DistView: Support for Building Efficient Collaborative Applications using Replicated Objects

Atul Prakash and Hyong Sop Shim
Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122, USA
Tel: 1-313-763-1585
E-mail: aprakash@eecs.umich.edu, hyongsop@engin.umich.edu

ABSTRACT

The ability to share synchronized views of interactions with an application is critical to supporting synchronous collaboration. This paper suggests a simple synchronous collaboration paradigm in which the sharing of the views of user/application interactions occurs at the window level within a multi-user, multi-window application. The paradigm is incorporated in a toolkit, DistView, that allows some of the application windows to be shared at a fine-level of granularity, while still allowing other application windows to be private. The toolkit is intended for supporting synchronous collaboration over wide-area networks. To keep bandwidth requirements and interactive response time low in such networks, DistView uses an *object-level replication* scheme, in which the application and interface objects that need to be shared among users are replicated. We discuss the design of DistView and present our preliminary experience with a prototype version of the system.

KEYWORDS: Groupware, multi-user interfaces, collaboration technology, shared windows, active objects, distributed objects, replicated objects, concurrency control.

INTRODUCTION

A great interest has developed in recent years in building collaboration tools that allow people to work with each other without the need for physical proximity [7, 17, 13, 18]. This interest has been partly motivated by the increasing availability of powerful interconnected workstations. The focus of this paper is on techniques for supporting synchronous collaboration among geographically separated users in a networked computing environment.

This paper describes DistView, a toolkit that we are developing to provide efficient and fault-tolerant collaboration support to object-oriented applications. Application developers may employ the DistView toolkit to convert existing object-oriented applications to collaboration-aware applications with minimal effort or to create new collaboration-aware applications from scratch.

DistView supports building of collaborative multi-window applications in which users can share some of their application windows with other users while still keeping other application windows private. Applications can be designed to be collaboration-aware so that several users can simultaneously interact with the application and with each other. The toolkit is intended for supporting synchronous collaboration over wide-area networks. To keep interactive response time low in such networks, DistView uses an *object replication* scheme, in which the application and interface objects that need to be shared among users are replicated. This paper describes the design concepts used in DistView in detail, an early version of which has been implemented on the NeXTSTEP operating system.

The rest of the paper is organized as follows. We first give an overview of a real-life collaborative application that helped formulate the initial goals of DistView. We then present the design goals for DistView. Next, we describe typical usage of a DistView-supported application. Then, we present the application model assumed by DistView. Next, we give an overview of the design approach used in DistView. After that, we describe the object replication and concurrency mechanisms used in DistView. We then provide some experimental results of the use of DistView. Next, we compare DistView with related work on groupware and multi-user toolkits. Finally, we summarize our conclusions and point out some interesting open issues.

A MOTIVATING EXAMPLE --- THE UARC SYSTEM

The primary objective of the Upper Atmospheric Research Collaboratory (UARC) project [8] is to provide space scientists with the means to effectively view and analyze data collected by various instruments located in Greenland. In the past, the scientists had to make annual trips to Greenland just to gather data. In order to eliminate the needs for these costly trips and to better support collaboration between the

CSCW 94- 10/94 Chapel Hill, NC, USA
© 1994 ACM 0-89791-689-1/94/0010..$3.50

scientists, our research team at the University of Michigan has developed the UARC system. Based on the client-server architecture, a UARC server gathers data from the instruments and broadcasts them to UARC clients which run at various sites around the world. The clients then graphically display the server data in various data windows (see Figure 1). A UARC client also includes a "chat window" through which scientists may communicate with the group by typing and sending their messages.

Over the past year, the UARC system has been extensively tested and used with success on several full-scale, real-life scientific experiments. However, the UARC system did not provide any support for synchronization of data display windows among the scientists. Each scientist had access to the same underlying data sets, but could be looking at different parts of the data sets or displaying the data in a different manner. A study of the usage of the UARC system showed that when having on-line discussions with their colleagues through the chat window, scientists often needed to describe the information displayed in their data windows [12] so that other interested scientists could set their windows to show the displays. The problem remained that this manual synchronization of displays was temporary; since the displays were independent, as soon as they were interacted with, they would go out of synchronization.

It appeared to us that synchronous collaboration among the scientists would be more effectively realized if the scientists could *share* their windows with other scientists. That is, upon observing something interesting in a window, a scientist would simply *export* the window and other scientists would simply *import* the window. Furthermore, it would be desirable if the displays in the exported and imported windows remain synchronized when the windows are interacted with.

GOALS FOR DISTVIEW

Based on our experience with UARC, we have identified the following goals for DistView to help support synchronous collaborations effectively:

1. *The sharing of application workspace should be allowed at the granularity of individual windows.*

   Experience with observing the use of the UARC system indicated that sharing the entire interface of an application would impose a severe screen real-estate problem on the users. Furthermore, scientists using the UARC system often had different goals and specializations, and thus were normally interested in looking at different data sets. It was therefore important that they retained the ability to have private application windows in a collaborative environment. A similar case arises in group editing when a user may wish to open a new window for looking at a private file. We thus felt that allowing the sharing of the application's interface at the granularity of windows will provide the necessary flexibility in collaboration as well as will make the interface intuitive to use for the users. This was especially important in the UARC project because users are space scientists, who are new to groupware technology. We could simply tell them that windows are similar to paper that you can pass around to other scientists and any interactions you do with them will be visible to other scientists.

2. *Users should be able interact simultaneously with the application.*

   In other words, DistView should support building of collaboration-aware applications. In the case of UARC, it would made the application quite unusable if only one scientist could interact with UARC at a time.

3. *Sharing should not lead to a substantial increase in interactive response times.*

   In an interactive application, response times are critical for effective use. Collaboration capability should be provided in such a manner that response times continue to be similar to those in a system without collaboration support.

4. *A synchronous collaboration system should perform adequately over a wide area network.*

   Because the users of a collaborative application may be scattered all over the world, the collaborative application should perform well over a wide-area network. More specifically, even if available network bandwidth becomes limited, the application should continue to function with good response times and low latencies.

5. *Developing applications using a toolkit should not involve extensive amount of coding over developing non-collaborative version of an application.*

   In other words, if one starts out by first building a non-collaborative application, adding DistView-provided collaborative capabilities to it should not involve extensive effort.

All the features above are not met by existing toolkits for building synchronous groupware systems, at least not unless application developers are willing to do extensive distributed systems and user-interface programming. Our goal in designing DistView was that the above requirements should be met with the application developer requiring as little knowledge as possible beyond that of developing non-collaborative applications.

EXAMPLE OF A DISTVIEW-BASED APPLICATION

We added capabilities to share windows to the UARC system described in the previous section using a preliminary version of DistView. In this section, we give an example of how added collaboration facilities are used in the system.

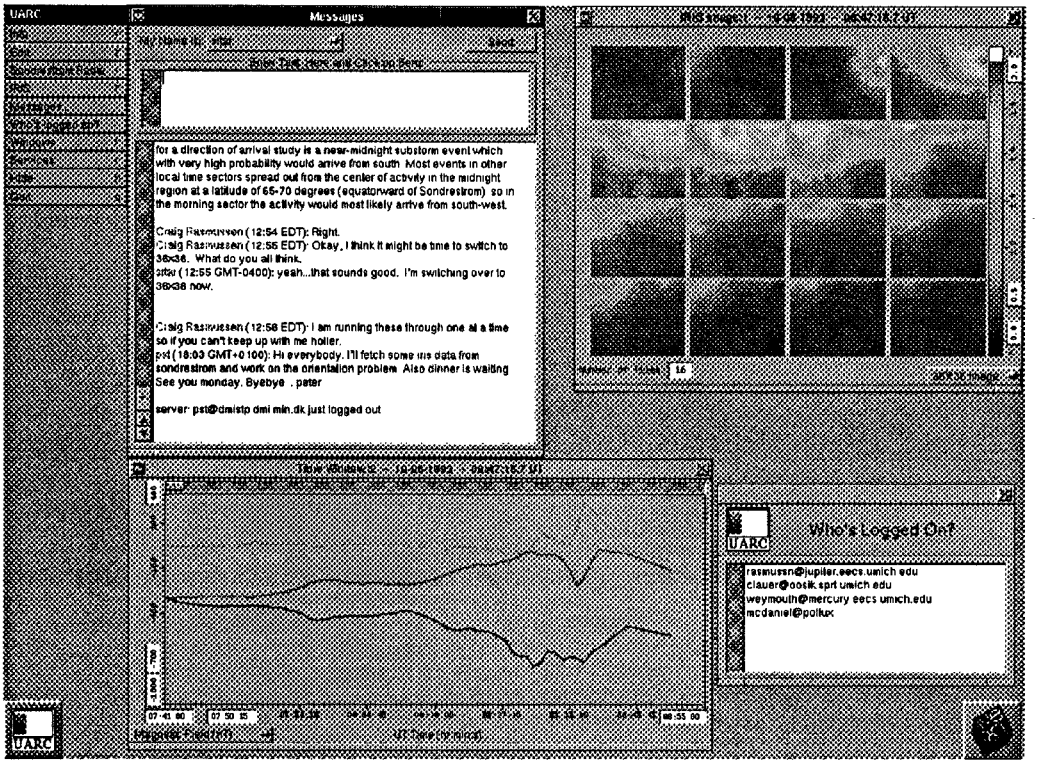
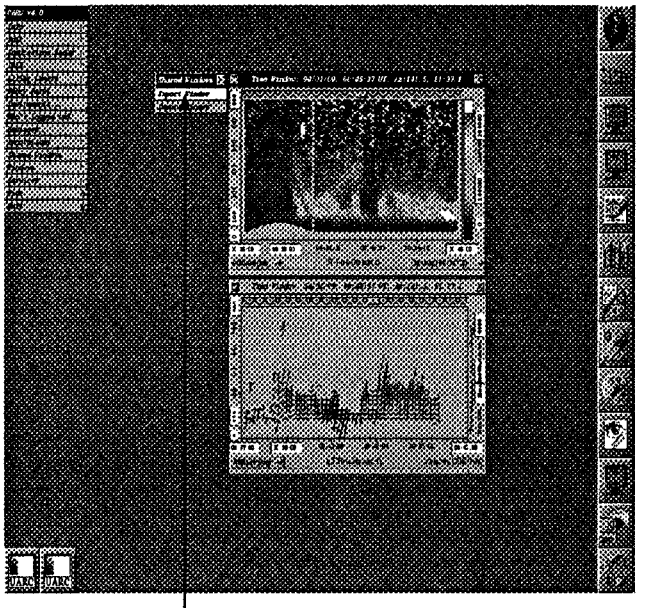


Figure 1: Interface provided to a user by the UARC system. Each user has independent control over what data sets they look at and the type of displays used for presenting the data sets. At the top left is a multi-user chat window to allow users to communicate with each other.

Figure 2 shows how a user exports a window, in which something interesting is observed, to the group. The user simply selects the window to be exported by clicking on the window and then selects the `Export Window` command from an available menu. The application also provides feedback to the user that the window has been exported by changing its background color (alternatively, its title could have been changed). Through the chat window, the user can inform other users about the newly exported window.

Figure 3 shows another user, who is running the UARC application, importing a window. The user first selects the `Import Window` command from the application menu. This results in popping up of a list of windows that have been previously exported. Each item on the list contains the exporter's login name and the title of the window that was exported. The user selects the window to be imported from this list.

Figure 4 shows the displays in the exported and imported windows after the import operation is complete (normally, these windows would be displayed on different consoles). The state of the imported window remains synchronized with the state of the corresponding exported window even if the windows are interacted with. For example, moving the scrollbar in one window causes corresponding movement of the scrollbar in the copies of the window. Changing the size



A user chooses the "Export Window" menu command to export the selected window (the window with the black title bar).

Figure 2: To export a window, user selects the window to be exported, and then clicks on "Export Window".

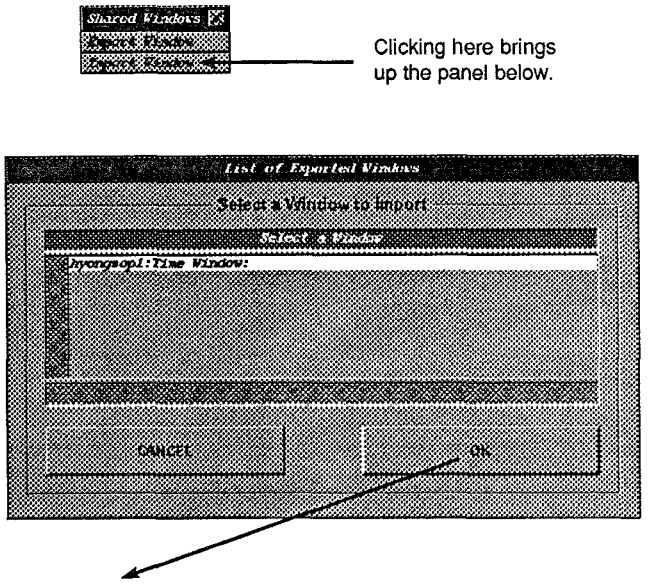


Figure 3: To import a window, user selects the "Import Window" from the menu and then selects the window to be imported.

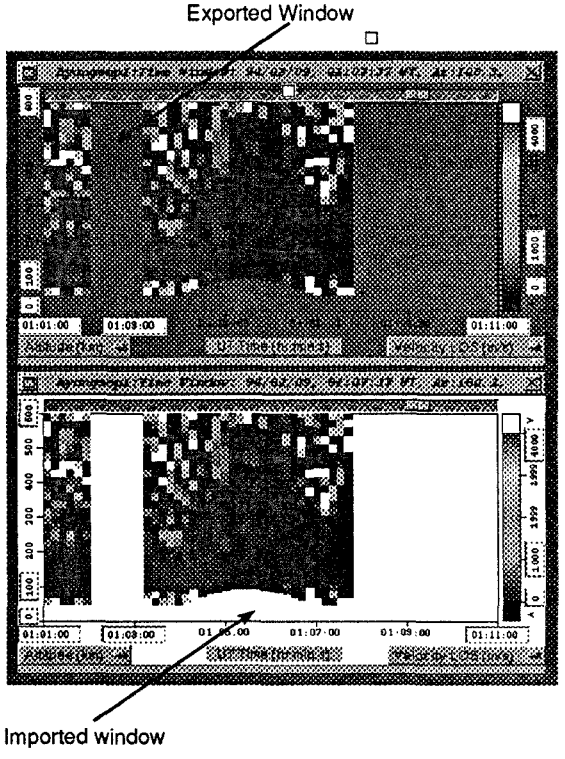


Figure 4: Corresponding imported window and exported windows with state synchronized. The UARC application is coded to set different background colors for imported and exported windows.

of the window also leads to the corresponding change in size in the copies of the window. Mouse movements within the window are also displayed in the copies so that users can use the mouse as a telepointer.

Note that the entire application interface does not have to be shared. All users can still have private application windows whose behavior is under their control. Displays in private application windows can get affected by an action on a shared window only if they are displaying the same data and an action on the shared window leads to modification of the common data. Users can independently iconify, move around, or close shared windows.

DistView attempts to replicate all the necessary state associated with a shared window so that imported windows can continue to be shared even if the exporter crashes.

DistView provides features so that application developers can provide desired level of concurrency control to ensure consistency of replicated data and consistency of displays in shared windows. The current version of the UARC system allows simultaneous user interactions with different windows but limits interactions with a shared window to one user at a time. If in some application, one wishes to allow simultaneous interactions in a shared window (e.g., as in shared editing), DistView provides mechanisms for application writers to safely program such behavior.

APPLICATION MODEL

DistView is designed for integration with object-oriented, window-based applications. Such an application consists of a number of *interface* and *application* objects. Interface objects correspond to widgets, such as buttons, menus, and scrollbars, which are commonly provided by user interface toolkits. Interface objects provide users with the means to interact with applications. Application objects, although not directly visible to end users, respond to user input on interface objects, invoke operations on interface objects, and maintain application data. The future trend in user interface toolkits is for them to be object-oriented. Several high-level X toolkits (e.g., InterViews and Fresco [11]), NeXTSTEP, and various Macintosh and PC-based toolkits already provide a library of standard interface objects whose behavior is known in advance.

We require that the application satisfies the *auto-update* property (i.e., there is a mechanism that keeps the state of the interface consistent with the state of application objects as their states change). In other words, the user interface should not allow a user to interact with an out-of-date application state, possibly leading to undesirable behavior. If the state of application objects changes, the change must be appropriately reflected in the interface. Ensuring this is not difficult and is usually satisfied by most interactive software anyway.

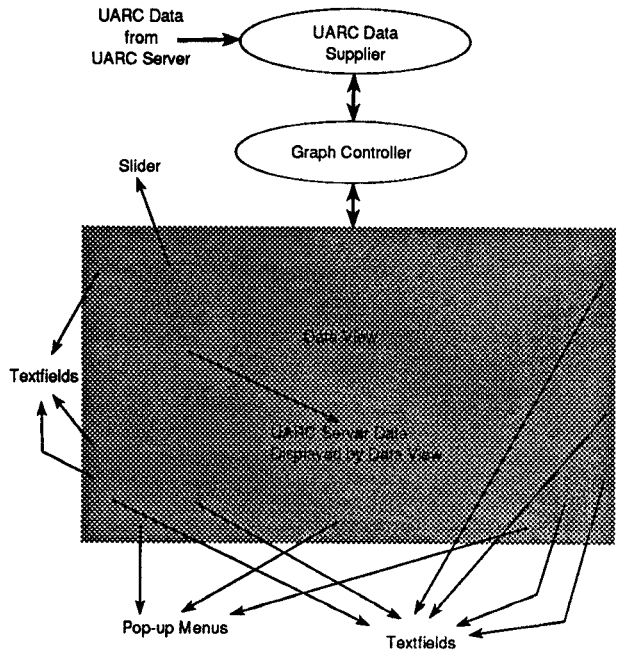


Figure 5: A UARC data window

A window itself is an interface object and consists of a number of other interface objects. The interface objects of the window form a hierarchy, with the window being the root. Also, these interface objects, including the window, may have references to other interface or application objects, to which messages are sent in response to user operations.

Figure 5 shows a typical UARC data window in a UARC client and the application objects that control the window's display. The pop-up menus, text-fields, and scrollbar (slider) in Figure 5 are examples of interface objects. The Graph Controller and the UARC Data Supplier are examples of application objects.

The window in the figure graphically displays the UARC data (currently, data from instruments such as radar in Greenland). When a scientist moves the knob of the scrollbar in order to view a different part of data, the window sends a message to the Graph Controller object, requesting the desired amount of data. The Graph Controller, in turn, sends a message to the Data Supplier, requesting the data. Going the other way, when new data arrives over the network, the Data Supplier sends a message containing the data to the Graph Controller, which then sends another message containing the data to the window so that the window can display the new data. The Graph Controller also invokes an operation on the scrollbar by sending it a message, so that the scrollbar may appropriately adjust its knob for the new data.

Each UARC data window has its own Graph Controller whereas all UARC clients have a single UARC Data Supplier. A scientist may select a different mode of display by using the provided pop-up menus. The scientist may also view different parts of the server data by entering desired ranges in the text-fields or by moving the knob on the scrollbar.

DESIGN APPROACH

In DistView, we have adopted the following design approach in order to meet the goals stated earlier.

Window-level sharing: Whenever appropriate, users should be able to share their application workspace with other users at the window-level rather than being required to share it in its entirety.

Support for concurrency control: Locking mechanisms are provided so that simultaneous interactions by users can be supported, without leading to undesirable or inconsistent results.

Replication: Interface objects and application objects which belong to shared windows are replicated in order to promote local processing.

Figure 6 shows the high-level components of DistView. When a window is *exported*, its name and the address of its exporting process are made available to other sites through the *Shared Window Server*, a central process known to all the sites that wish to share application windows. An *export window manager* is also created within the exporting process. The export window manager is responsible for transferring the state of the window to an importing process.

A user can get the list of windows available for import from the Shared Window server. When the user *imports* a window from the list, an *import window manager* is created within the importing process to handle the window's import. The import window manager sends a message to the exporting process, requesting the window's export manager to send the state of the window. The state sent results in replication of the window, including all its interface objects, within the address space of the importing application.

To transfer the state of its exported window, the export window manager recursively traverses down the hierarchy of the window, retrieves the type and state of each interface object it encounters, and sends the information over the network to the import window manager that has requested the window[1].

Upon receiving the stream, the import window manager retrieves the needed information on the selected window from the stream and creates a local copy of all the interface objects belonging to the window, initialized to the received states. Furthermore, if it finds that these newly created interface objects have references to other objects, those objects are also

[1]We require that all user-interface objects are designed to provide a standard mechanism to access and set their state.

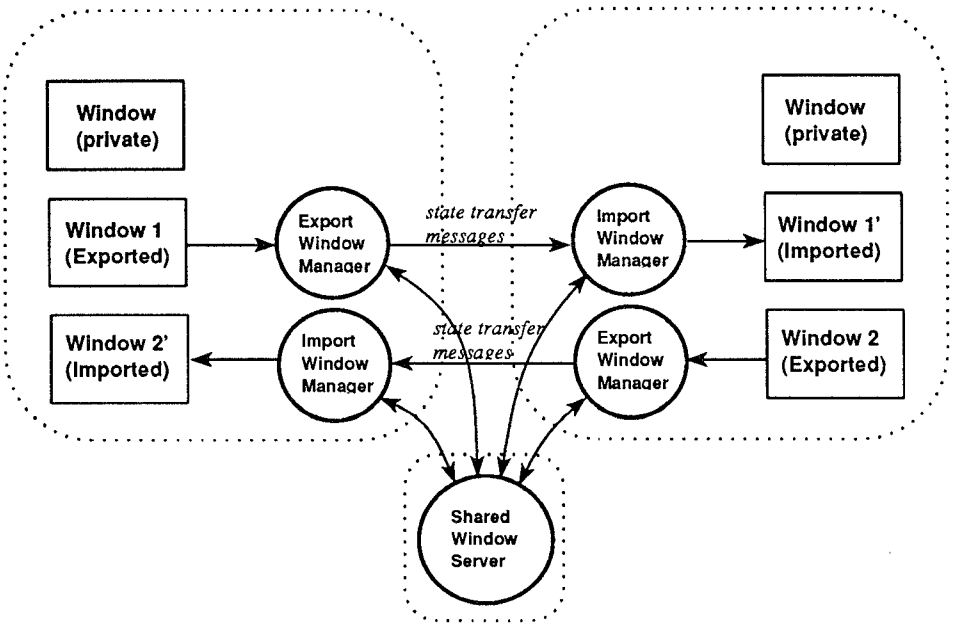


Figure 6: The application is participating in two separate collaboration sessions through Window 1 and Window 2. Note that users may have private windows that are not shared at the same time. The dotted boxes distinguish between different processes.

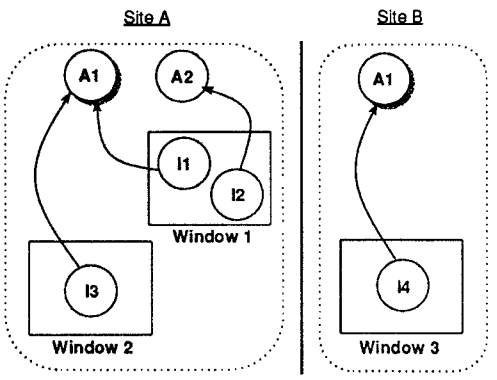


Figure 7: The state of the multi-user application at sites A and B before sharing of Window 1. A1 and A2 are application objects, and I1, I2, I3, and I4 are interface objects. Objects with shadows are replicated.

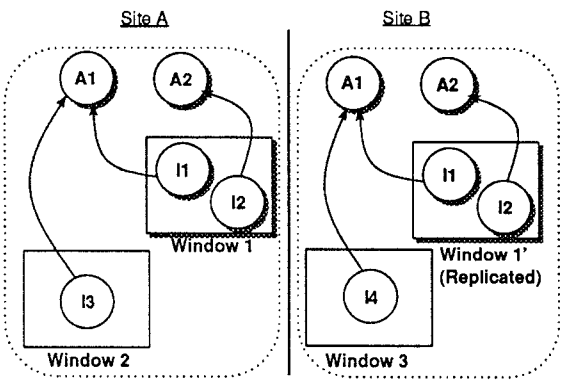


Figure 8: The state of the multi-user application at sites A and B after the user at site B imports Window 1.

replicated, unless the application writer has marked them as not suitable for replication (this issue is discussed further in the next section). The state of the exported window and that of the replica are synchronized from the time of import.

An example of the object replication scheme of DistView is shown in Figure 7 and Figure 8. The figures show the interface objects, and the applications objects they have references to, contained in application instances running at different sites A and B. Figure 7 shows the situation immediately prior to the import of Site A's Window 1 by site B. Figure 8 shows the situation immediately after Window 1 is imported by site B. Upon import of Window 1 at site B, a copy of all the interface objects of Window 1 is made at site B. A local copy of application object $A2$ is also made because this object is referenced by the interface object $I2$ within the exported window. However, a copy of the application object $A1$ is not made because a copy already existed at site B prior to the import of Window 1.

The next section describes in more detail how and when objects are replicated and mechanisms for maintaining their consistency.

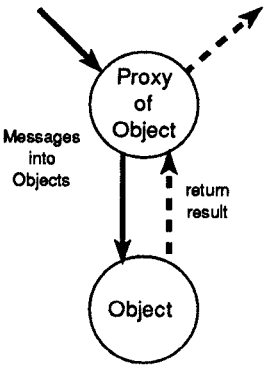


Figure 9: Outside world communicates with a proxy object. Proxy object is the only one that directly communicates with an application or interface object.

OBJECT REPLICATION AND SYNCHRONIZATION

Proxy objects

DistView synchronizes the states of shared windows so that they always maintain and display the same view to the users. The synchronization scheme used in DistView is to intercept the user actions on an interface object within a window and to broadcast them to the window's replicas, which then locally update their states. Interception of users' actions is done by using *proxy* objects. For every object in an application, there is a proxy object with the same external interface. The proxy object in this case is responsible for broadcasting the operation to other replicas as needed and also returning the results (See Figure 9).

Proxy object classes can be automatically synthesized by parsing the specifications of actual interface and application objects similar to the way remote procedure call stubs are generated from procedure call specifications. Calls or declarations that create regular application or interface objects are replaced by calls to create corresponding proxy objects (this can also be done automatically by a parser). Proxy objects in turn create the real application or interface objects and retain a reference to them.

Notice that proxy objects introduce a level of indirection in the system — all operations on an object must first go through its proxy object. The proxy object corresponding to the object is the only one that should carry out an operation on the object directly. At the expense of a minor performance penalty, use of indirection gives substantial flexibility since all the knowledge about dealing with replication can be in the proxy classes. A similar technique of introducing a level of indirection is used in Fresco, the object-oriented application programming interface currently under development for X, in order to providing uniform access to local and remote user interface objects [11].

Interface Object Replication

When a window is imported, DistView replicates all the interface objects of the window and the window object itself within the address space of the importing application. The state of an interface object transferred from an export window manager to an import window manager for correct replication includes its type (class), its type-specific internal state, its references to other objects, and its location within its parent window.

Application Object Replication

If an application object is replicated, all the replicas must have consistent behavior if the same sequence of operations is done on them. We say that an *application object* can be safely replicated if it is feasible to replicate its state and furthermore the replicas will behave identically at different sites when the same sequence of operations are carried out on them.

Not all application objects can be safely replicated. For instance, if some object reads a file on the local file system, it will behave differently at sites with different file systems. In that case, we require that the application designer designate the object as not being suitable for replication. Each application object is required to provide a method *do_replicate()* that returns *true* if the object is safely replicatable and its replication upon sharing is desired, and *false* otherwise. Application objects that are directly or indirectly referenced by the interface objects in the window and whose *do_replicate()* method returns true are the only ones that are replicated. If an object is referenced but cannot be replicated, a proxy is still created. This proxy can forward any calls on it to the proxy at the site where the object actually resides (via techniques similar to those used in implementing remote procedure calls) and return any results back to the caller.

Replica groups

An object and its replicas form a *replica group*. A replica group is assigned a globally unique group identifier by DistView. Group identifiers are generated by the shared window server as needed by an application. If a reference to an object needs to be passed from one address space to another, it is first converted to the object's globally unique identifier by the sender and converted back to the corresponding local object reference by the receiver.

An *object manager* is created as part of each site's application at start-up time and remains within the application until the application terminates. The application object manager maintains the mapping between globally unique identifiers and the pointers to the local proxies of the objects.

A method invocation by a replica group A on replica group B has to be carefully and efficiently implemented. Following requirements must be met by algorithms to implement method invocations among replicas, in order to ensure consistency of replicas:

- The algorithms should ensure that all the members of group A receive the same return result from the call.
- All members of group B should execute the call.
- The algorithms used should take into account the possibility that the members of replica group A and group B may differ in count and may not be located at the same sites. For example, replica group B may consist of only a single object because the object is not replicatable.
- It is desirable to take the possibility of failures into account.

The algorithms used are incorporated in the proxies. Thus, application developers do not need to be aware of these algorithms. Full discussion of the algorithms to accomplish the above is beyond the scope of this paper. For one solution to the problem, see the algorithms used in the Arjuna system [14]. As part of the DistView project, we are currently attempting to develop a generic replicated object service that improves on the algorithms used in the Arjuna system. The solution is facilitated by the use of lock objects discussed in the section below.

Lock objects

In general, some way must be provided to prevent members of a replica group from becoming inconsistent because it is possible for two simultaneous users' actions to be received and executed in different orders at different sites. One solution to the problem is to use atomic broadcast protocols to ensure that all broadcasts are received in the same sequence at all sites [4, 5]. However, in this case, even the sender of a broadcast has to wait until it receives its own message from the network before it can execute the message. In fact, it may receive other sites' messages prior to receiving its own message due to atomic ordering requirements. This situation is clearly undesirable because it leads to poor interactive response-times. A potentially more serious situation is that it could also lead to erroneous results because operations received from other sites could have been done on the application's state between the time the operation was generated and the time it was actually carried out.

In DistView, we use locking techniques similar to those used in DistEdit to prevent such inconsistencies. The key idea is that all user operations must acquire appropriate locks to ensure that interface and application objects, when updated concurrently, lead to correct results and consistent replicas.

Using locks has the advantage that, once the locks are acquired, operations on objects can be done locally first and then broadcast, giving good interactive response times. Different users may hold different locks, so concurrent actions are still supported.

Browsing operations normally do not need to acquire any locks on application objects, and thus can be done in parallel with update operations initiated at other sites. The interface-update module (see the Application Model subsection) will ensure that the interfaces of shared windows eventually become consistent as soon as any pending update operations complete.

Locks in DistView are provided by *lock objects*. A lock object provides a method, *lock()*, which returns 1 if a lock can be acquired and else it returns 0, and a method *unlock()* to release the lock. Application designers can provide their own application-specific object classes to implement more sophisticated locking schemes, For instance, to allow simultaneous editing on a text user interface object as in the DistEdit toolkit [9], application designers can provide a text-specific object that maintains a lock table containing portions of text that are locked. DistView-provided lock objects would then be used only to control access to the lock table, rather than to the text object itself.

An alternative strategy would have been for proxy objects to automatically lock their application object when operations were done on them. However, this strategy is not suitable for several reasons. First, in an application, there may be a large number of application objects (e.g., items on a linked list). One user-level operation may need to access and update a large number of objects. If locks were acquired dynamically as objects were accessed, there was a danger that some lock requests could fail after some objects were already updated. Since interactive applications typically do not provide support for unrolling partially done actions, as in transaction mechanisms, such a situation could leave the application in an undesirable state. Second, even if all object locks could be acquired at the beginning of the user's operation, acquiring a large number of locks could mean significant network delays. And, finally, keeping lock objects independent of application objects is a more flexible scheme. Application designers could start out by using a single lock for the entire application (i.e., a floor-control based policy) and then progressively add more concurrency by using multiple locks — with each lock responsible for controlling access to fewer and fewer objects.

If acquiring or releasing locks requires going over the network, users could still perceive substantial increase in interactive response times because each user's operation may involve acquiring some locks, doing the operation, and then releasing the locks. In fact, if in a group session, only one user is interacting with the application, the same overheads would occur. Such a situation is clearly undesirable.

We can make acquisition and release of locks more efficient as follows. First, lock objects, like any other object, have a global reference that can be passed around between sites. Each site that has a global reference to a lock object should

create a local copy of the lock object. The site which has the lock has a *token*. Only one site can have the token at a time. When a site with the lock releases the lock, it is treated as a *hint* that the lock is no longer needed. The site still retains the token, but marks it as *available* for other users. If the same site wishes to relock the object, it can be done immediately without going over the network by simply marking the token as *unavailable*. If another site wishes to acquire the lock, it sends a message out to the group requesting the lock. The sites without the token ignore the message. The site with the token transfers the token if its token is marked *available*, else it denies the lock request. This scheme requires that one site be initialized to always hold the token (which may be marked available to indicate that no lock is held). If the site requesting the lock receives no response to its message within a certain timeout period, a suitable token recovery protocol can be run to isolate any unreachable sites from the group and reintroduce the token.

The performance impact of the above scheme is that network latencies in acquiring locks occur only if the lock has to be acquired from some other site. If only one user is repeatedly acquiring and releasing locks, there is no network latency except for the first lock request. We believe that this is likely to be an efficient strategy in practice, especially if one user is doing most of the interactions and others are simply watching.

EXPERIENCE WITH DISTVIEW

A preliminary prototype of DistView has been implemented and incorporated in the UARC application and several other smaller applications. Because the UARC application is implemented in the NeXTSTEP environment, DistView is also implemented in the same environment. Below, we discuss our experience with incorporating DistView in UARC.

Effort to Incorporate DistView in Applications

The shared window server of DistView runs as a separate process from the applications that incorporate DistView. The code for export and import window managers and object managers is also provided by DistView. As such, these components of DistView contribute little to the overall cost of incorporating DistView into existing applications. In the case of the UARC system consisting of about 7000 lines of code, it required adding 488 lines of code to allow interfacing with the current version of DistView. Most of this code is for acquiring and storing state information of application-specific objects and, in our experience, is not difficult to write for most objects. A small amount of code was needed to provide locks on shared windows. The state transfer for interface objects is handled by DistView. DistView uses primitives provided by NeXTSTEP to inquire about window hierarchies and states of window objects in a hierarchy. DistView creates a similar window hierarchy at the importing site. Overall, we feel that little effort is required to use DistView provided the application is written to satisfy the DistView's application model.

Imported By	Average Time Taken
Same UARC Client	1.48 sec.
Different UARC Clients	1.78 sec.

Table 1: The average time taken to import a window by the same UARC client which exported the window and by a different UARC client

Performance

In order to measure the performance of DistView, we ran three UARC clients simultaneously on three different NeXT workstations connected via an Ethernet. A window similar to Figure 5 was exported by the UARC client at one of the workstations. The UARC clients at the other workstations imported the window. The window was also imported by the same UARC client that exported the window.

Table 1 shows the average times taken to import a window, by the same UARC client which had exported the window and by other UARC clients. This compares the cost of state transfer within the same process on one machine and between processes across machines. The table indicates that there is little difference in the import times between the two cases. This result indicates that the amount of information required to be transferred to replicate even a rich window is not large.

Table 2 shows the average times taken to process user-actions in a collaboration session through an exported window when there is no window sharing. It also shows the additional processing time required when window sharing is present. Results indicate that the processing times of user-actions on the exported window does not significantly increase when the window is shared. The results may be attributed to the fact that all operations are done locally first before being broadcast.

Notice that for the operations in Table 2, several seconds are required to process a user operation even when there is no sharing. This is because those operations typically require the application to repaint the entire window. Because the window is quite rich, repainting the window takes substantial processing time. This also indicates that a shared-X style strategy of running only one instance of the application that controls multiple identical windows would probably have increased the response time substantially.

We have not yet systematically measured the performance of the system over long-haul networks. However, actual usage of the new UARC system between sites in California, Michigan, and Denmark indicates that the sharing of windows works sufficiently efficiently in such environments that any additional response time is not noticeable.

User Actions On	Avg. Redraw Time	Added Time w/ 1 Member	Added Time w/ 2 Members
Textfields	4.45 sec.	0.030 sec.	0.074 sec.
Slider	5.62 sec.	0.036 sec.	0.071 sec.
Pop-up Menus	5.61 sec.	0.032 sec.	0.063 sec.
Window (Resize)	3.54 sec.	0.018 sec.	0.028 sec.

Table 2: Times taken to process various user-actions in a UARC collaboration session with no member, one member, and two members

RELATED WORK

Different systems provide sharing of views at different levels of granularity. For instance, commercial products Timbaktu for Macintosh and ScreenCast on the NeXTs provide full screen-level sharing of arbitrary, unmodified applications. The disadvantage here is that users cannot easily do any private work and their entire screen is taken over by another user. Furthermore, applications are collaboration-unaware — simultaneous interaction with an application is not supported. Security is also potentially compromised since applications run with owner's privileges, though the risk may be acceptable in a synchronous environment.

Various systems allow a single-user application to send the same output to multiple consoles [1, 2, 3]. In these systems, usually based on X windows, a pseudo-server intercepts low-level protocol messages between the X server and desired X clients and multicasts the messages to X servers that are controlling other consoles. These systems have similar advantages and disadvantages as that in screen sharing, except that users can also run private applications on their screen. The type of sharing supported is still limited in the sense that the entire interface of an application must be shared — it is difficult to support an application in which some windows are private and some are shared. Furthermore, like screen sharing programs, applications are collaboration-unaware — simultaneous input must normally be prevented through some floor-control mechanism. From a performance point of view, this approach places a heavier demand on the network bandwidth, and performance is potentially inadequate if the interfaces of applications are frequently updated (e.g., when scrolling or repainting a window) because all screen updates are broadcast.

DistView is closer in goals to systems such as Suite [6], Rendezvous [15], and Groupkit [16] that support collaboration-aware applications. A major distinction from these systems is that DistView provides direct support for replicating objects to address performance and fault-tolerance concerns in collaboration over wide-area networks. We have found it desirable to replicate application objects (data) because we have usually found that read accesses to data are much more frequent than write accesses. For instance, if a window is resized, moved, or exposed, or scroll operations are done on the window, most window systems typically require the application to redraw the contents of the window. In particular, in the UARC project which supports collaboration over wide-area networks, it simply would not have been feasible to use an architecture with a centralized data repository. Fault-tolerance is also, of course, a potential advantage of replication. Furthermore, the application writer is given mechanisms to handle concurrency control through the use of locks — an issue which is typically not adequately addressed in existing toolkits.

The replication approach used in DistView builds on our work on the DistEdit toolkit [9, 10], in which the underlying data objects representing editors' text buffers were replicated. DistView goes further because arbitrary interfaces can also be replicated, providing not just consistent but identical displays of data in windows that are shared. Furthermore, DistView is designed to support a much more general class of applications.

DistView is also different from above systems in the granularity of sharing it provides. Systems such as Suite and Rendezvous can potentially provide a lot of flexibility in view sharing. However, then selecting the right amount of flexibility can be difficult for both system designers and end-users. In DistView, the unit of sharing is a physical interface object, a window on the user's screen. If a window is shared, a user can assume that all users will have identical views displayed in that window. Users can also assume that their other windows are private unless the (shared) application objects being displayed in their private windows get modified by other users.

Note that DistView does not prevent application designers from implementing more flexible forms of sharing. Recall that a private window can get affected by operations on a shared window, provided those operations affect the state of an application object being displayed in the private window. This mechanism can be exploited, if desired, to provide more flexible kinds of sharing by storing some of the user interface attributes (e.g., color of text) in application objects. If a particular private window uses those application objects, doing an operation on a shared window (e.g., changing the color of text) can be made to lead to an update of a private window (e.g., changing the text there to the same color).

Another way to provide slightly different looks among copies of a shared window is for the application to bypass the proxies

and directly modify the state of the underlying user interface objects. Casual use of this mechanism is not usually recommended for obvious reasons. However, it was used effectively in the UARC system to provide a different background color for imported and exported windows (as shown in Figure 4).

The mechanisms for maintaining consistency of views are also different in DistView from the above systems. In Rendezvous, programmers have to define constraints between instance variables in different views and a constraint propagation system is used to ensure consistency. In Suite, couplings have to be defined between interface variables so that their views remain consistent. DistView, because of its simpler sharing philosophy, requires no such identification of instance variables. At the time of importing a window, it directly queries the current state of the application to determine exactly what needs to be replicated. Thus, we expect that DistView will more easily handle sophisticated interfaces that may be difficult to characterize in terms of a few instance variables.

Independent of our work on DistView, Tou, Berson, and Estrin have also proposed use of object-level replication schemes for implementing collaborative applications using Object World [19]. Object World requires similar functionality from application and interface objects as in DistView and also has the goal of easing construction of collaborative applications. However, there are also some differences. One difference is that in the design of DistView, one of our primary goals is to ensure good performance. DistView provides locking algorithms and synchronization algorithms that will ensure good response times and consistency in interactive applications even in the presence of collaboration. Object World at present appears to be primarily meant for sharing within a LAN environment. It currently does not guarantee consistency among objects replicated across LANs and relies on users to correct inconsistencies among replicas when inconsistencies are detected. Another difference is that DistView provides explicit support for window-level sharing for arbitrary interfaces whereas Object World uses a strong sharing model in which the entire application's state is shared.

CONCLUSIONS AND OPEN ISSUES

This paper has presented DistView, a toolkit currently under development to support building of collaboration-aware applications. DistView allows development of applications in which users can share a subset of application windows with other users.

DistView uses a replicated object approach that is designed to ensure good interactive response times and to keep network bandwidth requirements low. Efficient locking primitives are provided to users for concurrency control at the desired level of granularity. Experience with an early version of DistView indicates that good response times can indeed be obtained in collaborative work. The first version of DistView was meant primarily to quickly add shared window capability to the UARC system. For instance, proxy classes were manually created and the support for replicating objects is somewhat kludgy. A new version of the DistView system in the form of a toolkit is currently under development. We are also considering developing a version for the X window system after selecting a suitable object- oriented user interface toolkit.

The design decision to do sharing at the window-level raised some interesting issues. In particular, one issue is that of the life-time of shared windows. The question is when does a shared window become unavailable for use by the users? One option is to destroy a shared window at all sites when the original exporter decides to destroy the window. A second option is when no one is using it. Yet another option is to allow an exported window to live even when no one is using it. The last option is interesting because it allows a simple form of ubiquitous computing [20]: a user could export all his application windows, go to another workstation, start up the same application, and import all the previously exported windows. In the future, we plan to explore some of these issues further.

ACKNOWLEDGEMENTS

We hank Nelson Manohar, Amit Mathur, Ramani Penmetsa, Susan McDaniel, Terry Weymouth, Craig Rasmussen, Gary Olson, Daniel Atkins, Robert Clauer and other members of the UARC team for their constructive comments and suggestions. We also thank the reviewers for excellent feedback. Support for this work has been provided by the National Science Foundation through the cooperative agreement IRI-9216848.

REFERENCES

1. H. M. Adbel-Wahab and M. A. Feit. XTV: A framework for sharing X window clients in remote synchronous collaboration. In *Proceedings, IEEE Tricomm '91: Communications for Distributed Applications and Systems*, April 1991.
2. S.R. Ahuja, J.R. Ensor, D.N. Horn, and S.E. Lucco. The Rapport Multimedia Conferencing System: A Software Overview. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 52–58, March 1988.
3. J.E. Baleschwieler, T. Gutekunst, and B. Plattner. A Survey of X Protocol Multiplexors. *ACM Computer Communication Review*, 23(2):13–22, 1993.
4. K.P. Birman, A. Schiper, and P. Stephenson. Lightweight casual and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.
5. J.M. Chang and N.F. Maxemchuck. Reliable broadcast protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, Aug. 1984.
6. P. Dewan. Flexible User Interface Coupling in Collaborative Systems. In *Proceedings of the ACM*

CHI'91 Conference on Human Factors in Computing Systems, pages 41–48, April 1991.

7. C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some Issues and Experiences. *Communications of the ACM*, pages 38–51, January 1991.
8. R. Clauer et. al. UARC: A prototype upper atmostpheric research collaboratory. *EOS Trans. American Geophys. Union*, 267(74), 1993.
9. M. Knister and A. Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, October 1990.
10. M. Knister and A. Prakash. Issues in the Design of a Toolkit for Supporting Multiple Group Editors. *Computing Systems – The Journal of the Usenix Association*, 6(2):135–166, Spring 1993.
11. M. Linton and C. Price. Building distributed user interfaces with Fresco. In *Proceedings of the 7th X Technical Conference*, pages 77–87, January 1993.
12. S. McDaniel, G. Olson, and J. Olson. Methods in Search of Methodology-Combining HCI and Object Orientation. In *CHI '94 Proceedings)*, pages 145–151, 1994.
13. C.M. Neuwirth, D.S. Kaufer, R. Chandhok, and J.H. Morris. Issues in the design of computer support for co-authoring and commenting. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 183–195, Los Angeles, California, October 1990.
14. G.D. Parrington. Reliable distributed programming in C++: the Arjuna approach. In *Proceedings of USENIX/C++ Conference*, pages 37–50, April 1990.
15. J.F. Patterson, R.D. Hill, S.L. Rohall, and W.S. Meeks. Rendezvous: An architecture for synchronous multi-user applications. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 317–328, Los Angeles, California, October 1990.
16. M. Roseman and S. Greenberg. GroupKit: A groupware toolkit for building real-time conferencing appliations. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 43–50, Toronto, Canada, October 1992.
17. D. Bogia S. Kaplan, W. Tolone and C. Bignoli. Flexible, active support for collaborative work with ConversationBuilder. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, Toronto, Canada, October 1992.
18. M. Stefik, G. Foster, D.G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the Chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, Jan. 1987.
19. I. Tou, S. Berson, and G. Estrin. Prototyping synchronous group applications. *IEEE Computer*, 27(4):48–56, May 1994.
20. Mark Weiser. The Computer for the 21st Century. *Scientific American*, September 1991.