Prasun Dewan[1]

2. Shared Window Systems

In this chapter we will look at shared window systems, which provide the basis of several commercial collaborative tools used today such as LiveMeeting and Webex. In this process, we will study several concepts including single-user window systems, proxies, WYSIWIS and near- WYSIWIS coupling, and replicated/centralized architectures.

WYSIWIS

As we saw earlier, a collaborative application couples the users, that is, allows each user's output to be influenced by some input of another user. Thus, each user command can result in output to both that user and others. We will refer to the output of a command by the inputting user and observing users as *local* and *remote feedback*, respectively. Coupling determines (a) *what* the remote feedback is for each remote user, and (b) *when* this feedback is given.

There is a large theoretical design space of possible couplings. As a developer of a collaborative application or infrastructure, we must choose one or more of these points.

The choice is tricky if we are designing an application, as we must choose a coupling that is conducive to the kind of collaboration supported by the application. If it's not designed right, users can be forced to share more or less than what they desire.

This choice is even trickier if we are designing a general infrastructure, as we do not know the specifics of an application. In fact, it seems impossible that an infrastructure could automatically support a reasonable set of couplings. After all, each (single-user and collaborative) application determines what the local feedback is to each input command. If an infrastructure does not know the local feedback of an input command, how can it know the remote feedbacks for different observers?

The key to automating coupling is to determine not the relationship between input and remote feedback, but the relationship between local and remote feedback. If somehow an infrastructure could determine the local feedback of an application, then it could automatically display the remote feedback by applying some application-independent function to the local feedback. The simplest such function is the identity function, which makes the remote and local feedback the same. The coupling defined by this function is called WYSIWIS (What You See Is What I See).

In the definition of WYSIWIS, we ignored the timing aspect of coupling. The term, WYSIWIS, really implies that users see the same thing at all times, that is, the remote feedbacks are given at the same time as the local feedback. However, because of network delays, we can never achieve this. Therefore, we must consider some approximation of this goal. Some possibilities, in order of increasing guarantees, are:

- *Eventual consistency*: If all interaction stopped, all users will see the same thing after a finite amount of time. This definition is easy to implement, but the fact that it gives no timing guarantee seems inconsistent with an objective of WYSIWIS, which is synchronous collaboration.
- *"Best-Effort" real-time:* The infrastructure makes a "best effort" to give immediate feedback. This means that as soon as it receives local feedback, it asks existing communication technologies to send the feedback to others.
- *Real-time feeling*: The delays are such that the users feel they are working in real time. However, this is a subjective definition.
- *Guaranteed real-time*: The delays are not noticeable by other users. Research has shown that users do not notice delays of less than 50ms.

Current infrastructures support best-effort real-time, though some have special features to reduce the problem of high network delays.

The figure below summarizes the above definition of WYSIWIS coupling.

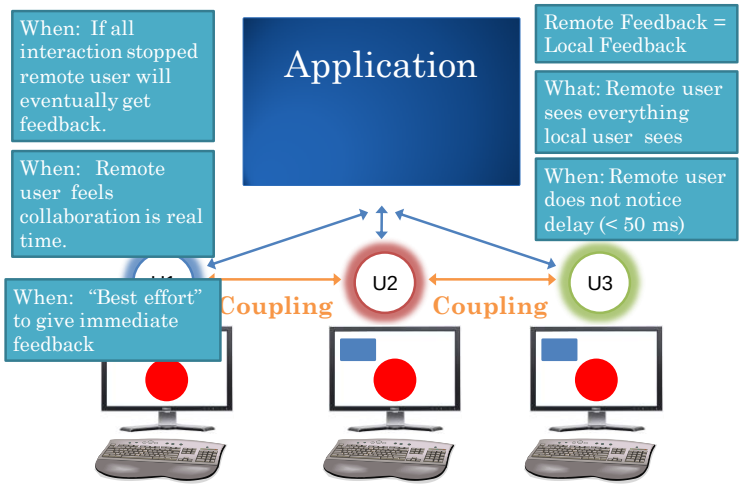


Figure 1 WYSIWIS Coupling

Pros and Cons of WYSIWIS

In (strict) WYSIWIS, each user of a collaborative application sees the same thing. This means that if a user moves/scrolls/minimizes a window, the window moves/scrolls/minimizes on all other screens also.

Researchers have found that such close coupling can lead to “window wars” and “scroll wars” as different users try to fight over the positions of windows and scrollbars. WYSIWIS also has the problem of high communication overhead – every output event is sent to all other users. This is particularly an issue when users do not want to share some of these events, such as window and scrollbar events.

On the other hand, it is easy to understand – every user knows what the other users are seeing. Moreover, it is application-independent – we don’t need to make any application-specific assumptions about the local feedback to determine the remote feedback. The application-independence implies that the coupling can be implemented automatically by an infrastructure. Let us see below how this can be done.

Shared I/O Systems

In general, automating some functionality requires making some assumptions. For example, a user-interface infrastructure automates I/O by making assumptions about the kind of user-interfaces needed in interactive applications. For instance, Java user-interface libraries assume users do not want round buttons.

Similarly, a collaboration infrastructure must also make assumptions to automate coupling and other collaboration functions. In particular, an infrastructure that computes remote feedback as a function of local feedback must make some assumptions about the nature of the input and output of an application, that is, the I/O system used by the application. It must also decide to what degree the application and the I/O system are aware of collaboration issues.

A class of collaboration infrastructures, which we will refer to as *shared I/O systems*, assume that both the application and I/O systems are completely unaware of collaboration. All awareness is concentrated in the shared I/O system.

A shared I/O system must be able to tap the input and/or output of an application. For now, we will assume that it is possible to insert a proxy between the application and the I/O system, just as it is possible today to put a proxy between a web client and web browser. This is shown in the figure below. As indicated by the two colors shaded in the proxy box, the proxy appears to the application to be the I/O system, and to the I/O system to be the application. The proxy forms the collaboration infrastructure. We will also refer to it simply as infrastructure.

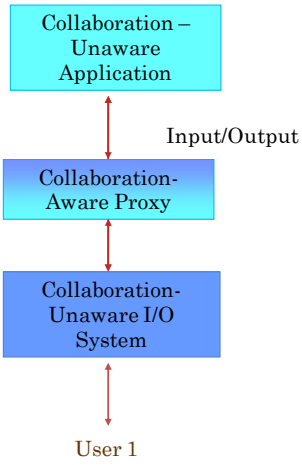


Figure 2 Proxy-based Shared I/O System

It is possible to create different kinds of shared I/O systems based on the underlying I/O system. Let us look at some choices for I/O systems.

The programming languages we use support Console I/O that allows applications to read and write lines of textual input. Such I/O is also called teletype I/O because it is based on the typewriter model of entering and viewing lines of text. We could build a shared I/O system, but such a system would not support menus, graphics, buttons and other aspects of GUIS (Graphical User Interfaces).

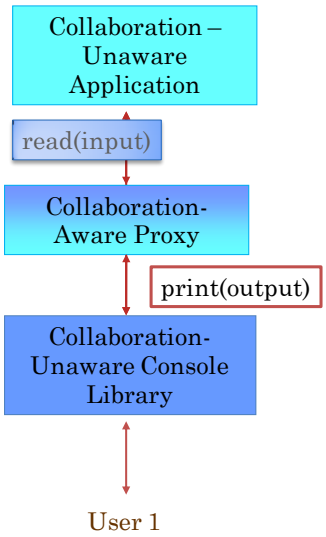


Figure 3 Shared Teletype System

Those of you who have gone beyond teletype user-interfaces have probably used toolkits such as AWT and Swing. These toolkits provide objects called *widgets*. Examples of widgets include text boxes, sliders, menus, and buttons. To users, widgets are screen regions with which they can interact. To applications, they are objects with state such as the text displayed in a textbox or the number displayed by a slider. The output of an application consists of calls made to the widgets that change their state. The inputs of an application are events received from the widgets about changes to their state in

response to user actions. For instance, when the user changes the text in a text box, the application gets a “text changed” event.

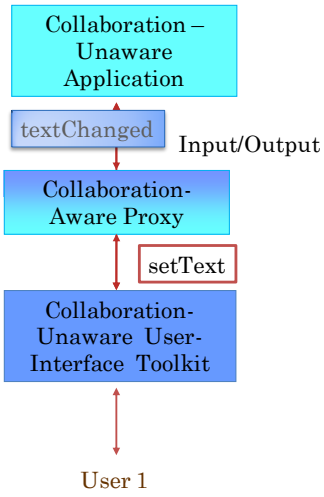


Figure 4 SharedToolkit

Today a variety of toolkits are offered. For instance, Java programmers are offered at least three toolkits – AWT, Swing, and SWT – to create desktop interfaces. A shared toolkit can only offer collaboration facilities to programmers who use the toolkit. Moreover, as we will see later, it is more difficult to add proxies between a toolkit and an application.

A toolkit is implemented on top of a window system, which provides facilities to create rectangular regions on the screens. Unlike a toolkit, a window system does not type the rectangular regions as textboxes, sliders, and other widgets. Thus, it does not interpret the user input, and simply sends the hardware input events to the application. These include key and mouse button presses and releases and the position of the mouse. For example, as shown in the figure below, when a user presses the key, ‘a’, the window system sends the application an input indicating the status of the ‘a’ key, the name of the window with which the user was interacting when the key was pressed, the x and y coordinates of the mouse in the window, and other pieces of information not shown in the window such as the status of the control, shift, and other modifier keys.

In general, there are fewer window systems than toolkits, and in some of these systems, it is easy to add a proxy. Moreover, there is more commonness among different window systems than toolkits. Therefore, it is possible to create translators between these systems, allowing users to collaborate using different window systems. As a result, in this chapter, we will focus on shared window systems. The figure below shows the architecture of such a system. Here, the collaboration-unaware application includes not only the code written by the application programmer but also toolkit libraries referenced by this code.

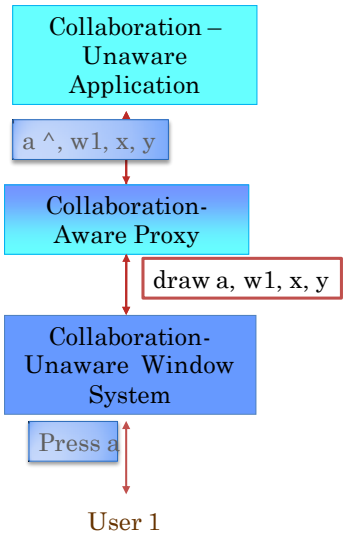


Figure 5 Shared Window System

Centralized Architecture

The figures above show the architecture of a shared window system when there is only one user in the collaboration. In the multi-user case, each user interacts with a separate (collaboration-unaware) window system. In other words, the window system is "replicated" on each user's computer. The word "replica" is quoted because the window systems are not true replicas - windows created by single-user applications on each computer are not shared. It is also possible to replicate the (collaboration-unaware) application. However, for now we will assume that the application is centralized on one of the user's computers. Typically, a shared window system allows a user to create a collaborative session by sharing an application. The shared application runs on this user's computer, which is called the master computer. Users who join this session share this copy of the application, and their computers are called slave computers. On each computer, collaboration-aware proxies interact with each other to share the windows created by the application. This architecture is shown in the figure below.

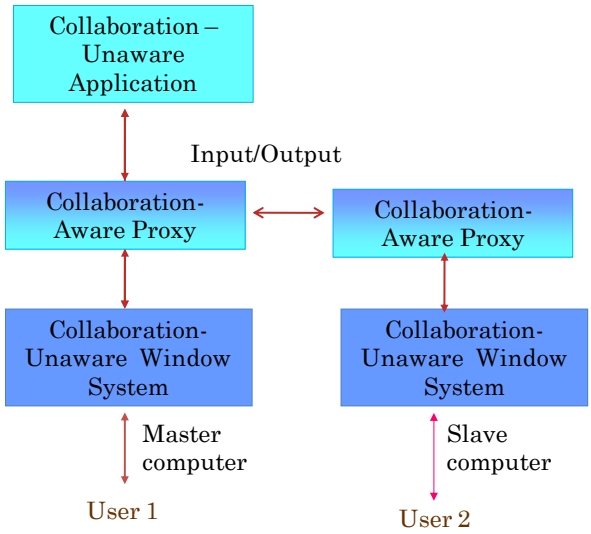


Figure 6 Centralized Architecture

In the figure above, we have created an infrastructure proxy module on each computer. It is also possible to create a single proxy on the master computer that communicates with the window systems of all computers. However, such an architecture can be implemented only on top of a distributed window system – a window system such as the X window system that can interact with remote applications.

The above architecture generalizes to a centralized I/O system by simply replacing the collaboration-unaware window system with a collaboration-unaware I/O system.

Input Relay and Output Broadcast

The architecture above does not explain the nature of the communication among the proxies created on different computers. The figure below shows the basic elements of this communication. To ensure all users see the same window contents, output from the application is broadcast to all window systems. To allow all users to interact with the central application, the input from all users is relayed to the central computer.

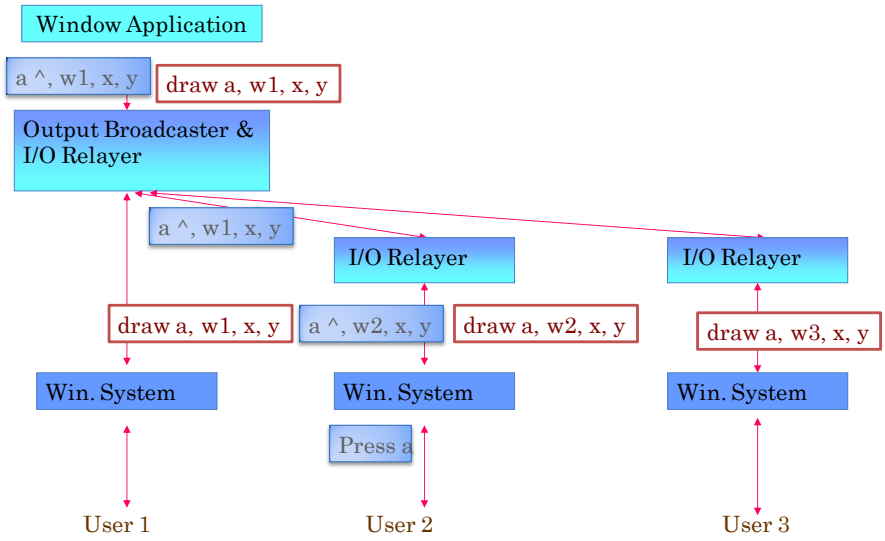


Figure 7 Input Relay and Output Broadcast

Let us follow the event sequence shown in the figure to concretely understand input relay and output broadcast. Suppose a user presses the key *a* while the pointing device is at position *x, y* of window *w2*. In the single-user case, the window server would send information about this event to the local window client. However, in this architecture, this event is intercepted by the local I/O relayer, which forwards it to the infrastructure module on the master computer, which in turn forwards it to the central window client on that computer. The output operation executed by the client to draw the input character is then broadcast to all the window servers via the infrastructure modules.

As the window systems are collaboration-unaware, they assign different identifiers to their local copies of each shared window. For instance, in the figure above, window w1 in user 1's window system is called w2 and w3 on user 2 and 3's window systems, respectively. Therefore, the collaboration-aware infrastructure modules must translate among these identifiers when relaying input and broadcasting output. In the figure above, the proxy on user 2's computer translates w2 to w1 before submitting user 2's input to the master. Conversely, the proxies on user 2's and user 3's computers translate the w1 in the output to w2 and w3, respectively. Thus, the collaboration-unaware application thinks it is interacting with a single window, w1, when in fact multiple versions of this window exist on different computers. The infrastructure module translates between the identifier on the master computer, which is the one the application uses, and those on the slave computers.

The basic idea of input relay and output broadcast applies to any centralized shared I/O system. However, there are several complications that are specific to shared window systems, which are discussed below.

Concurrency, Invalid I/O Sequences, and Floor Control

Consider concurrent input from two different users, as shown in the figure below, where two users simultaneously press the 'a' and 'b' keys. These events will be received in some order by the application

– the local user's input will likely be received first, as shown in the figure. The output to these events can get undesirably mixed, but if this does happen, it may not be a real problem, as users know this is happening and can coordinate to ensure only one of them interacts. Thus, it seems a shared window system can and should allow concurrent input to allow users to enter non-conflicting input.

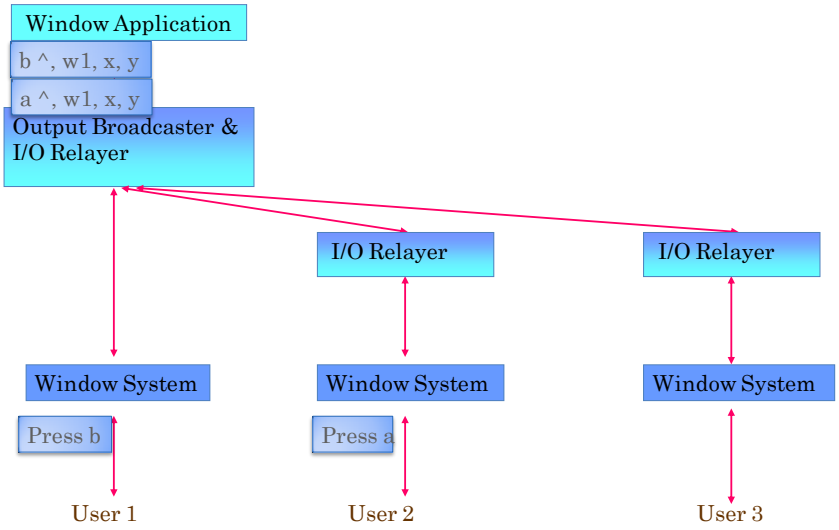


Figure 8 Valid concurrent input

In the above example, the input received by the collaboration-unaware application could have also been generated by a single user by pressing the 'b' key before releasing the 'a' key. In general, however, concurrent interaction can result in input sequences that cannot be generated by a single-user system. The figure below shows such a sequence.

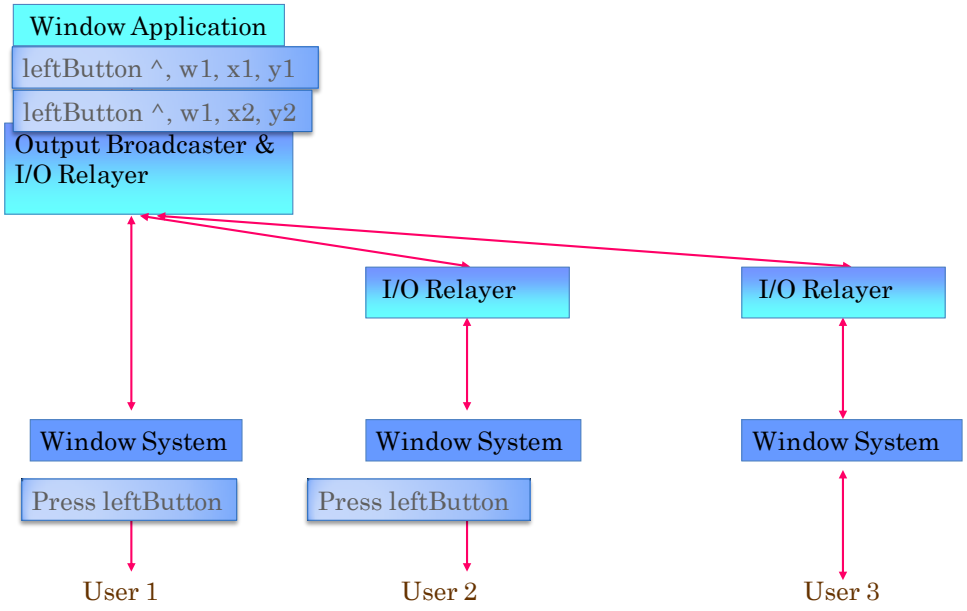


Figure 9 Invalid Concurrent Input

Here the two users have pressed the left mouse button simultaneously. The application thus gets two left mouse button presses without an interleaving left mouse button release, which can never happen in the single-user code. An input sequence that cannot be generated by a single user can break explicit or implicit assertions made by the application and the I/O libraries to which it is linked.

For this reason, shared window systems force explicit floor control – only one user interacts with the application at one time and an explicit sequence of user commands is required to transfer control from the floor holder to another user. These commands ensure that such invalid sequences cannot occur. In the example above, if both users press the mouse button simultaneously, only the flood holder's event will be processed.

Processing Window Manipulation Events

In the example above, we considered window events that must be processed by the application. There are several events that can be processed by the window server without requiring application intervention. We will refer to these as window manipulation events, as they modify data of interest to the user interface but not the program component. The figure below shows an example of such an event: the move window event generated in response to the user moving a window. This event is processed by the window system itself, which moves the window. In some modular systems, a separate component called the window manager generates the event and decorates a window with borders that allow users to indicate that the window should be moved. Here we assume that the window manager, if it exists, is part of the window system. Other examples of window manipulation events are minimize, maximize, and bring window to the front or back of another window; that is, change its stacking order.

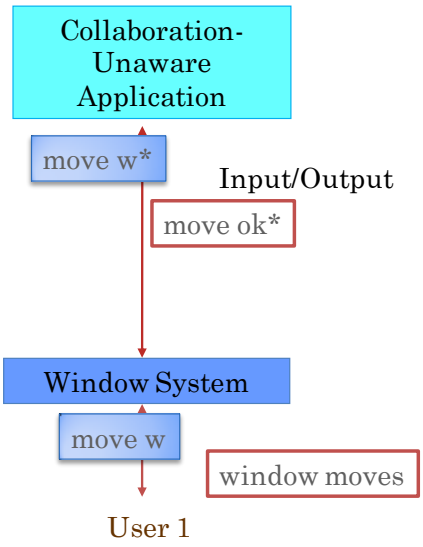


Figure 10 Processing of Internal Events in the single-user case

As shown in the figure above, an application can express interest in such events, in which case the window system sends them to the application and processes them only if the application says it should.

In the rest of the discussion, we will assume these events are not sent to the application. However, we will assume that the window system allows applications to intercept these events.

The figure below shows how window manipulation events are processed. The proxy on each computer asks its local window system to forward these events to it. It then forwards this event to all other proxies, which also move the window.

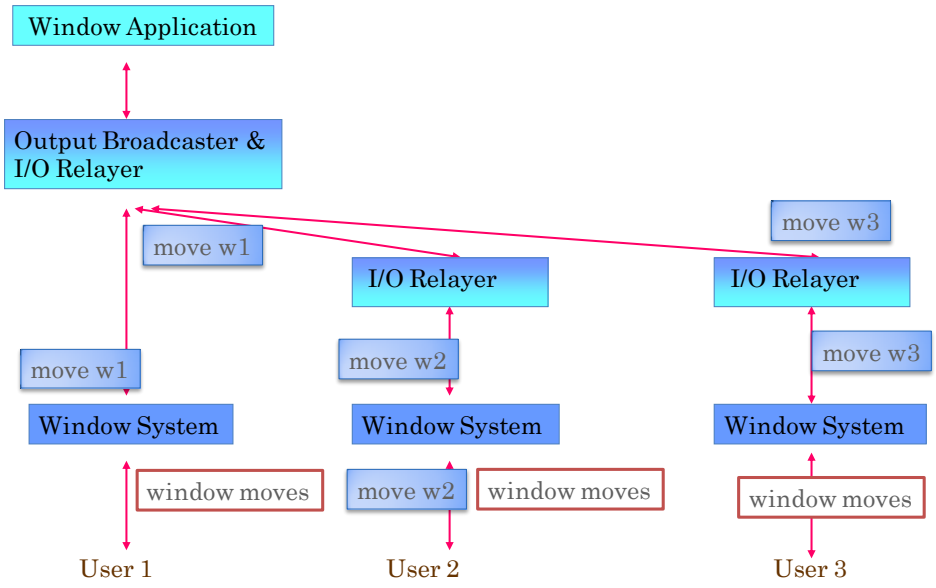
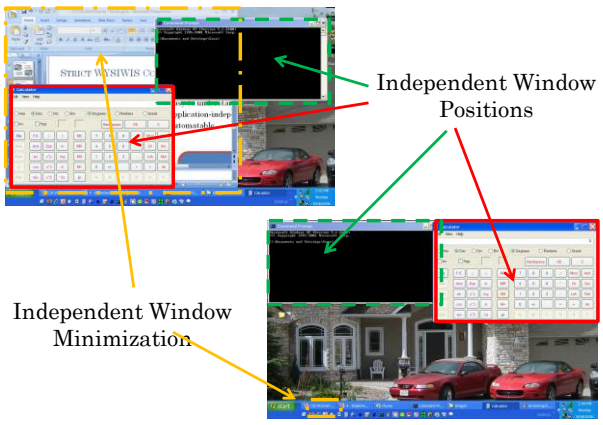


Figure 11 Bradcoast of Syntactic Sugar Events

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



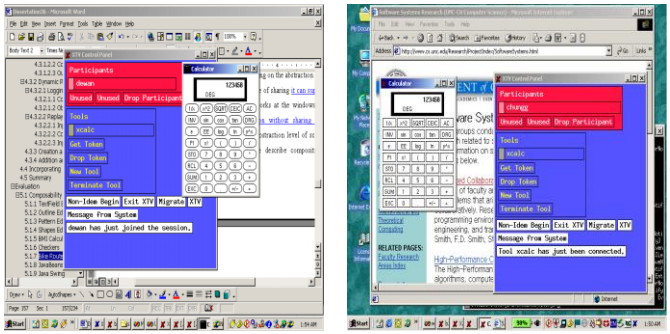


Figure 12 Minimal Window Coupling in XTV

This approach also raises implementation problems when an application uses absolute screen coordinates. In particular it can lead to incorrect display of pop-up menus, as shown in the figure below.

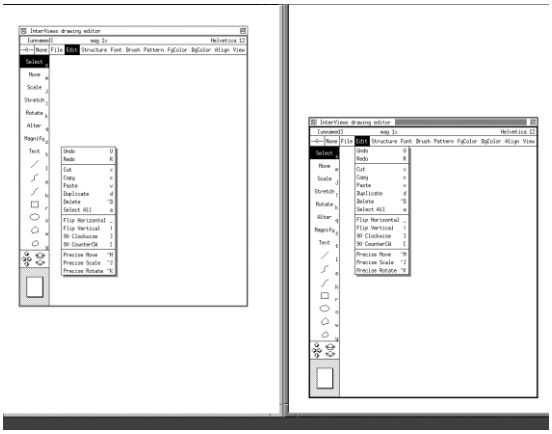


Figure 13 Incorrect Handling of Pop-up Menus

Here, the right user displays a pop-up menu that is shown at the same absolute position on the left user's screen, but at the wrong relative position with respect to the menu item that triggered the pop-up. On the other hand, the sub-windows of the shared window, such as the palette window for adding shapes, are at the correct relative positions on both screens. The reason that the pop-up is displayed incorrectly is that it is not a sub-window of any other window, and is, thus, drawn at absolute coordinates by the application. It is possible for a shared window system to display pop-ups at the same position relative to the window in which the mouse was when the pop-up was triggered by converting coordinates appropriately. The figure below shows an XTV version that does so.

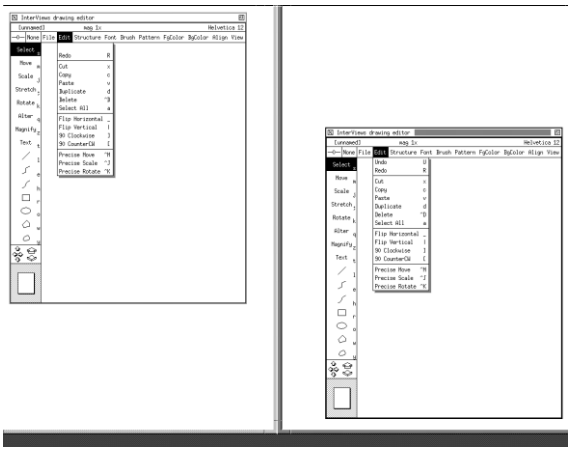


Figure 14 Converting pop-up menu coordinates in the shared window system

Thus, we see above that sharing window manipulation events has its pros and cons. A compromise approach is to position shared windows consistently in a virtual desktop. The virtual desktop is itself a window that can be moved, iconified, and resized. This is the approach implemented by several commercial centralized shared window systems such as NetMeeting, PlaceWare, and Webex. In these systems, the virtual desktop window is a virtualization of the physical desktop of the computer hosting the centralized applications. Thus, the user of this computer does not see the virtual desktop. The figure below illustrates this approach. In this figure the physical desktop of the user on the left is displayed as a virtual desktop window on the screen of the user on the right, which contains only the two shared windows on the physical desktop.

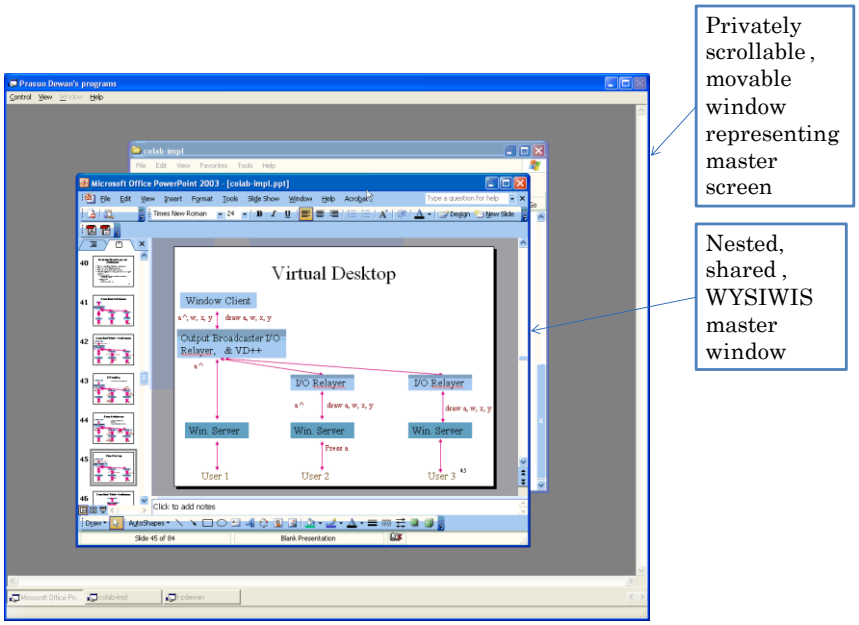


Figure 15 Maximal Window Sharing with Virtuual Desktop

Coupled vs. Uncoupled Expose Regions

Consider now another semantic issue, raised by the figure below. When the master computer occludes a shared window with a private window (left screen), in some systems such as NetMeeting, that area is blacked out on the slave computers. While this ensures that the other users cannot see what the master user is not seeing, it is not clear it is desirable, as the example in the figure demonstrates. Therefore, other systems such as XTV do not black out occluded regions. Sometimes this choice is based on implementation issues. To understand why, we need to understand how occluding is handled.

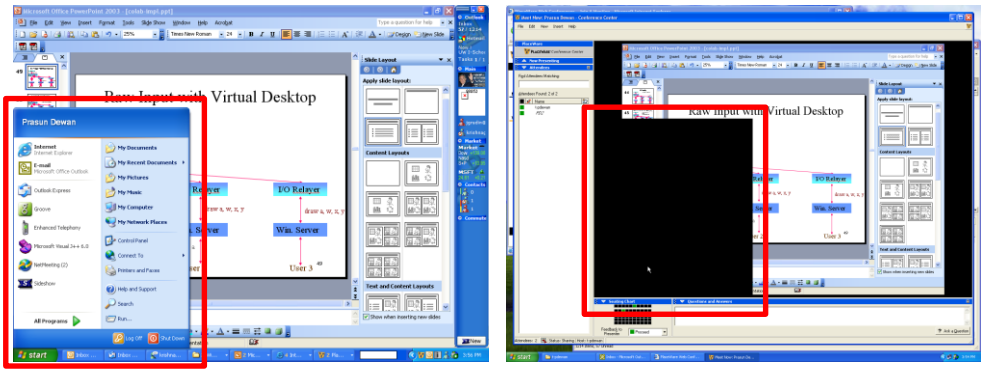


Figure 16 Blacking out occluded regions

In general, when a window is resized or (un)occluded, an expose event is sent to the application, with the rectangles of the window that are exposed; that is, not occluded. The application can respond by redrawing only the exposed regions, or the complete window. In the case where the application draws the complete window, the window system clips the application output; that is, ensures it has no effect on unexposed regions of the window.

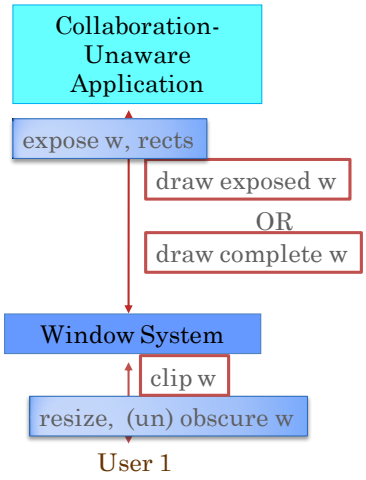


Figure 17 Handling expose events

How occluded regions are shared depends on several factors, including whether the application draws the complete window or only the exposed regions.

Assume first that the application redraws only in the exposed regions. In this case, none of the window systems knows what should be displayed in the unexposed regions. In the master computer, the contents of some other window are shown in these regions. In the slave computers, the areas of the shared window that are unexposed in the master computer may, in fact, be exposed. However, as these window systems do not know what should be displayed in these regions, it is safe to just blacken these areas. A less conservative approach may be to keep track of and show the old contents of the window in these regions. Under this approach, if the regions covered on the host computer change, they will not be shown accurately to the other computers.

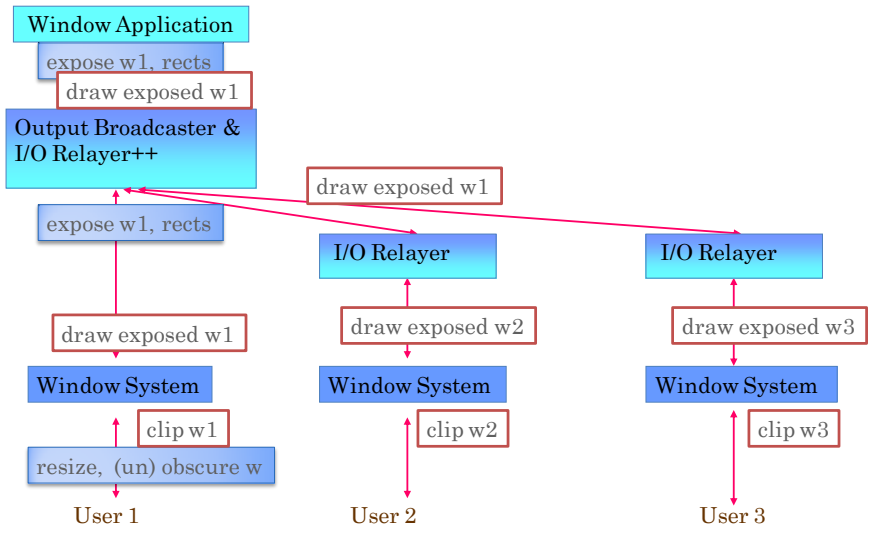


Figure 18 Application draws only exposed regions

When the application redraws the complete window, each window system can simply clip the output according to which parts of the window are exposed in it, as shown in the figure below.

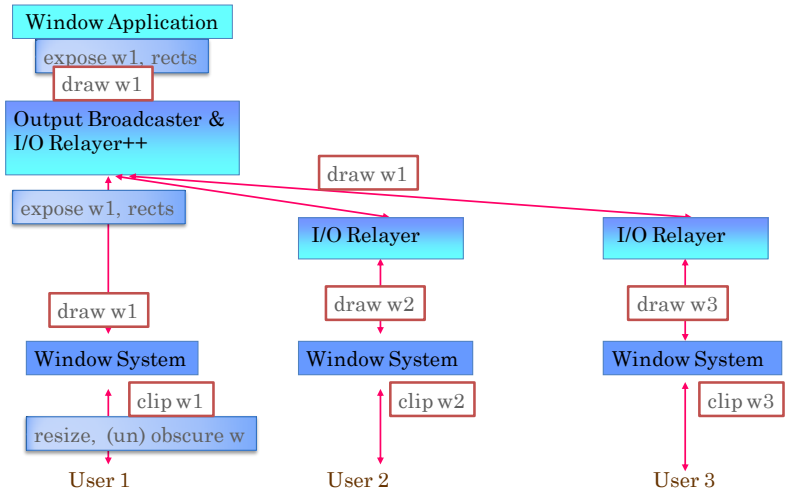


Figure 19 Application draws complete window

Thus, in this case, there are no black holes on slave computers. However, slave computers may see what the master cannot. This may raise privacy issues – a master cannot cover up a private region with a private window to prevent others from seeing it. To address this problem, some systems send expose events from the master window system to all other computers. This allows the proxy running on each computer to clip the output to only those regions that are exposed in the master computer. Thus, more work is needed (broadcasting of expose events) to support coupled expose regions rather than uncoupled expose regions – which is contrary to the impression that the former is an inherent limitation of window sharing.

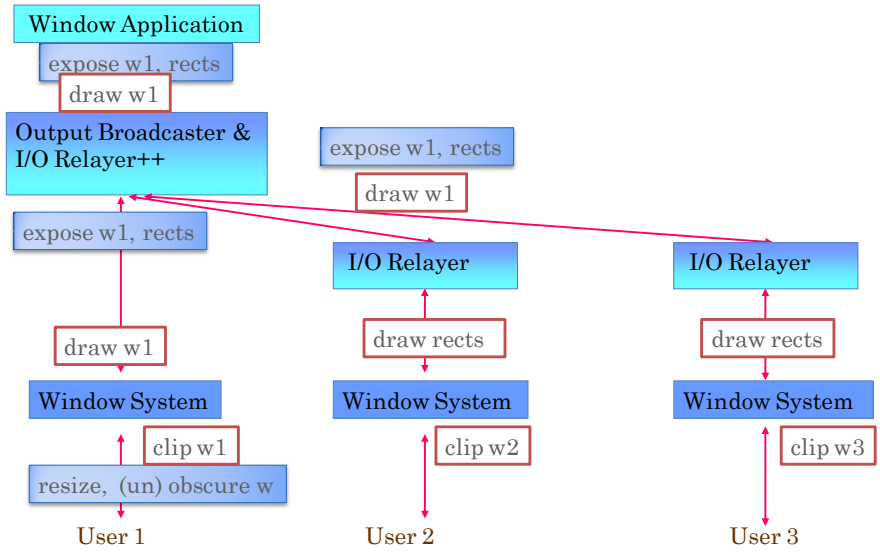


Figure 20 Broadcasting expose events

In general, expose events are not sent from slave computers as the same window may be exposed differently on different computers.

It is not clear if displaying unexposed parts of a window on another computer is really a user-interface problem. If it is not, then we would not need to broadcast the expose events and could send an application the union of the exposed regions on all computers.

Degrees of Window Coupling

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

- window sizes
- window contents.

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

- window positions
- stacking order of windows
- exposed regions

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

Handling Queries

As we have seen above, a window system does more than send keyboard and mouse events and process draw requests. It generates and processes window-manipulation events. In addition, it sends expose events to the application, and maintains application and window state that can be queried by the application. For example, it provides a listFonts() call that allows an application to determine the list of fonts supported by it. The keyboard, mouse, expose, and window manipulation events can be considered as input as they inform the application about user actions and transfer information to it. The draw requests can be considered as output as they give instructions on what to display, and transfer information from the application to the window system. The query calls such as listFonts(), on the other hand, are in between as, like draw requests, they are calls made by the application, and like input events, they transfer information to the application. In general, calls made by the application that return values have these two conflicting properties. How to handle such calls is an issue in shared window systems.

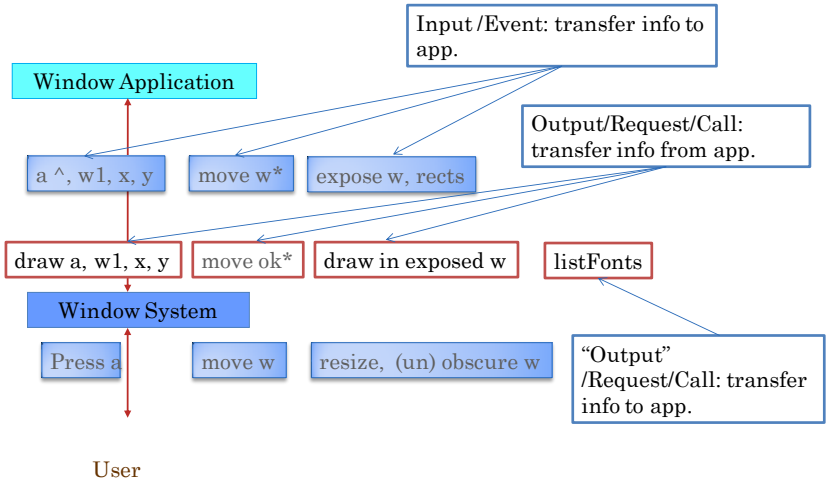


Figure 21 Application Window-System Communication

A query such as listFonts() can be handled in several ways. It can be sent to a single window system (either the master window system or a slave window system). If the specific window system does not matter, it is more efficient to send it to the master window system, as there is no need to send messages on the network or do any translation, as shown below.

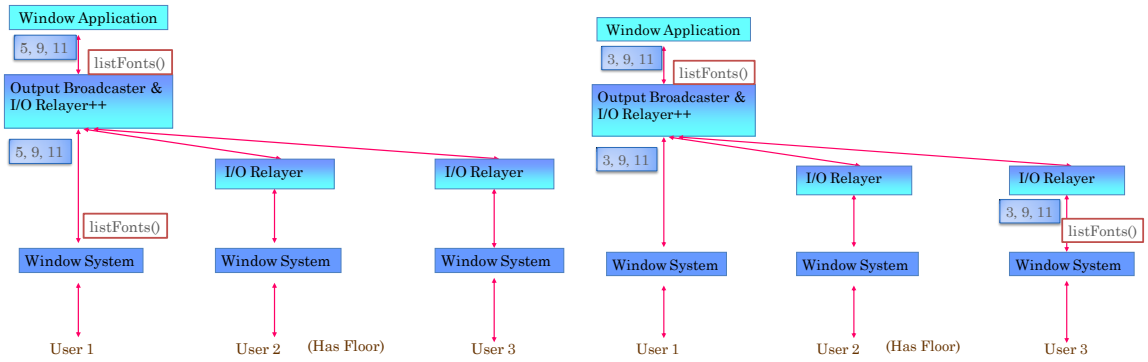


Figure 22 Unicasting to the master or slave window system

An alternative is to broadcast the request to all window systems and do some query-specific composition of the replies. For example, in the case of listFonts(), the intersection of the font identifiers returned by each window system can be returned to the application.

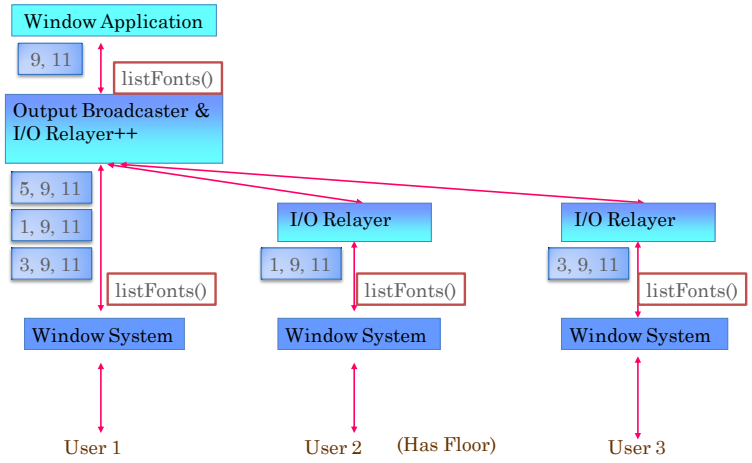


Figure 23 Broadcasting the query to all window systems

Yet another alternative is to send the query to the floor holder. This alternative must be taken for any call that queries the state of the mouse and keyboard, as they are targeted at the computer of the window system of the active user. For example, the call, queryPointer(), which queries the position of the mouse, must be sent to the floor holder.

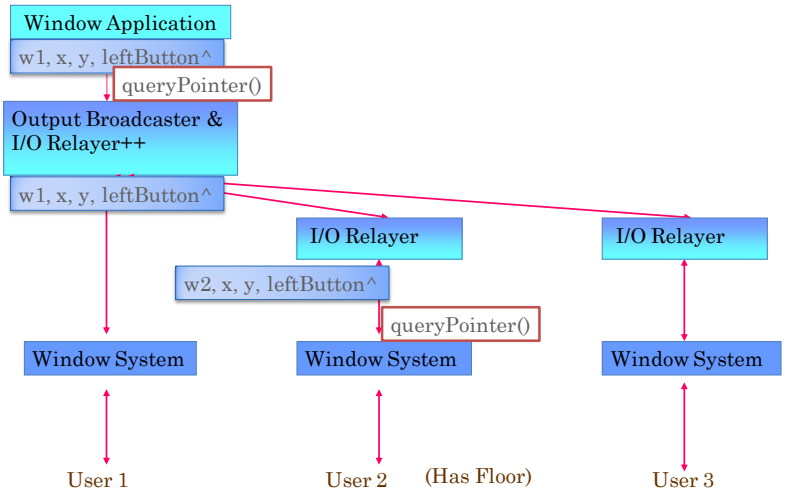


Figure 24 Unicasting calls that query mouse and key state to floor holder

Replicated Window Architecture

So far, we have considered only centralized window sharing architectures. In the replicated architecture, each computer runs a copy of the shared application. Thus, in this architecture, there is no master-slave asymmetry. Each computer is a master and behaves in the same way.

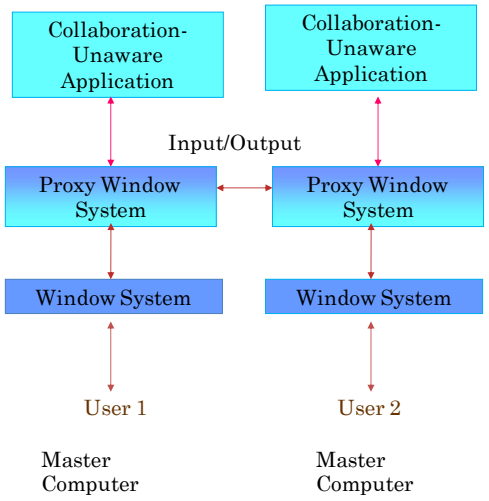


Figure 25 Symmetric Replicated Architecture

In the replicated architecture, each user's application computes its output. To ensure all users see the result of a user's input, the input is broadcast to all other applications via the infrastructure modules. Thus the replicated architecture is a dual of the centralized architecture. In the centralized architecture, output is broadcast, while in the replicated architecture, input is broadcast. In general, as input is produced by humans rather than the computer, the size of the input is smaller than the size of the output.

The figure below shows event processing in the replicated architecture. When user 2 presses the key, 'a', the input event is sent to its replica, which computes its output. In addition, the event is sent to the replicas on other users' computers, which compute the output for their local users.

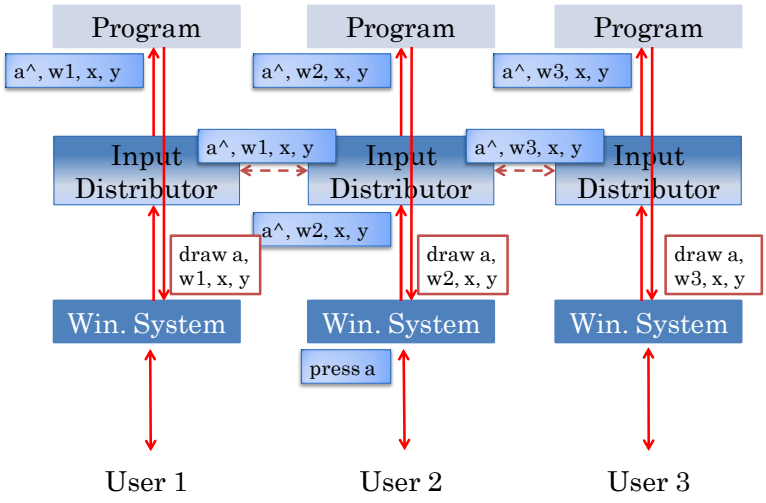


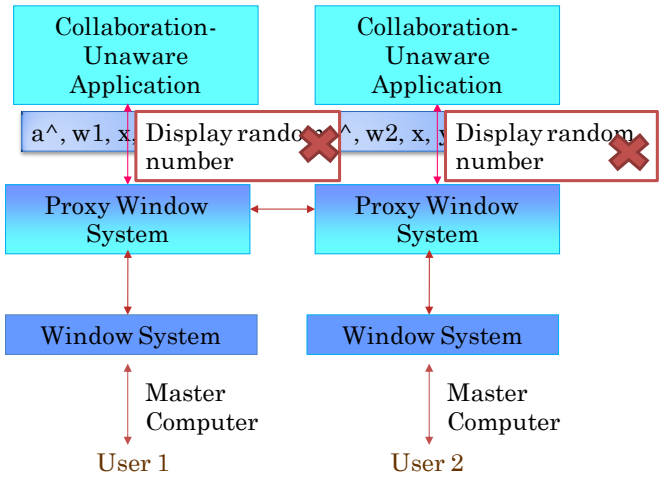
Figure 26 Event flow in the replicated architecture

As in the centralized architecture, we assume that floor control is used to ensure that only one user inputs at one time. Thus, an infrastructure module does not have to check with other modules, before delivering input to its replica. As a result, the active user never has to wait for his input or output to travel to some other network. This is not the case in the centralized architecture when a user on a slave computer inputs – the user must wait for his input to reach the master and the output from the master to reach his computer. We will do a performance-based comparison of the two architectures in more depth later.

However, the replicated architecture has some correctness problems.

Non-Deterministic Operations

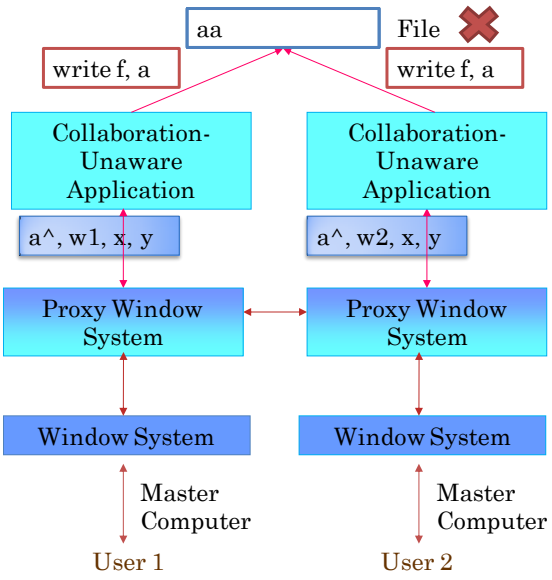
The replicated architecture assumes that if all replicas receive the same input sequence, they will produce the same output. This does not happen for non-deterministic applications, that is, applications whose behavior does not depend entirely on the input sequence. The figure below shows an example of such an application. When the user presses the 'a' key, it displays a random number. In this example, the replicas will show different random numbers to their users, thereby violating our goal of WYSIWIS coupling.



Therefore replicated architectures assume the shared applications are deterministic.

Writing Centralized External Resources

Even if we make this assumption, correctness problems can still occur when an application writes to a centralized resource. As shown in the figure below, the same information gets written multiple times, once by each replica.



It is possible to address the problem of writing to a file by replicating the file on the computers of each user. In this case, the application replicas write to separate file replicas, which are always consistent.

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

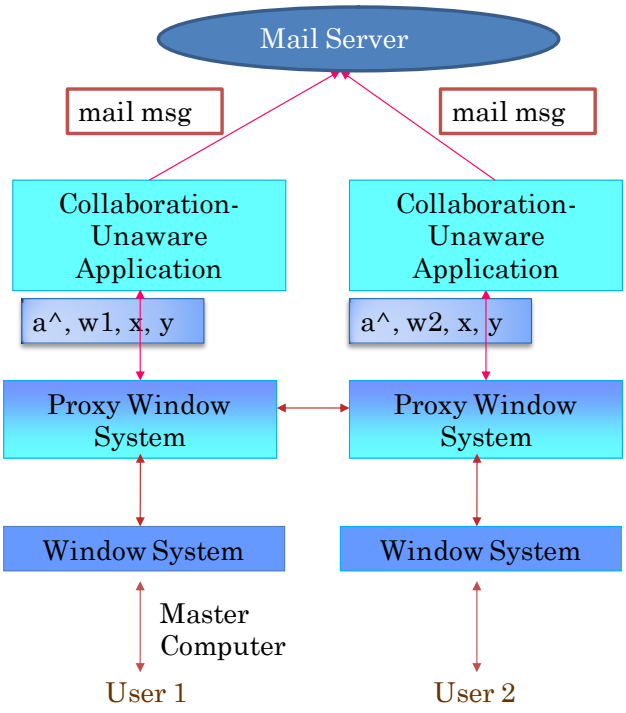


Figure 27 Correctness Problems with Executing a Non-Idempotent Operation

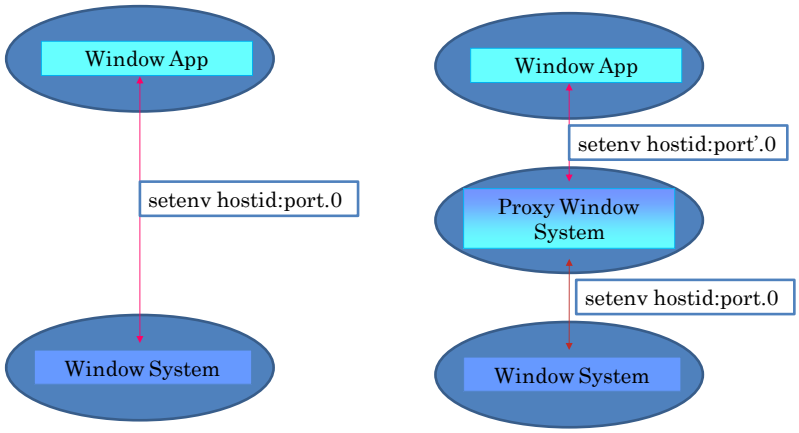
The problem here is that mail or writing to a centralized external resource is an non-idempotent operation, that is, an operation whose multiple executions are not equivalent to a single execution. By

replicating the external resource, we changed the operation – accessing a replicated instead of a central resource - but it is not always possible to do so, as the mail example shows.

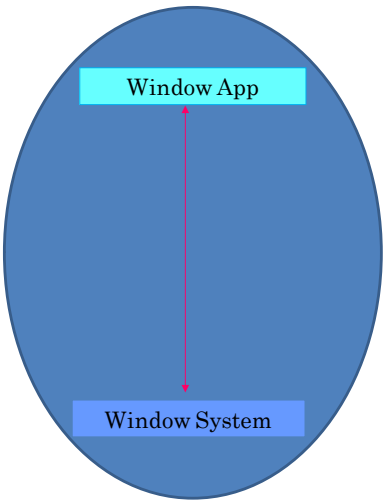
Because of these correctness problems, none of the current commercial systems support replicated window systems even though these systems are more efficient in many scenarios, as mentioned above.

Proxy vs. Listeners and Polling

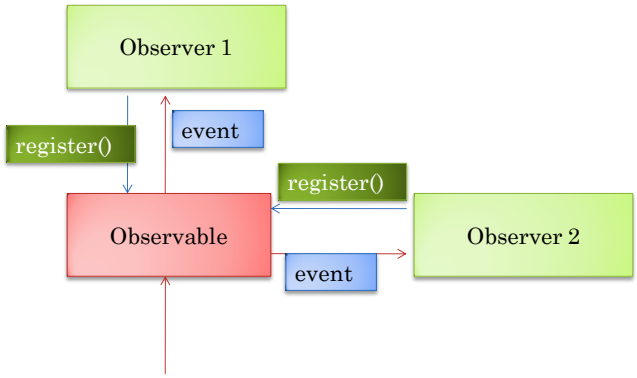
Both the centralized and replicated architectures require insertion of proxies between the window system and application. This is relatively straightforward when the two run in separate processes. To illustrate, consider the X window system. The binding between an application and the X window server is made by setting an environment variable in the application. An environment variable is essentially an argument passed to the application that can also be inherited from the process such as shell that started it. In X, the environment variable HOST specifies the address (host and port) of the X server. A proxy between the application and X server can be inserted by assigning to the HOST variable of the application the address of the proxy, and the HOST variable of the proxy to the actual window server, as shown below.



This approach is more problematic in a window system such as Microsoft Windows, in which the application and window system run in one process.

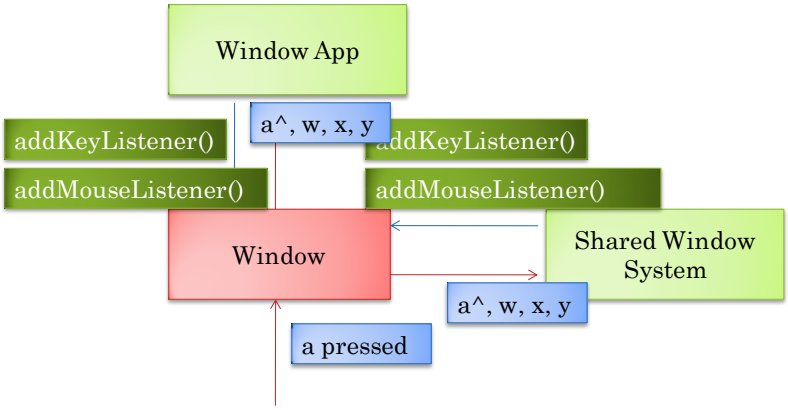


Library proxies are possible but not common. What is more common is a window observable. An observable is an object that sends certain state events to other objects, called observers, that are interested in receiving these events. An observer expressed its interest in receiving some class of events by calling a register method in the observable associated with these events.



Window observables announce user inputs to its observers. For example, an observer that calls addkeyListener() (addMouseListener()) in a window to register its interest in receiving keyboard (mouse) events, receives these events whenever the user interacts with the keyboard (mouse).

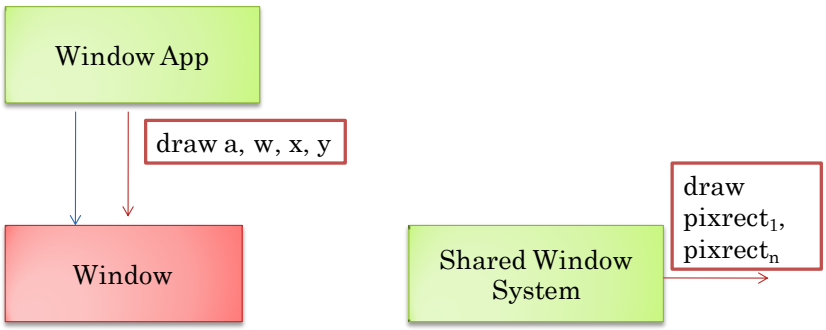
An observable window can be shared by making the shared window system an observer of its keyboard and mouse events, as shown below.



The proxy is no longer a proxy between the master window application and window – thus, it no longer has to relay input to the application and output to the window system. However, it still an intermediary between the slave window systems and the window application. This approach requires a shared window system to enumerate all windows of an application.

While a window announces input events to observers, it does not let observers know about output sent to it by the application. This is consistent with, say, a Java text field, which does not generate notifications when the application calls sets its text. So how does a shared window system output commands to other windows?

An output call makes some change to the framebuffer. The shared window system can indirectly determine the nature of output by observing the framebuffer. As it does not receive output notifications, it can periodically poll those portions of the framebuffer that contains the contents of the windows of the shared application, and send diffs since the last poll to each slave computer, which can then apply them to its windows. The diffs essentially contain a series of pixel rectangles in the application windows that have changed since the last poll.



Thanks to Sami for his corrections