W Prinz, M Jarke, Y. Rogers, K. Schmidt, and V Wulf (eds.), *Proceedings of the Seventh European Conference on Computer-Supported Cooperative Work, 16-20 September 2001, Bonn, Germany*, pp 99-118

Flexible Support for Application-Sharing Architecture

Goopeel Chung & Prasun Dewan
Department of Computer Science
University of North Carolina at Chapel Hill, USA
{chungg,dewan}@cs.unc.edu

Abstract. Current application-sharing systems support a single architecture for all collaborations, though different systems support different architectures We have developed a system that supports a wide range of architectural mappings, which include, to the best of our knowledge, all of the existing architectures defined so far including the centralized, replicated, and hybrid architectures Instead of being bound to a specific I/O protocol such as the X protocol, it is based on a simple abstract I/O protocol to which specific I/O protocols must be mapped by client-supplied code. We have used the system to perform experiments that compare the performance of the centralized and replicated architectures. Our experiments show that the choice of the architecture depends on the computers used by the collaborators, the speed of the connections between the computers, and the cost of the operations performed by the shared application Under some conditions the centralized architecture gives better performance, while under others the replicated architecture gives better performance Our results contradict the popular belief that the replicated architecture always gives better performance, and show the need for supporting both architectures in a system.

Introduction

A variety of systems have been developed to allow multiple users to share an application, that is, concurrently provide input to and view output of an application (Dewan, 1993). These systems may also provide other collaboration functions such as concurrency control, access control, undo, and merging (Dewan, Choudhary et al., 1994). However, these functions go beyond the basic capability

to share the I/O of applications among multiple users and, thus, will be ignored in this paper. Several, if not most, systems allow sharing of applications without any support for such high-level functions, or with support for very primitive forms of the functions.

Application-sharing systems differ in the architectures they offer to share the application. In general, users' actions are processed by multiple application layers such as the kernel, window system, toolkit, user-interface management systems, and application semantics. An example of the actual layers is given later. Each layer receives output from the layer above it and input from the layer below it. Different architectures differ in:

- the way in which they replicate these layers. Some systems completely replicate the application (Abdel-Wahab, Kim et al., 1999; Lantz, 1986; Chabert, Grossman et al., 1998), some completely centralize it (Ishii and Ohkubo; Li, Stafford-Fraser et al., 2000), some centralize the window client but replicate the window system (Abdel-Wahab and Feit, 1991), while still others centralize the top-level (semantics) layer but replicate the layers below it (Dewan and Choudhary, 1992; Hill, Brinck et al., 1994).
- the layer whose input and output is shared by the users. Architectures have been developed to support sharing of frame-buffer events (Li, Stafford-Fraser et al., 2000), window events(Abdel-Wahab and Feit, 1991; Lantz, 1986), and higher-level events (Dewan and Choudhary, 1992; Hill, Brinck et al., 1994).

These architectures can be described using the generalized zipper model (Dewan, 1998). The model assumes that if a layer is replicated, all layers below it are also replicated. Differences in architectures can result from differences in the top-most layer that is replicated. The higher this layer, the more the replication degree of the architecture. This degree is thus increased by "opening the zipper" until we reach the fully replicated architecture. Figure 1 and 2 show the zipper opened to different degrees for the same set of layers.

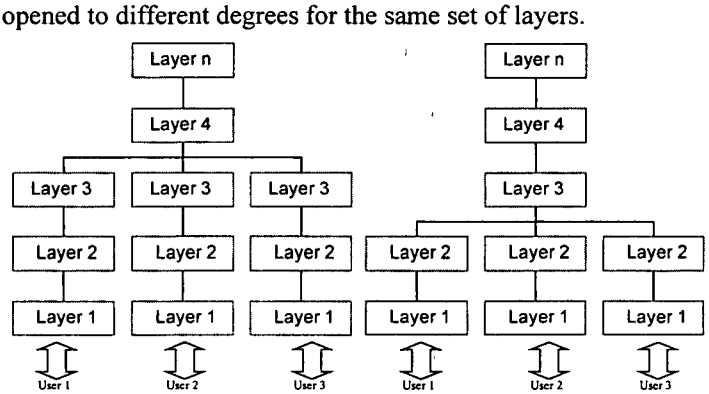

Figure 1 Varying the Sharing and Replication Degree.

Differences in architecture can also result from the level of the layer whose I/O is shared. The higher the level of this layer, the lower the sharing degree supported by the architecture. In the left architectures of Figure 1 and Figure 2, the I/O of layer 4 is shared, while in the right architectures of the two figures, the I/O of layer 3 is shared. Sharing of I/O of layer `n` implies sharing of the I/O of layer `n+1`. Thus, the right architectures offer a higher sharing degree.

The centralized and replicated architectures differ in how they support sharing of the I/O of a layer. If the layer is centralized, then its output is broadcast to all layers below it (Figure 1). If the layer is replicated, then it receives the input of all of the replicas of the layer below it. For instance, in the left architecture of Figure 2, the middle replica of layer 4 receives the input of the replicas of all layer 3. To avoid cluttering the diagram, we have not shown that other replicas of layer 4 also receive the same input to ensure that they are synchronized with each other.

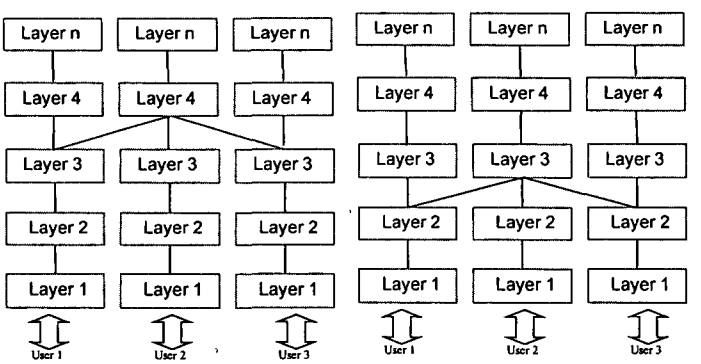

Figure 2. Varying the Sharing Degree in Replicated Architectures.

There are pros and cons of choosing a particular sharing degree (Dewan and Choudhary, 1995; Stefik, Bobrow et al., 1987). In a very tightly coupled collaboration, users may wish sharing of I/O at the frame-buffer level so that all users see a common screen. In another collaboration, users may wish sharing of I/O at the window level so that they can share some windows but not others. In a collaboration that is even more loosely coupled, they may wish sharing of I/O at the toolkit level so that they can scroll to different parts of a window.

There are also pros and cons of replicating the layer whose I/O is shared. The main advantage is that it requires lower communication bandwidth because input, rather than output, must be distributed among the application-sharing site. Thus, response times are less sensitive to network latency. This is important when the collaborators are connected by slow networks.

However, there are many disadvantages of replicating a layer. Performing computationally-expensive operation multiple times in different replicas (a) wastes computing resources, (b) in an application-sharing session involving lightweight computers (such as mobile computers), does not allow this processing

to be offloaded to a heavyweight computer connected by a fast network to the lightweight computers, (c) creates bottlenecks when the operation accesses a shared (non-replicated) resource such as a file (since all replicas tend to access the resource at the same time); and (d) causes undesired semantics when the operation is a non-idempotent action such as sending a mail message. Moreover, good response times can be achieved only if input events are given to the local site first, and then distributed to other sites. This can result in different copies of the application receiving different sequences of input events, though some application-specific techniques such as operation transformation (Sun and Ellis, 1998) can be used to correctly synchronize the replicas.

Thus, the ideal architecture of an application-sharing session depends on several factors such as the layers and semantics of the shared application, the network connections and computers of the users, and the degree of sharing desired. Yet, each of the existing application-sharing architectures supports a single architecture for all collaboration sessions.

In this paper, we describe an infrastructure that supports a whole design space of application-sharing architectures, which includes the architectures of all application-sharing systems known to us. Instead of being bound to a specific layer, such as the VNC frame buffer (Richardson, Stafford-Fraser et al., 1998) or the X window system (Scheifler and Gettys, 1983), it is bound to an abstract inter-layer communication protocol to which specific inter-layer protocols can be mapped. A client of the infrastructure is responsible for translating between the specific and abstract communication protocols. The infrastructure is implemented in Java.

The infrastructure we describe does not allow the architecture to dynamically change during a collaboration session. But it does allow different collaboration sessions to use different architectures based on the application and other factors mentioned above.

By building such an infrastructure we make three contributions:

(1) We provide an application-sharing system that is not bound to a specific layer.

(2) We allow the architecture of an application-sharing session to be tailored to the application, users, computers, networks, and the task.

(3) We provide a basis for comparing the performance of different architectures under the same conditions. Such a comparison cannot currently be done because different architectures are implemented by different systems, which have differences other than the architecture supported. Architecture comparisons are much more meaningful when the compared architectures are created by configuring a single system.

The rest of the paper is organized as follows. We first describe the inter-layer communication protocol and the role of the translator. Next, we formally identify the range of architectures we can support. We then describe how our

infrastructure supports this range. The next two sections describe our experience with writing translators for existing systems and the results of experiments we have done comparing the performance of different architectures. Finally, we present conclusions and directions for future work.

Layer-Independent Sharing

Current application-sharing infrastructures are bound to I/O protocols defined by specific user-interface layers. They intercept the input and output of an application using this protocol, distributing the input of all users to the application and the output of the application to all users. The key to developing a layer-independent application-sharing protocol is to recognize that this distribution task is independent of the specifics of an I/O protocol. Thus, an application-sharing infrastructure can be defined in terms of an abstract I/O protocol. If the specific I/O protocol defined by some layer can be translated to this abstract protocol, then the infrastructure can support automatic sharing of the layers above it, as shown in Figure 3.

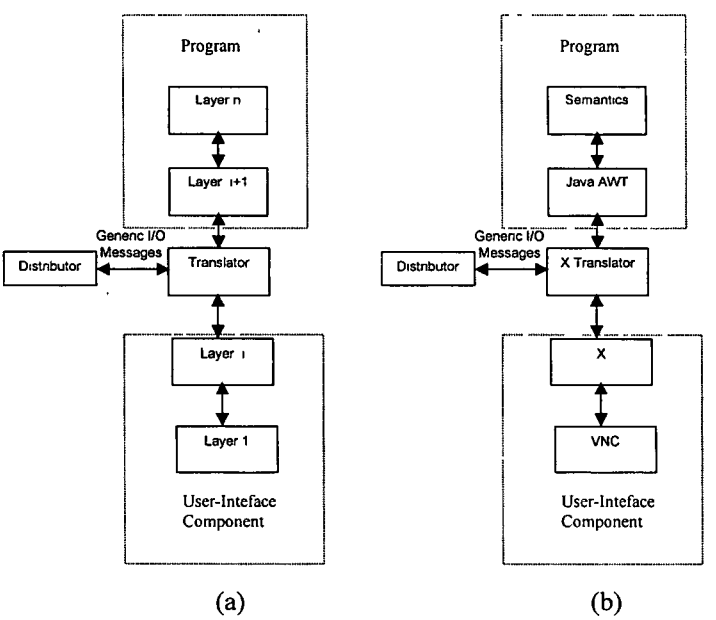

Figure 3. Translating and Distributing Messages.

If an application has `n` layers, and if the I/O protocol defined by the layer at level `i` is translated to the abstract protocol, then layers `1..i` are referred to as the user-interface component, and layers `i+1..n` are referred to as the program component (Figure 3(a)). A layer-specific translator module translates between

the abstract and concrete module while a layer-independent distributor is responsible for the actual distribution task. Thus, for each user-interface component, we build, not a complete application sharing system, but simply a module translating between the generic I/O messages understood by the distributor and the specific I/O messages understood by the component. For example, for X, we build a module translating between the X and distributor protocols (Figure 3(b)).

As we shall see later, an instance of the distributor and translator is created for each computer participating in a collaborative session. However, we will sometimes refer to all the translators (distributors) involved in supporting the sharing of an application as a single translator (distributor).

The idea of basing collaboration support on a generic protocol to which specific protocols are converted has been used earlier in the design of the DistEdit (Krasner, G. E. and S. T. Pope, 1988) collaborative replicated editor. DistEdit allows users editing a common document to use different editors. It defines an abstract editor protocol to which specific editor protocols must be converted. The DistEdit protocol defines only input editor events because it is designed to support only replicated editors. Our protocol defines arbitrary input and output events because it is designed to support arbitrary centralized and replicated applications.

The abstract I/O protocol is defined in Java and consists of the following output call:

- `output(Object` *object*`)` sends an arbitrary object specifying an output value.

and the following input call:

- `input(Object` *object*`)` sends an arbitrary object specifying an input value.

Both the distributor and translator define and use methods in this protocol, as shown in Figure 3. Before we discuss how exactly the input and output calls are processed, let us first define the range of architectures we support based on this protocol.

Range of Application-Sharing Architectures

Given n users sharing an application, we can define a set **U** of user interface components:

U = $\{u_i \mid 1 \le i \le n$, n is the number of users, and u_i is the user interface component running on user$_i$'s computer.$\}$

and a set **P** of program replicas that can run on the computers:

P = $\{p_i \mid 1 \le i \le n$, n is the number of users, and p_i is the program replica that can run on user$_i$'s computer.$\}$

In a particular architectural mode, each user interface component is mapped to one of the program replicas in **P**. Therefore, there is a many-to-one mapping f from **U** to **P**,

f:**U**→**P**, such that $f(u_i)=p_j$, $u_i \in \mathbf{U}$, $p_j \in \mathbf{P}$ if p_j generates output messages for u_i in response to u_i's input messages.

In order to synchronize all of the program replicas in $f(\mathbf{U})$, input messages from a user interface component u_i are broadcast to all program replicas in $f(\mathbf{U})$. In order to synchronize all of the user interface components in **U**, output messages from a program replica p_j in $f(\mathbf{U})$ are broadcast to all user interface components u_i such that $f(u_i)= p_j$. We shall see how exactly this is done in the next section.

According to the most general definition of f, it is possible that $p_j \in f(\mathbf{U})$ and $f(u_j)\neq p_j$ – i.e. a user interface component is not mapped to the local program replica when some other user interface components are mapped to the latter. One of our major goals for supporting a range of architectures is to allow each user interface component to be mapped to a program replica that provides the best average response time for it. As we prove below, a mapping f, where $p_j \in f(\mathbf{U})$ and $f(u_j)\neq p_j$, violates the goal. In other words, if the goal is to be true, it must be the case that $f(u_j)=p_j$ if $p_j \in f(\mathbf{U})$. Therefore, our system supports only supports mappings f, where $f(u_j)=p_j$ if $p_j \in f(\mathbf{U})$.

Before giving the proof, let us first define what we mean by response time. We assume a simple model of interaction between a user interface component and the program replica to which it is mapped. In this model, the user interface component initiates the interaction by sending a single input message to the program replica, and the program replica responds with a single output message. Given that $output_h$ is the output message a program replica produces in response to an input message $input_h$, and that a user interface component u_i is mapped to a program replica p_j, we define the response time $rt_{i,j}(h)$ as follows.

$rt_{i,j}(h) = rtt_{i,j}(h) + proc_j(h)$, where

$rtt_{i,j}(h)$ is the *round trip time* associated with the exchange of $input_h$ and $output_h$ between u_i and p_j: i.e. the time it takes for $input_h$ to travel from u_i's computer to p_j's, plus the time it takes for $output_h$ to travel from p_j's computer to u_i's. If i=j, $rtt_{i,j}(h) = 0$.

$proc_j(h)$ is the time it takes for p_j to process $input_h$ and produce $output_h$.

We also make a simplifying assumption that $input_h$ and $output_h$ represent respectively all of the input and output messages of each of the user interface components – i.e. $rt_{i,j}(h)$ is the average response time that u_i gets from p_j for each of its input messages.

Given these assumptions, we now prove the following.

If there exists a mapping f, where $p_j \in f(\mathbf{U})$ and $f(u_j)\neq p_j$, then there must exist at least one user interface component that does not receive the best average response time available from one of the program replicas in **P**.

Proof: Suppose that $f(u_j)=p_k$, where $j \neq k$. If p_k does not provide the best average response time for u_j, the proof is trivial. Now, suppose that p_k provides the best average response time for u_j. Then, it must be the case that $proc_j(h) > rtt_{j,k}(h) + proc_k(h)$. Because $p_j \in f(\mathbf{U})$, there must exist u_i such that $f(u_i)=p_j$ and $i \neq j$. If we look at the make-up of the response time for u_i, it is $rt_{i,j}(h) = rtt_{i,j}(h) + proc_j(h)$. We can improve response time of u_i by mapping it to p_k, and sending its input and output messages through u_j's computer. In this case, $rt_{i,k}(h) = rtt_{i,j}(h) + rtt_{j,k}(h) + proc_k(h)$, and hence, $rt_{i,k}(h) < rt_{i,j}(h) = rtt_{i,j}(h) + proc_j(h)$ because $rtt_{j,k}(h) + proc_k(h) < proc_j(h)$. Therefore, u_i cannot receive the best average response time available by being mapped to p_j, violating our assumption.

Therefore, if we are to provide the best average response time for all of the user interface components, it must be the case that the inverse of ($p_j \in f(\mathbf{U})$ and $f(u_j) \neq p_j$) is true – i.e. $f(u_j)=p_j$ if $p_j \in f(\mathbf{U})$.

Thus, given a user-interface component whose I/O protocol can be translated to the abstract protocol, our system will support any of the architectural mappings defined by the function above. The I/O protocols on which current application-sharing systems are based can be mapped to this protocol. Moreover, the architectural mappings supported by these systems are a subset of those we can support. As a result, the range of architectures we can support includes all of the application-sharing architectures we know of. Consider how some example architectures can be supported by our system:

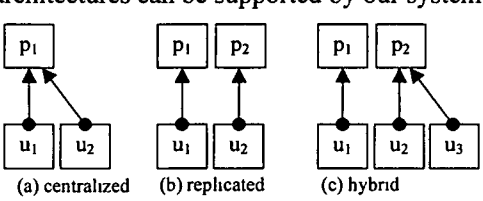

Figure 4 Examples of the Architectural Mappings Supported.

(1) Shared VNC (Li, Stafford-Fraser et al., 2000): The user-interface component here is the VNC layer, and the program component consists of all of the layers above the user-interface component. The user-interface component of each user is mapped to a central program component.

(2) XTV (Abdel-Wahab and Feit, 1991)/ Shared X (Garfinkel, Welti et al., 1994): Same as above except that the user-interface component consists of X and all layers (such as VNC) below it.

(3) Multiuser Suite (Dewan and Choudhary, 1992): Same as above except that the user-interface component consists of the Suite user-interface management system and all layers (such as Motif, X, and VNC) below it.

(4) JCE (Abdel-Wahab, Kim et al., 1999)/ Habanero (Chabert, Grossman et al., 1998): The user-interface component consists of the Java AWT layer

and all layers below it, and the program component consists of all layers above the user-interface component. Each user-interface component is mapped to a local replica of the program component.

(5) DistView (Prakash and Shim, 1994): Same as above except that the user-interface component consists of all layers involved in implementing DistView interface objects.

(6) GroupKit (Roseman and Greenberg, 1996): Same as above except that the "user-interface" component consists of GroupKit shared environments.

(7) Hybrid Architecture: The examples (1)-(3) above are centralized architectures (Figure 4(a)) while (4)-(6) are replicated architectures (Figure 4(b)). Keith Lantz proposed (Lantz, 1986) (but as far as we know, never implemented) a hybrid architecture in which, as in the replicated architecture, there are multiple program replicas, and as in the centralized architecture, multiple user-interface components are mapped to a replica (Figure 4(c)). This kind of architecture is also allowed by the mapping function defined above.

In general, our system can support any architecture defined by the layered zipper model (Dewan, 1998) as long as the communication protocols used by the layers are special cases of our abstract protocol. In addition, we can support hybrid architectures of the kind shown in Figure 4(c), which were not included in the zipper model.

Input/Output Distribution

We now describe how input and output are processed to support the architectural mappings defined above. As mentioned earlier, we replicate the distributor and the translator on all of the users' computers, and have the replicated distributors carry out the input/output distribution tasks for the client system. Figure 5 shows the distributor-translator replication for the mapping f_1: $\mathbf{U}_1 \rightarrow \mathbf{P}_1$ shown in Figure 4(c). As we can see, there are two different sets of distributors – a set $\boldsymbol{M}$ of distributors that have program replicas in $f_1(\mathbf{U}_1)$ running on respective computers, and a set $\boldsymbol{S}$ of the remaining distributors that do not. The mapping f_1 forces a dependency relationship from a distributor $\boldsymbol{s}$ in $\boldsymbol{S}$ to a distributor $\boldsymbol{m}$ in $\boldsymbol{M}$, in that $\boldsymbol{s}$'s local user interface component depends on $\boldsymbol{m}$'s local program replica for producing its output messages. For this reason, we refer to $\boldsymbol{s}$ as a slave distributor, and $\boldsymbol{m}$ as $\boldsymbol{s}$'s master distributor. A master distributor $\boldsymbol{m}$ does not depend on any other distributor in the sense that its local user interface component does not depend on a remote program replica for processing its input messages. Like distributor$_1$ in Figure 5, a master distributor may not have a slave distributor.

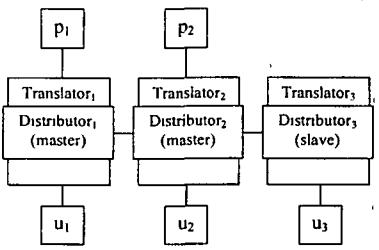

Figure 5 Master and Slave Distributors.

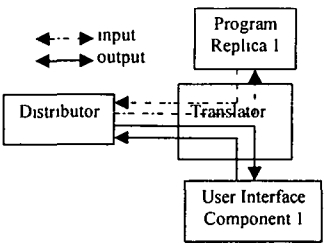

Figure 6 Distributor-Translator Interaction

Supporting a mapping f_i requires both the distributors and the translators to work together to distribute input/output messages to correct destinations. As mentioned before, the part that each translator has to play to enable correct distribution is simple. As Figure 6 shows, each translator works as an intermediary between the local distributor, the local program replica and the local user interface component – it simply translates and relays input/output messages from one party to another. In particular, the translator translates generic input/output messages from the local distributor into translator-specific messages, and relays the translated input and output messages to the local program replica and the local user interface component respectively. In response to translator-specific input and output messages respectively from the local program replica and the local user interface component, the translator translates them into generic messages, and relays the translated messages to the local distributor.

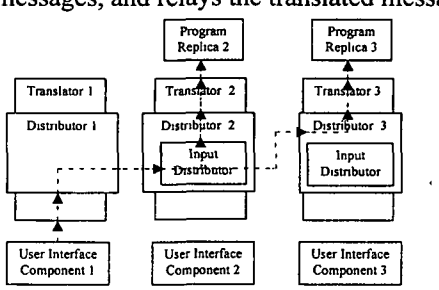

Figure 7. Input Distribution.

Given an input message sent from a translator, the distributors must cooperate to distribute the input message to all of the program replicas in $f_i(\mathbf{U}_i)$. In order to see how we have implemented the distribution, we follow an input message as it travels through different distributor modules. When a distributor receives an input message from the local translator, it must distribute the message to all of the master distributors.

In order to facilitate the distribution, we can consider having a centralized input distributor, to which each distributor passes the incoming input messages for the distribution. The input distributor would need to keep track of all of the

master distributors in the current mapping The major drawback of having a single input distributor is that it forces each input message to go through the input distributor's computer regardless of the locations of the input message's origin distributor and its master distributor, thereby causing poor response time.

We can improve the response time by making a distributor send each input message directly to its master distributor. Thus, we can consider installing input distributors on all of the computers, and having each distributor send input messages to the local input distributor. However, with this approach, any change to the composition of master distributors must be synchronously broadcast to all of the computers to guarantee that an input message is distributed to the same set of master distributors regardless of which input distributor distributes it. With asynchronous broadcast, an input message can be distributed to a different set of master distributors depending on which input distributor performs the distribution. This means that a new master distributor may miss some input messages from an input distributor that is notified about the new distributor belatedly. In order for the update to be synchronous, we must make sure that all of the current master distributors have received the same sequence of input messages before we begin the update. Moreover, we must make sure that no input distributor distributes an input message until the update ends. Therefore, the more input distributors we have, the more expensive the synchronous updates are.

As Figure 7 shows, we resolve this conflict by installing input distributors only on computers with master distributors. When a distributor receives an input message from the local translator, it passes the input message to the master distributor's input distributor (local input distributor if the master distributor happens to be itself). This approach supports direct input message delivery from slave distributors to their master distributors, and limits the range of master distributor composition updates to master distributors only.

Once a program replica receives an input message, it processes the message, and responds with an output message. The output message has to be distributed to all of the user interface components mapped to the program replica. Because only master distributors have program replicas running on their respective computers, they are the only distributors receiving output messages directly from their local translators. Each master distributor has a module called output distributor, which keeps track of all of the distributors with user interface components mapped to the local program replica. On receiving an output message from the local master distributor, the output distributor broadcasts the message to all such distributors, each of which sends the output message to the local translator. Figure 8 shows the itinerary of an output message, completing our discussion of input/output distribution in our system.

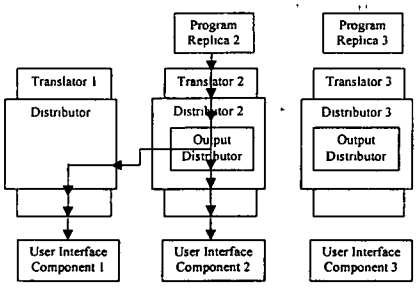

Figure 8. Output Distribution.

Translator

As we have seen in the discussion above, the client of the distributor is responsible for translating between the specific and abstract I/O protocols. If the specific I/O protocol is defined by a user-interface tool such as X, VNC, or JCE, then the translator must be implemented once for the tool (Figure 3(b)) – all clients of the tool can then be shared without making them collaboration aware. On the other hand, if the I/O protocol is specific to an application, then a special translator for the application must be implemented. Moreover, it may not be possible to translate all application-specific protocols to the generic protocols. The range of I/O protocols that can be translated measures the composability of our system, and the effort required to translate the (composable) protocols measures the automation of our system.

A translator must implement the I/O protocol defined earlier to receive input and output from the distributor. In addition, it must implement:

- `startProgram()`, which is responsible for starting the program component on the local computer.
- `startUIComponent()`, which is responsible for starting the user-interface component on the local computer.

These two methods are used to create the various architectural mappings supported by our system. The mapping of a user-interface component to a program component is done by the distributors based on the mapping defined by the user.

Composability & Automation Experiments

To evaluate the composability and automation of our system, we used it to share four applications: a text field editor, an outline editor, an object-based drawing

editor, and a design pattern editor. We developed the text field and outline editors for the particular purpose of this evaluation, while the drawing editor and the pattern editor had been built before. The drawing editor was implemented at our university while the pattern editor was implemented elsewhere.

The text, outline and pattern editors are divided into program and user-interface components that communicate using special cases of the abstract protocol defined by us. Thus, implementing translators for them was straightforward. The drawing editor, on the other hand, is not based on the layered framework assumed in our design. Instead, it is based on the MVC (Model-View-Controller) framework (Krasner and Pope, 1988), in which input and output processing are separated into controller and view objects, and the semantics are implemented in a model object. A controller invokes methods in the model in response to user input, which change the state of the model. The model responds to a state change by notifying the view that it has changed. The view responds to the notification by querying the model for its new state and updating the display appropriately. A model can be composed with more than one view-controller pairs, each implementing a different user-interface.

We can map this architecture to the layering framework we have assumed in our framework by mapping the model to the program component and the controller and view, together, to the user-interface component. However, there is a fundamental difference between the assumed and MVC framework. The program component does not push output information to the user-interface component by sending output messages, instead the latter pulls this information from the latter in query requests.

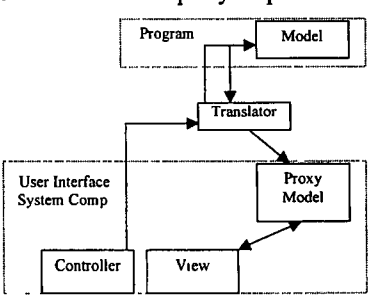

Figure 9 Mapping MVC to Layered Model

Despite this problem, we managed to map the MVC framework to the layered framework, by adding an extra component to the architecture. Figure 9 shows how we did the mapping. The model maps to the program component, and the controller, view, and a "local proxy" of the model together form the user-interface component. The controller serves as the source of the input messages sent by a user-interface component. As mentioned earlier, the view cannot serve as the sink of output messages. Therefore, we created a local proxy model that serves as this

sink. In order to synchronize the real model and the proxy model, we make sure that a user input message (generated by the controller) is sent to both objects. The translator and distributor treat the copy sent to the real model as an input message, and the copy sent to the proxy model as an output message.

This mapping is clearly clumsy, creating an additional copy of the model on each computer, which can reduce both the time and space efficiency of the system, and cause undesired semantics if the model implements non-idempotent operations. Thus, with this mapping, the "centralized" architecture creates one copy of the real model in the program component but multiple copies of the proxy model in the user-interface components, incurring some of the disadvantages of the replicated architecture. One good side effect, though, is that the proxy model serves as a local cache for the object, giving some of the benefits of caching described in (Graham, Urnes et al., 1996). This example was presented here to show the limitations rather than the strengths of our approach. The technique does show that if replicating a model does not cause any performance or semantic problems, the MVC framework can be mapped to the layered framework assumed in our design.

To evaluate the automation of our approach, we counted the lines of Java code added to clients to compose with our distributors. Table 1 shows our results.

text field	outline	drawing	pattern
74	135	115	220

Table 1. Lines of Java Code for composing with the Distributor

Ideally, the effort required to compose with the system should be considerably less than manually distributing input and output. This is indeed the case, as the size of our distributor is several thousands of lines. Moreover, the functionality of the distributor can be increased (by adding support for dynamic architectural adaptations, for instance) without increasing the functionality of the translator.

Performance Experiments

As mentioned earlier, by supporting multiple architectures in a system, we provide a basis for evaluating the performance of different architectures – in particular, the centralized and replicated architectures. We performed a number of experiments to compare the two architectures. In these experiments we varied the application, its location in the centralized architecture, processing powers of the computers used in the collaboration, and network delays.

As the basis of our comparison, we measured, for each architecture, the fastest time in which two users finished the same task. Instead of the error-prone method of using live users to provide input, we used an actual collaboration log we acquired from MITRE. It is a sequence of n $\langle user_i, input_i \rangle$ tuples, where $input_i$ is

entered by $user_i$. Two input provider modules, each representing the two users, sequentially execute the tuples - the input provider module of $user_i$ provides $input_i$ after $user_i$'s user interface component receives the program's output in response to $input_{i-1}$. This log-based method is more realistic than the approach of using synthetic workloads, which has been used so far for evaluating the performance of collaborative application (Chung and Dewan, 1996, Graham, Urnes et al., 1996; Bhola, Banavar et al., 1998).

computer assignment	Desktop (500 MHz) Desktop (500 MHz)		Desktop (500 MHz) Laptop (133 MHz)	
delay	LAN 0ms	Germany 72ms	Germany +modem 162ms	India 370ms
client	shapes (small operation cost)		knapsack (large operation cost)	
architecture	centralized		replicated	

Table 2 Parameters Varied in the Experiments.

Table 2 shows the parameters we varied in our experiments and the values we used for them. Each entry in the first row of the table shows a computer assignment to $user_1$ and $user_2$ respectively. The two assignments allowed us to test collaborations involving computers with similar and significantly different processing power.

The second row of the table shows the three different network conditions we simulated (or used in the LAN case) and corresponding delays we imposed on the communication between the two users' computers. To base the delays on reality, we measured the average ping round trip times to actual sites for WAN and modem cases. For all experiments, we used the 10 Mbps LAN facility in the Computer Science Department in UNC-Chapel Hill. In the LAN case, we imposed no extra delays on the messages exchanges between the two users. In the Germany case, we assumed the first user was directly connected to the LAN and the second was in Germany, and we added 72ms to each message sent between the two users. The Germany and modem case was the same as the Germany case except that we assumed that the first user was connected to the UNC LAN using a modem. Therefore we added the modem delay to the Germany delay. The India case was the same as the Germany case except that the second user was assumed to be in India.

Row three shows the actual client systems we used. They were chosen to represent two classes of client systems - one with moderate operation cost per input and the other with a high operation cost. For moderate operation cost, we used the object-based drawing editor, which we call shapes. For high operation cost, we used a program implementing a solution to the knapsack problem Though, to the best of our knowledge, such a program has not been used so far for a real collaboration, we believe it is a realistic example since we can expect that two users, especially students, would find it useful to understand the properties of

the knapsack problem collaboratively. For each of the programs, we created a task with an input sequence appropriate for the program while trying not to violate the order among input events in the original MITRE log.

The last row of the table shows the two architectures used in the experiments.

	LAN	Germany
Centralized	2123	2904
Replicated	1823	2623

Table 3. Desktop-Desktop running Shapes.

	LAN	Germany
Centralized	147132	149545
Replicated	147492	154112

Table 4. Desktop-Desktop running knapsack

Table 3 and 4 show the task completion times when we used the two desktop computers. Our hypothesis was that the replicated architecture would always provide the better performance when the computers are of equal power. Table 3 shows that this is true when the users share the drawing editor. However, Table 4 shows that it is not true when the users share the knapsack program. We do not know the exact cause of this result, but conjecture that the knapsack program's long operations (which run on both computers) hamper efficient communication between the two computers.

	LAN	Germany	Germany +modem	India
Centralized	3365	4466	6780	11156
Replicated	5408	6990	7180	8181

Table 5. Desktop-Laptop running Shapes.

Table 5 shows the results when the two users share the drawing editor using a desktop and a laptop computer, and in the central case, the program runs on the more powerful computer. In the LAN case, because the communication cost is low, the better performance is given by a central program running on the desktop machine. As the network delay gets larger, the replicated architecture gives better performance. Interestingly, it is only in the India case that the replicated architecture gives better performance than the central architecture.

	LAN	Germany	Germany +modem	India
Centralized	8460	10980	14770	16320
Replicated	8070	7960	7470	8630

Table 6. Laptop-Desktop running Shapes.

Table 6 shows the results of the same experiments we did for Table 5 except that this time, in the central case, the program runs on the less powerful computer. As expected, the central case always gives worse performance than the replicated case. The difference between the performance of the two architectures is not that significant in the LAN case. This is probably because the replica on the less powerful computer slows down the collaboration for both users.

	LAN	Germany	Germany +modem
Centralized	153891	154302	150566
Replicated	808743	816023	812969

Table 7 Desktop-Laptop running knapsack

Table 7 shows the results of the same experiments we did for Table 5 except that this time we used the knapsack program, which has a much higher operation cost than the drawing editor. As expected, the relative performance of the centralized architecture is much better. We did not try to run the central program on the less powerful computer because of the high operation cost.

It is generally believed that the replicated architecture always gives better performance than the central architecture. It is because of this belief that many researchers have implemented replicated systems, either living with the synchronization and other problems of this architecture or providing complex solutions to these problems. Our experiments show that the replicated architecture does indeed give better performance in some cases. However, there are also realistic cases where it gives worse performance if we consider task completion times. In many of the latter cases, the replicated architecture will probably give better response times. However, task completion times are more important because they take into consideration the fact that a replica on a slow computer can slow the group's progress.

Our results also suggest that it may be useful to dynamically change the architecture mapping as users join or leave the collaboration. Typically, the central component is run on the computer of the first user who joins a collaborative session. As can be seen by comparing tables 5 and 6, if later a user with a significantly more powerful computer joins the session, it may be useful to dynamically migrate the central program to this computer. Similarly, as Table 5 shows, it may be useful to dynamically replicate and centralize an application as new users connected to others through slow connections join and leave the collaboration. Whether dynamic architectural adaptations actually improve task completion times would depend on their cost.

Conclusions and Future Work

This paper makes the following contributions:

- Layer-Independent Application-Sharing: It describes novel mechanisms for supporting a layer-independent application-sharing system. The mechanisms include the abstract I/O protocol and the division of the application-sharing responsibility into a client-specific translator and a client-independent distributor. It presents the results of composability experiments that evaluate the mechanisms, showing that a client-specific

translator is easier to write than a complete, client-specific application-sharing system.

- Multiple architectural mappings: It formally defines a range of architectural mappings and describes novel distributed mechanisms to implement this range.
- Architecture Experiments: It experimentally shows that the performance of an application-sharing architecture depends on the computers used by the collaborators, the speed of the connections between them, the cost of the operations performed by them, and the location of the central component in a centralized architecture. As far as we know, results of experiments comparing the performance of different architectures have not been reported so far, though the performance of different techniques for implementing the same architecture have been reported (Graham, Urnes et al., 1996). Moreover, it is generally believed that the replicated architecture always gives better performance. Our experimental results show that, in many realistic situations, the centralized architecture gives better performance.

This work is only a first cut at providing flexible support for the application-sharing architecture. There are several possible directions for extending it:

- More Flexible I/O Protocol: It would be useful to extend the abstract I/O protocol so that it can support a greater variety of concrete I/O protocols such as the one supported by the MVC framework.
- More experiments: It would be useful to perform composability and performance experiments using a wider range of clients. Moreover, it would be useful to measure the cost (such as translation cost) client systems pay for genericity. Furthermore, it would be useful to create and use additional logs in performance experiments. Finally, it would be useful to see if it is possible to devise experiments involving live users that can be used to correctly compare the performance of different architectures. Other kinds of systems such as databases and compilers have relied on benchmarks, but these are not interactive systems. As we understand better the functionality and architecture of application-sharing systems, it is important to make evaluation of their performance a first-class issue.
- Architecture Policy: It would be useful to develop techniques for automatically determining the architecture to be used for an application-sharing session based on properties of the network connections, the computers of the collaborators, and the application.
- Dynamic Architecture Adaptation: It would be useful to explore and evaluate techniques for dynamically migrating, replicating, and centralizing a program component. The idea of dynamically migrating a program component has been explored earlier. Our previous work has extended an X-specific application-sharing system with the capability to dynamically migrate the X client (Chung and Dewan, 1996). Similarly, general

distribution systems such as DACIA (Little and Prakash, 2000) support migration of arbitrary objects, but do not provide application sharing. It would be useful to extend these works to provide a client-independent application-sharing system that supports dynamic architecture adaptations. Finally, it would be useful to separate mechanisms to support architectural adaptations from those that distribute I/O so that the adaptation mechanisms can be used in existing application-sharing systems.

- Pervasive Application-Sharing: As the idea of a world populated with a wide-variety of computers ranging from palmtops to live boards is realized, the idea of supporting uninterrupted collaborations among mobile users becomes a possibility. In this world, the collaborators may use computers with varying processing power, and particular collaborators may change computers based on their location. Our experiments hint that flexible support for application-sharing architecture would aid pervasive collaborations. For example, if two users are using desktop computers, then the application should probably be replicated, but if one is using a handheld and the other a desktop, then the application should probably be centralized on the desktop. It would be useful to port our approach to lightweight computers to see how well it works for supporting mobile collaboration.

Acknowledgments

This work was supported in part by U S. National Science Foundation Grant Nos IRI-9508514, IRI-9627619, CDA-9624662, and IIS-997362.

References

Abdel-Wahab, H., O. Kim, et al. (1999): 'Java-based Multimedia Collaboration and Application Sharing Environment', *Colloque Francophone sur l''Ingeniere des Protocoles*, April 1999

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

Bhola, S , G. Banavar, et al (1998). 'Responsiveness and Consistency Tradeoffs in Interactive Groupware', Proceedings of ACM Computer Supported Cooperative Work, 1998.

Chabert, A , E Grossman, et al. (1998): 'Java Object-Sharing in Habanero', *Communications of the ACM*, vol 41, no. 6, June 1998, pp. 69-76.

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

Dewan, P. and R. Choudhary (1992): 'A High-Level and Flexible Framework for Implementing Multiuser User Interfaces', *ACM Transactions on Information Systems*, vol 10, no. 4, October 1992, pp. 345-380.

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

Dewan, P., R. Choudhary, et al. (1994). 'An Editing-based Characterization of the Design Space of Collaborative Applications', *Journal of Organizational Computing*, vol. 4, no 3, 1994, pp 219-240

Dewan, P and R Choudhary (1995). 'Coupling the User Interfaces of a Multiuser Program', *ACM Transactions on Computer Human Interaction*, vol. 2, no. 1, March 1995, pp. 1-39

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

Garfinkel, D., B. Welti, et al. (1994)· 'HP Shared X: A Tool for Real-Time Collaboration', *Hewlett-Packard Journal*, April 1994

Graham, T. C. N., T. Urnes, et al. (1996)· 'Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware', *Proceedings of the Ninth Conference on User Interface Software and Technology*, October 1996.

Hill, R., T. Brinck, et al. (1994): 'The Rendezvous Architecture and Language for Constructing Multiuser Applications', *ACM Transactions on Computer Human Interaction*, vol. 1, no. 2, June 1994.

Ishii, H. and M. 'Ohkubo Design of a Team Workstation', *Multi-User Interfaces and Applications*, North Holland.

Krasner, G E and S. T. Pope (1988): 'A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80', *Journal of Object-Oriented Programming*, vol 1, no. 3, August/September 1988, pp. 26-*49.*

Lantz, K. A. (1986): 'An Experiment in Integrated Multimedia Conferencing', *Proceedings of Conference on Computer-Supported Cooperative Work*, December 1986

Li, S. F, Q. Stafford-Fraser, et al. (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, vol 2, , March 2000, pp. 717-721.

Little, R. and A. Prakash (2000). 'Developing Adaptive Groupware Applications Using a Mobile Component Framework', *Proceedings of ACM Computer Supported Cooperative Work*, 2000

Prakash, A. and H. S. Shim (1994): 'DistView: Support for Building Efficient Collaborative Applications using Replicated Active Objects', *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, October 1994.

Richardson, T., Q. Stafford-Fraser, et al. (1998): 'Virtual Network Computing', *IEEE Internet Computing*, vol. 2, no. 1, January/February 1998

Roseman, M. and S. Greenberg (1996): 'Building Real-Time Groupware with GroupKit, A Groupware Toolkit', *ACM Transactions on Computer-Human Interaction*, vol. 3, no. 1, 1996

Scheifler, R. W. and J Gettys (1983)· 'The X Window System', *ACM Transactions on Graphics*, vol 16, no. 8, August 1983, pp. 57-69.

Stefik, M., D G Bobrow, et al (1987). 'WYSIWIS Revised. Early Experiences with Multiuser Interfaces', *ACM Transactions on Office Information Systems*, vol. 5, no 2, April 1983, pp. 147-167.

Sun, C. and C. Ellis (1998) 'Operational Transformation in Real-Time Group Editors· Issues, Algorithms, and Achievements', *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, November 1998.