures.
- Session management: linking applications and users in a conference abstraction.
- Coupling: sharing the input and output of the users.
- Access control: protecting unauthorized sharing.
- Concurrency control and merging: Preventing or merging inconsistent concurrent interaction (of authorized users).
- Interoperability: Allowing autonomous sites and users to agree on common communication protocols.
- Firewall traversal: Allow users behind different kinds of firewalls to efficiently participate in the collaboration.
- Componentization: Divide collaboration functionality into independent components and providing components to compose them in popular ways.

As shown in the paper, each of these issues can and has been resolved in multiple ways by commercial and research systems. Each resolution has its advantages and disadvantages. A comprehensive solution to the collaboration problem must support as many of the current approaches as possible. Most current systems, in particular Microsoft products, provide a narrow range of the useful approaches. Therefore, there is great potential for extending these products.

Specifically, it would be useful to investigate the following:

- Centralized window sharing.
  - Remove expose coupling.
  - Add window-based access and concurrency control.
  - Provide multi-party sharing through firewalls without adding more overhead than necessary.
- Replicated window sharing,
  - Investigate it as firewalls would be more willing to allow its low bandwidth traffic.

- Tolkit sharing
  - Explore a replicated User or Avalon toolkit.
- Model sharing:
  - Decouple architecture and replicated types by using delegation rather than inheritance for adding sharing functionality.
  - Build a general property-based abstraction on top of replicated types to share collaboration-unaware C# types.
  - Support broadcast methods in C#.
- Multi-Architecture Sharing:
  - Allow users to dynamically change the level of sharing by transparently switching between NetMeeting and PlaceWare.
  - Support replicated/centralized/service-based centralized architecture for single/local/remote collaborators .
  - Provide a single system supporting arbitrary and dynamic architecture adaptations.
- Communication abstractions:
  - Extend the Indigo RPC so it meets the needs of collaboration:
    - Add Multicast-RPC
  - Provide a separate stream multicast layer for language neutrality and composability.
  - Implement multicast RPC over the stream multicast.
- Coupling
  - Add externally configurable filtering component to determine which events are shared, with whom they are shared, and when they are shared to support flexible sharing.
- Concurrency control.
  - Support the different floor control policies for window- and screen- level sharing.
  - Support property based merging of arbitrary C# types.
  - Support T 120-like tokens to build fine-grained locks.
  - Support regular and optimistic locking of these tokens.
  - Support property-based locking for C# types.
  - Implement this locking over tokens.
- Session Management
  - Build N-party, publish/subscribe based session management on top of SIP
  - Support
    - Implicit and explicit session management.
    - Invitation-based and autonomous joins.
- Collaborative applications
  - Provide a replicated type for XAF trees.
  - Use tree-based operation transformations to support merging of XAF trees.

14 References

Abdel-Wahab, H. and K. Jeffay (1994). "Issues, Problems and Solutions in Sharing X Clients on Multiple Displays." Internetworking: Research and Experience **5**: 1-15.

Abdel-Wahab, H., O. Kim, P. Kahore and J. P. Favreau (April 1999). Java-based Multimedia Collaboration and Application Sharing Environment. Colloque Francophone sur l"Ingeniere des Protocoles (CFIP'99).

Abdel-Wahab, H. M. and M. A. Feit (April 1991). XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. Proceedings IEEE Conference on Communications Software: Communications for Distributed Applications & Systems.

Ahuja, S., J. R. Ensor and S. E. Lucco (1990). A comparison of application sharing mechanisms in real-time desktop conferencing systems. ACM Conference on Office Information Systems.

Ben Y. Zhao, L. H., Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz "Tapestry: A Resilient Global-scale Overlay for Service Deployment." IEEE Journal on Selected Areas in Communications.

Cadiz, J., A. Balachandran, E. Sanocki, A. Gupta, J. Grudin and G. Jancke (2000). Distance Learning Through Distributed Collaborative Video Viewing. CSCW.

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

Chung, G. and P. Dewan (October 1996). A Mechanism for Supporting Client Migration in a Shared Window System. Proceedings of the Ninth Conference on User Interface Software and Technology.

Chung, G., P. Dewan and S. Rajaram (1998). Generic and composable latecomer accommodation service for centralized shared systems Proc. IFIP Conference on Engineering for Human Computer Interaction, Chatty and Dewan, editors, Kluwer Academic Publishers.

Chung, G. and P. Dewan. (2001). Flexible Support for Application-Sharing Architecture. Proc. European Conference on Computer Supported Cooperative Work, Bonn.

Danskin, J. and P. Hanrahan (94). "Profiling the X Protocol." ACM Multimedia.

DataBeam "T`120 Framework." www.imtc.org/t120body.htm.

Dewan, P. "Coupling and Awareness in WYSINWIS (What You See Is Not What I See) Collaboration." \\t-pdewan1\public\coupling.doc.

Dewan, P. (1993). "Tools for Implementing Multiuser User Interfaces." Trends in Software: Issue on User Interface Software **1**: 149-172.

Dewan, P. (1998). "Architectures for Collaborative Applications." Trends in Software: Computer Supported Co-operative Work **7**: 165-194.

Dewan, P. (October 1990). A Tour of the Suite User Interface Software. Proceedings of the 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology.

Dewan, P. and R. Choudhary (April 1991). Flexible User Interface Coupling in Collaborative Systems. Proceedings of the ACM CHI'91 Conference.

Dewan, P. and R. Choudhary (March 1995). "Coupling the User Interfaces of a Multiuser Program." ACM Transactions on Computer Human Interaction **2**(1): 1-39.

Dewan, P. and R. Choudhary (November 1991). Primitives for Programming Multi-User Interfaces. Proceedings of the 4th ACM SIGGRAPH Symposium on User Interface Software and Technology.

Dewan, P. and R. Choudhary (October 1992). "A High-Level and Flexible Framework for Implementing Multiuser User Interfaces." ACM Transactions on Information Systems **10**(4): 345-380.

Dewan, P., J. Grudin and E. Horovitz (2003). "Privacy and Sharing." \\t-pdewan1\public\access-twc.ppt.

Dewan, P. and A. Sharma (Sep 1999). An Experiment in Interoperating Heterogeneous Collaborative Systems. Proceedings of European Conference on Computer Supported Cooperative Work, Kluwer Academic Publishers.

Dewan, P. and H. Shen (Nov 1998). Flexible Meta Access-Control for Collaborative Applications. Proceedings of ACM Conference on Computer Supported Cooperative Work.

Droms, R. and W. Dyksen (1990). "Performance Measurements of the X Window System Communication Protocol." Software Practice and Experience **20**(S2): 119-136.

Edwards, K. (October 1994). Session Management for Collaborative Applications. Proceedings of the ACM Conference on Computer Supported Cooperative Work.
Engelbart, D. C. (September 1975). NLS Teleconferencing Features. Proceedings of Fall COMPCON.
Greenberg, S. and D. Marwood (October 1994). Real-Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. Proceedings of CSCW'94.
Greenhalgh, C. and S. Benford (September 1995). MASSIVE: A Collaborative Virtual Environment for Teleconferencing. ACM Transactions on Computer-Human Interaction.
Groove "Groove." www.groove.net.
Hill, R., T. Brinck, S. Rohall, J. Patterson and W. Wilner (June 1994). "The Rendezvous Architecture and Language for Constructing Multiuser Applications." ACM Transactions on Computer Human Interaction **1**(2).
James Begole, Mary Beth Rosson and C. A. Shaffer (1999). "Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems." ACM TOCHI **6**(2): 95-132.
Joseph, A. D., A. F. deLespinasse, J. A. Tauber, D. K. Gifford and M. F. Kaashoek (1995). Rover: A Toolkit for Mobile Information Access. Proceedings of the 15th Symposium on Operating System Principles.
Kistler, J. J. and M. Satyanaraynan (February 1992). "Disconnected Operation in the Coda File System." ACM Transactions on Computer Systems **10**(1): 3-25.
Krasner, G. E. and S. T. Pope (August/September 1988). "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." Journal of Object-Oriented Programming **1**(3): 26-49.
Kum, H.-C. M. and P. Dewan (2000). "Supporting Real-Time Collaboration Over Wide Area Networks." Proc. ACM Conference on Computer Supported Cooperative Work.
Lantz, K. A. (December 1986). An Experiment in Integrated Multimedia Conferencing. Proceedings of Conference on Computer-Supported Cooperative Work.
Li, S. F., Q. Stafford-Fraser and A. Hopper (March 2000). "Integrating Synchronous and Asynchronous Collaboration with Virtual Networking Computing " Proceedings of the First International Workshop on Intelligent Multimedia Computing and Networking, Atlantic City, USA **2**: 717-721.
Munson, J. and P. Dewan (June 1997). "Sync: A Java Framework for Mobile Collaborative Applications." IEEE Computer **30**(6): 59-66.
Munson, J. and P. Dewan (November 1996). A Concurrency Control Framework for Collaborative Systems. Proceedings of the ACM Conference on Computer Supported Cooperative Work.
NEC (2000). Windows 2000 Terminal Services Capacity and Scaling, NEC.
Nieh, J., S. J. Yang and N. Novik (2000). A Comparison of Thin-Client Computing Architectures, Columbia University.
Placeware "Placeware." www.placeware.com.
Placeware (2000). Placeware documents.
Reality, A. "Presence AR." www.advancedreality.com.
Roseman, M. and S. Greenberg (1996). "Building Real-Time Groupware with GroupKit, A Groupware Toolkit." ACM Transactions on Computer-Human Interaction **3**(1).
Roussev, V., P. Dewan and V. Jain (2000). Composable Collaboration Infrastructures based on Programming Patterns. Procedings of ACM Computer Supported Cooperative Work.
Shapiro, M., F. L. Fessant and P. Ferreira: (2001). The IceCube approach to the reconciliation of divergent replicas. PODC.
Shen, H. and P. Dewan (November 1992). Access Control for Collaborative Environments. Proceedings of the ACM Conference on Computer Supported Cooperative Work.
Sinnreich, H. and A. B. Johnston (2001). Internet Communication Using SIP. NY, Wiley.

Stefik, M., G. Foster, D. G. Bobrow, K. Kahn, S. Lanning and L. Suchman (January 1987). "Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings." CACM **30**(1): 32-47.

Sun, C. and C. Ellis (Nov 1998). Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. Proc. CSCW'98.

Terry, D. B., M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser (1995). Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. Proceedings of the 15th Symposium on Operating System Principles.

Webex "Webex." www.webex.com.

Wong, A. Y.-L. and M. I. Seltzer (2000). Evaluating Windows NT Terminal Server Performance. Usenix.

15 Index