Collaborative Sharing of Windows between MacOS X, the X Window System and Windows[*]

Daniel Stødle
daniels@stud.cs.uit.no

Otto J. Anshus
otto@cs.uit.no

John Markus Bjørndalen
johnm@cs.uit.no

Abstract

This paper investigates how one best can share windows between many different computers in a collaborative, cross platform environment. Current collaborative solutions offering users shared application access are limited in that they either share an entire desktop, or that sharing is built into the collaborative application. We present an architecture of a system allowing windows on MacOS X to be shared with computers running the X11 Window System and Microsoft Windows, with windows either being pushed to a shared surface, or pulled from a sharing user's computer. The prototype implementation supports sharing MacOS X windows with X11 based systems. The windows are shared by sharing their pixel representation. Remote users can provide input to the shared windows. We show that the number of FPS observed by each user degrades linearly with the number of users for constantly changing windows, with our experiments ranging from 1 to 4 users. We also show that the event latency for fairly static windows averages at 10 ms.

1 Introduction

There are several approaches to sharing information with others in a group. A simple way would be to send an e-mail containing a document, picture or other data to the entire group. While simple, its usefulness is limited in several ways, including no simple way of merging the individually updated copies of the document again, and forcing each user to individually manage local copies. Fundamentally, the users share no display, and this will limit interactive group interactions.

A more complicated approach uses a remote desktop system to share one user's desktop with the rest of the group. This allows the whole group to see the same desktop, and may enable the other members of the group to provide input to the desktop in question, and thus create a more interactive collaborative environment. However, the whole group shares the same desktop, making this a too coarse grained solution limiting flexibility and protection between users[1]. In addition, the bandwidth requirements to share an entire display with a room full of computers can put a severe stress on the local network. Other

[*]This work has been supported in part by the NFR funded project No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices.

[1]The user sharing a desktop can have personal data visible on the display, for instance, and doesn't want to exit all her personal applications just to share some discoveries with others.

approaches to collaboration are based on applications supporting simultaneous multi-user access and use, including instant messaging, multi-peer A/V conferencing and “shared whiteboard” solutions.

A finer grained modification of sharing the whole desktop is to allow for sharing of individual windows instead of the whole desktop. Existing systems supporting similar functionality are Microsoft NetMeeting [1] and various X11-based sharing tools [2], [3], though these are limited in their platform dependence and the X11-based solutions are plagued by compatibility problems.

We have considered two scenarios for sharing windows:

- Making a window’s contents available to other members of the group by making it visible on a shared surface such as a display wall, possibly allowing users of the shared surface the ability to interact with the window and modify its state
- Making a window’s content available on each user’s local desktop, and possibly giving users the ability to interact with the window and modify its state

The requirements and limitations of each scenario will be analyzed, and possible solutions presented, along with the architecture and prototype implementation of a window sharing system that allows sharing of windows between different window systems and computer platforms.

2 Requirements

The primary goals for our window sharing system are to 1) make it non-invasive by not needing changes done to the window servers, operating system, or applications, 2) have it scale well and provide good performance when the number of users and shared windows grow, and 3) make it easy to set up and use. The system must also work across different window systems and computer platforms, and it should run without needing special privileges.

Sharing windows presents a number of key design issues:

1. Should sharing a window be a sender- or receiver initiated action? I.e., should the sender “push” a window to others, or should the receiver(s) “pull” the windows they desire to look at or interact with?
2. How can a shared window be made portable across many different platforms?

The answers to these questions will vary depending on the setting where the window sharing system is to be used. For instance, pushing windows can be natural when users “push” their windows to a display wall. A pull-model may be more fitting when there are many users wanting to share their windows with other users, but not necessarily with *every* user. In this case, users offer a number of windows, and the other users “pull” windows from the sharing users.

The requirements for sharing a window have several other aspects than the ones mentioned above. Access control and floor management are two key areas that this paper doesn’t explore further, posing questions such as how access to shared windows is controlled, and how input from several users to one shared window is handled.

Pushing and pulling windows

The requirements for pushing and pulling a window mainly differ from each other in their interface. Pushing a window requires the user to specify where the window should be pushed to, whereas pulling a window needs only present a choice of the various shared windows available. On an implementation level, this comes down to discovering available computers with shared windows, and then either accessing or sharing to one or more of the discovered computers.

Portability

Sharing a window between desktops requires examining the mechanism by which the content of a window can be shared. The two approaches are sharing pixels, and sharing drawing operations. Most windowing systems work by providing their client applications with a number of drawing operations. Using these operations, they can compose their user interface. As an example, an application can tell the window server that it wants to draw a line starting at point A, ending at point B, having some thickness and some color. The window server will then modify the pixels visible on the display to draw the line, possibly performing other tasks such as clipping and applying translucency.

The window sharing system must support many different window systems, all with different (though similar) drawing operations. If the source code to the window systems on MacOS X and Windows was available, it would be possible to create a proxy server that could translate native drawing operations into some platform independent format by intercepting the requests before they reach the native window server.

The simplest way to platform independence is achieved by just exporting the actual pixels resulting from the drawing operations to the receiving clients.

3 Architecture

The window sharing system consists of four components: Two platform dependent pieces called `WShare` and `WClient`, and two platform independent pieces called `RClient` and `RShare`. The letters refer to Window and Resource, respectively, and their relationship is displayed in Figure 1.

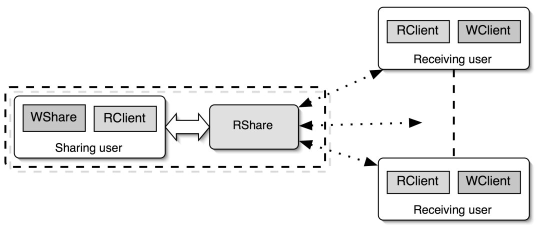


Figure 1: Overall architecture of the window sharing system.

The two platform independent components, which will be referred to as the base layer, deal with the low-level details of sharing resources. Instead of restricting the base layer to merely share windows, a broader abstraction was chosen to facilitate sharing other items, such as cursors and keyboards, as well as more obvious items like disks or files, at a later time.

The base layer consists of a resource server and code that facilitates interfacing with the resource server. It supports an infrastructure for publishing resources, and exports a simple interface to allow other applications to subscribe to the shared resources. It also facilitates message passing between the application sharing a resource, and the applications subscribing to a shared resource. It does not concern itself with details such as what protocol or format is being used to exchange messages. Messages are merely regarded as a stream of bytes from the resource server's point of view, having some length and enough information so as to allow the resource server to properly route the message to where it is supposed to go. The base layer also handles server advertisements and discovery, to ease window sharing on local area networks.

The `RShare` component acts as the resource server in the figures. The resource server can run as a standalone application on any computer, but it is also possible for the `RShare` component to be tightly integrated with the `RClient` component, as alluded to in Figure 1. The primary reason for this tighter integration between the two components is to save bandwidth, thus improving performance. Allowing the resource server to run in either mode is important in order to realize the two window sharing scenarios: Pushing a window to a shared surface, or pulling windows from different users onto ones own desktop. The corresponding deployments are shown in Figure 2.

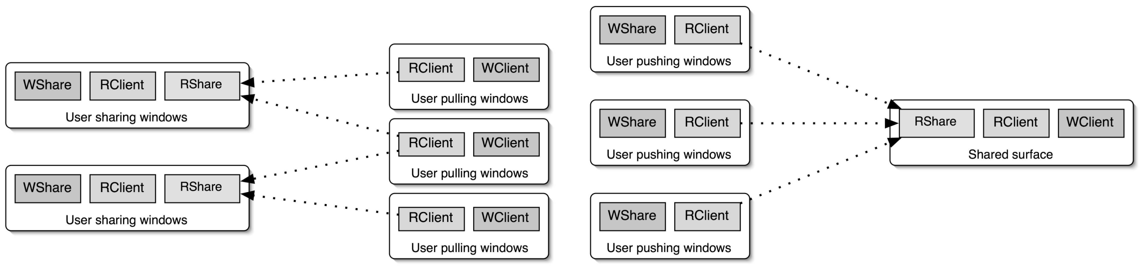


Figure 2: Pulling (left) and pushing (right) windows.

`WShare` and `WClient` are responsible for handling the platform dependent issues when sharing windows. They define their own protocol for exchanging updates to a window's contents, as well as providing input to a shared window. In addition, `WShare` must supply the user with an interface for selecting windows to share, and also provide a steady stream of updates to the subscribed users as they become available. `WClient` has less of a responsibility - all it must do is bring up a representation of a remotely shared window, and then accept and forward input to it to the resource server. It may also need to support an interface for locating and subscribing to shared windows, depending on the context in which it is used.

4 Window sharing protocol

The approach taken in the prototype implementation to share windows was to share the raw pixels. This has the advantage of being completely portable, and requires little less than a suitable display to show the pixels to the user. It also has some disadvantages: Shared windows will not be able to take advantage of the possibly greater resolution of a large shared surface, or adapt gracefully to a lower resolution display. In addition, the bandwidth requirements for a pixel based solution are usually much greater than those of a protocol-based solution.

The window sharing protocol is very similar to the one employed by VNC [4], but not identical. The only reason for this was to accelerate development of the prototype

- developing a new protocol appeared to be simpler than re-implementing the remote framebuffer protocol utilized by VNC or integrating sources from one of the open source VNC clones. It was also simpler to integrate the new protocol with the resource sharing framework developed as part of the prototype.

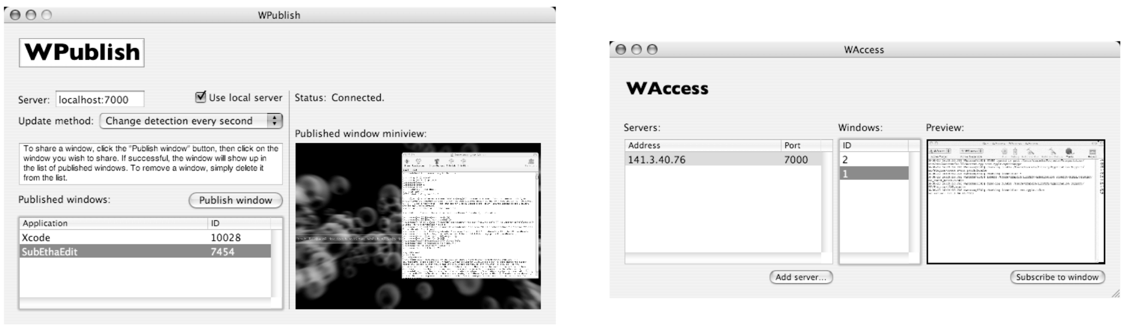
Once the user has selected a window to share, the resource will be published to the resource server. Once one or more users bind to the shared window, updates will start to accumulate at the server at some semi-fixed rate (usually one update per second or more). The server provides the different clients with updates as they are requested by the clients. Each update packet contains encoding information, area covered by the update, and the compressed pixels. Clients can provide input to the window by sending "post event" messages, which are then forwarded by the `WShare` component to the window in question.

When a user binds to a shared window, the client will first request the size of the window. Once this is done, it will request a refresh. The server will send a refresh to the client, and start accumulating invalid areas for that client. Once the client has received the entire refresh, it will indicate its willingness to receive further updates by sending another refresh request. The server will respond with new refresh messages as changed pixels become available.

5 Implementation

This section describes the prototype implementation of the window sharing system, focusing on the MacOS X implementation of the `WShare` and `WClient` components.

The MacOS X implementation of the window sharing components consists of two applications called WPublish and WAccess, respectively supporting sharing windows and accessing shared windows. Screenshots of the applications can be seen in Figure 3.



Server: localhost:7000 [✓] Use local server
Status: Connected.
Update method: Change detection every second

To share a window, click the "Publish window" button, then click on the window you wish to share. If successful, the window will show up in the list of published windows. To remove a window, simply delete it from the list.

Published windows:

Application	ID
Xcode	10028
SubEthaEdit	7454

Servers:

Address	Port
141.3.40.76	7000

Windows:

ID
2
1

Figure 3: Sharing (left) and accessing (right) shared windows on MacOS X.

In order to understand the MacOS X implementation of `WShare`, it is necessary to first get a grasp on how MacOS X handles windows and passes events from the user to applications. On MacOS X, "everything" is a window. The menubar, menus, submenus, palettes, tooltips, icons on the desktop – they are all windows. In addition, the concept of an application is different from what one might be used to from an X11 or Microsoft Windows environment.

An application consists of a menubar, which has commands that act upon the currently active window, as well as any number of windows and palettes. The menubar is not associated with any one window, and is always positioned at the top of the screen. There is usually no “root” window (i.e., no equivalent to the Windows Multi-Document Interface, where the root window is per-application, or the X11 root window which serves as the parent window to all other windows) - all windows are independent of all other windows, and only occasionally form a hierarchy.

The underlying implementation is also slightly different from the usual implementation on other platforms, in that nearly every window has its own backbuffer. When an application makes changes to its windows, the changes will first be rendered to the backbuffer, before the window server composites the changes to screen, taking into account overlapping, possibly translucent windows, windows layered below in case the window being rendered is also translucent, and syncing to the VBL. This gives MacOS X an advantage over other window systems, at least when it comes to sharing windows with others: A window can be shared, and kept up-to-date, without actually being visible on screen.

In addition, an application on MacOS X can be written targeting one of three different event models: Cocoa events, Carbon events and X11 events. This adds to the complexity of a reliable window sharing implementation, as MacOS X’s window server doesn’t allow posting of events to specific windows. In fact, it doesn’t even present an API for posting events to specific applications. This implies that although it is possible to share the window’s graphical contents, allowing other users to interact with the windows may prove difficult. On the other hand, sharing the entire desktop is comparatively quite simple; functions exist that allow both the graphical contents of the desktop to be retrieved as well as “global” posting of mouse and keyboard events.

While MacOS X makes it easy to detect when and where updates occur on the display, using this API to implement change detection for WPublish doesn’t work. The reason is that windows may be obscured, or not even visible on screen, yet still receive updates to their backbuffer. The API provided in MacOS X only reports changes that happen on the display - changes to backbuffers are *not* reported. WPublish solves this by keeping its own copy of the window, and comparing the copy to the actual backbuffer once every second or more. WPublish allows the frequency to be increased by the user, at the cost of more CPU power being used to share the windows. It also gives the option of simply transferring the entire window at some frequency, at the cost of potentially much greater bandwidth requirements. Even so, depending on the contents of the window, and how often the window actually receives updates, the second approach may be favourable. An example of this would be sharing a window displaying a movie, where the contents would change all the time anyway.

The WAccess implementation is fairly straightforward, giving the user the ability to subscribe to windows from different resource servers. It uses the base layer’s discovery mechanism to detect servers on the local subnet, and also provides the user with the ability to enter a specific server address. The user can see a preview of a window before subscribing to it.

The platform independent WClient

In order to test the window sharing on more than one platform, a platform independent `WClient` was written that utilizes SDL [5] to do its input and display handling. It is a bare-bones client that simply takes a server address and resource ID as arguments, and then attempts to subscribe to the specified window. The benefit from using it is that it, by virtue of both SDL and the base layer code being portable, runs with very little effort on many different platforms.

Pushing windows

Pushing windows was accomplished by writing a combined resource server and window client, referred to as the push-receiver. Once the server has started up, the window client will open a connection to the resource server, waiting for announcements of the availability of new window resources. As soon as a window resource is detected, it forks off an instance of the platform independent `WClient`, instructing it to connect to the local server and subscribe to the newly published window resource.

Performance considerations

In order to make the prototype implementation perform well, the window refresh messages have support for various compression methods. The compression that currently is used is a simple run-length encoding (RLE) algorithm, that converts identical runs of pixels into the pair {length, pixel value} and sends this instead. The encoding supports both 16- and 32-bit pixel depths.

Using RLE encoding has great advantages when the windows that are shared contain many large areas of the same color (such as the white background in a text editing window), but does not perform well in cases where the window contains very diverse pixels (such as a movie being played, or a digital picture).

The next optimization is called local-mode, and allows the `RClient` component to communicate directly with the `RShare` server by placing messages directly on the server's input queue, instead of first transferring them over their communications socket. This optimization is only enabled when the server and client run as part of the same process, and is initiated explicitly by the client code. The optimization only allows messages from the client to the server, not the other way around. This is to protect the server from accessing invalid memory, caused by a rogue client attempting to enter local-mode. It is usually used by the sharing peer, as this is the case where the resource server and resource client most frequently run as part of the same process, and the amount of traffic from client to server is the greatest.

Finally change detection is used to minimize the number of pixels that need to be sent over the network.

6 Benchmarks

The benchmarks presented here measures time, frames per second (FPS) and bandwidth (MB/s) to gauge the performance of the system. The testing setup consisted of a PowerBook 1.25 GHz sharing windows to two PCs with Intel P4 processors at 3 GHz running Fedora Core II, connected via a local, 100 MBit ethernet network. The measured quantities are frames per second at each subscriber, as well as the bandwidth usage from the sharing computer. For all the tests, a target FPS of 15 was used[2].

The first benchmark measured FPS and bandwidth usage for sharing one window with the two PCs, starting at one subscriber and going up to four. The window was a QuickTime window sized at 400x401 pixels, playing a music video[3]. The pixel depth was 32 bits, and the tests lasted for approximately 60 seconds each. The tests were performed with the constant background load produced by playing the movie. This background load

[2]This setting causes a timer to fire at most 15 times per second in the WPublish application.

[3]QuickTime Player was instructed to composite frames using the "transparent" transfer routine, which prevents it from performing hardware-assisted overlay blits. If this had not been done, the only thing the subscribers would see would be an empty QuickTime window.

is not insignificant - measurements indicated that QuickTime Player regularly consumed more than 50% of the available CPU time during sharing.

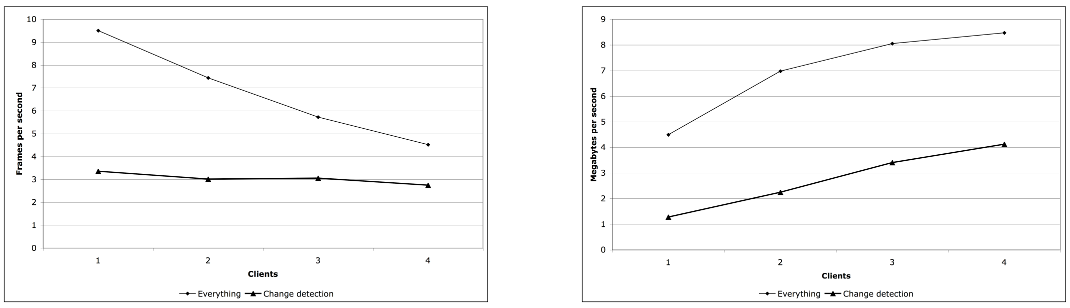


Figure 4: FPS (left) and bandwidth usage (right) for sharing one window with multiple subscribers.

The first graph in Figure 4 shows how the sharing software performs when using either the change detection based update method, or the plain "send everything" approach as the number of subscribers increases. The change detection approach is clearly limited by the amount of computation that needs to be performed, whereas the "send everything" approach fares better, declining slightly as the number of subscribers grow – a problem that is not visible in the change detection results, as the limiting factor here is how fast frames can be differentiated, and the speed at which change rectangles can be composed and broadcast. Change detection does not do well in this benchmark, which isn't surprising considering the constantly-changing nature of the window.

The second graph shows how the bandwidth usage increases with the number of subscribers, using either change detection or sending everything for the same QuickTime window. The growth is practically linear when using change detection. Sending everything appears to utilize the available bandwidth well, though the available bandwidth is not maxed out.

A second figure related to bandwidth is the approximate bandwidth savings provided by the change detection update method. One frame using the send-everything approach averaged 0.47 MB[4], which is about 25% more than frames using change detection, which averaged 0.37 MB. These savings must be seen in light of the radically lower FPS provided by the change detection, however.

The second benchmark examined how performance scales when the PowerBook is sharing more than one window, ranging from one to four. Each window had one client, running on either of the two PCs. Two of the windows contained fairly static contents, whereas the other two windows were QuickTime movies. The two static windows were an Excel spreadsheet and a text editing document, sized at 792x582 and 713x646 pixels respectively. The first movie remained the same as in the previous benchmark, while the second movie was sized at 352x353 pixels. Sharing is performed using either change detection or sending everything at a target of 15 FPS, and the windows were shared starting with the text document, then the Excel window, the first movie window and then

[4]The reason this is lower than the expected frame size of 0.61 MB (400x401 pixels at 4 bytes each) is mainly due to the RLE compression.

the second, smaller movie window. During all the tests, the various static windows were interacted with, in order to provoke updates.

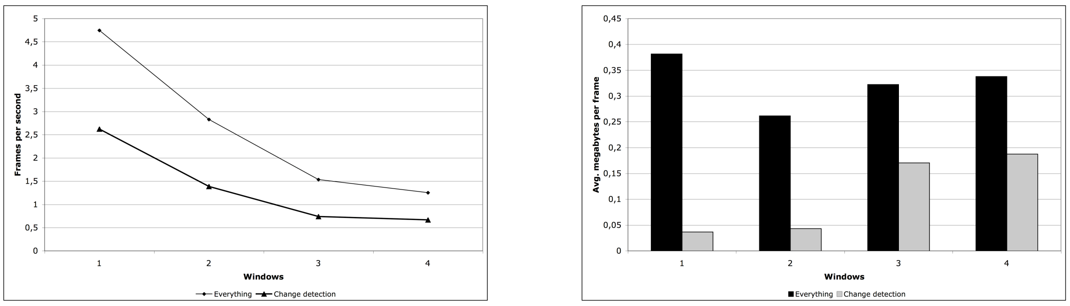


Figure 5: FPS (left) and average frame size (right) for sharing one to four different windows.

The graphs in Figure 5 show how performance deteriorates as more windows are being shared, and the average size of one frame. One question arising from the first graph is why the number of FPS is lower than it was for the first benchmark, the answer being that the first window shared in this benchmark had a much larger area than the window shared in the first benchmark, a factor that impacts both change detection and sending everything.

The second graph shows the benefit from using change detection. Sharing one window with change detection, compared with sending everything, has a massive impact on bandwidth bandwidth usage, with a factor of 10 separating the two approaches. This factor is reduced as more windows are shared, and changes notably as the movie windows are shared. Note, however, how the frame size remains lower than what was observed in the first benchmark, where change detection helped very little (keep in mind that the bars in the graphs are the average frame sizes for all the shared windows).

Change detection manages to keep the overhead from sharing the two large windows low, while not providing any great benefit to the movie windows. Unfortunately, change detection requires so much time to complete when sharing four windows that it devastates the number of FPS the sharer is capable of delivering, while also being in competition with two playing movies for processor time. A good way to improve on this would be to give the user the ability to assign different update modes for different windows.

When change detection isn't enabled, the drops in framerate are caused mainly by the work needed to constantly copy the windows and push them onto the network. Since the windows in this case are much bigger, the total amount of data that needs to be transferred is much greater than what was needed for the first benchmark, explaining the lower framerates. The overhead caused by getting the window's pixels from the operating system is also non-trivial, and greatly impacts the number of FPS the sharer is capable of delivering.

Finally, it is interesting to examine how much the remote windows lag behind their real counterparts, when a remote user is interacting with a shared window. In order to get an approximate estimate of the latency from action on the remote computer, until the result is visible for the remote user, the following strategy was used: A text editing window, sized at 431x347 pixels, was shared using change detection with a target FPS of 15. Since change detection was used, no refresh messages are sent unless there is a change in the window. Thus, when text is entered remotely, the refresh messages resulting from this are taken to be an implicit acknowledgement of the events posted from the remote computer. This does not produce an exact estimate in general, but works well for a window shared using change detection, and where "unprovoked" changes that could contaminate the measurements rarely or never occur.

The period of time measured is the time between the moment when the event is queued remotely, and the moment when the next refresh message has been processed at the remote end. Importantly, this includes the time it takes to display the refresh message, an important factor for interactivity. Measurements that are more than 10 times the current average are discarded. Assuming the change detection is able to run 15 times pr second, the maximum theoretical latency is (ideally) 66 ms (with no overhead from the network, etc), and the average can be expected to lie at 33 ms.

The results from this test were as expected from the interactive performance observed during the test, where the text appeared "instantaneously" as it was entered remotely. The average measured latency was 10 ms, with the minimum observed latency at 1.2 ms, and the maximum observed latency at 35.4 ms. The maximum *filtered* sample was 944 ms, which is clearly an unreasonable value. The results compare favourably with the theoretical figures. The reason why the measured average is better than the theoretical average may be due to the filtering being performed. When filtering was disabled, the average ended up at 35 ms, which is more in tune with the expected average, but the measurements appear to be skewed by a few samples that end up having ridiculously high latency values. This may be an indication that the filtering is flawed, or that the method used to measure the latency has problems. The spikes can also be explained by intermittent load on one of the computers.

In conclusion, the interactive performance is good, at least for tasks that are not too graphics-intensive. The send everything approach seems to be the best solution when bandwidth is of no concern, whereas change detection works better when bandwidth usage needs to be limited. The ultimate speed-limiting factor, however, is how fast the publisher is able to detect and send updates, and the speed at which remote peers can request refreshes.

7 Related work

There already exist a number of solutions for sharing either an entire desktop or individual applications with other users. VNC [4] encompasses a number of desktop sharing applications, all based upon the VNC protocol, including a number of free, open source versions. VNC is limited in that it only shares pixels visible on the display. This contrasts with the window sharing prototype in that it doesn't require the shared window to be visible in order to share it. Other VNC-like solutions include Apple and Microsoft's own OS-dependent implementations.

QuiX (later renamed to MaX) [6] is an application that works with older versions of MacOS to facilitate application sharing between what was then known as System 7 and the X Window System. It serves as an example showing that developing a protocol-based sharing solution has merit, but requires large amounts of work to complete, as it translates QuickDraw[5] drawing operations into X protocol drawing operations. This is clearly different from the approach taken by the prototype, which shares pixels, not drawing operations. Similarly, Microsoft's Terminal Services performs desktop sharing

[5]The low-level drawing toolkit supported by legacy MacOS.

by sharing GDI drawing operations, though without the difficulty of translating between platforms.

Microsoft's Messenger and their older NetMeeting software also support application sharing, but only work between computers running the Windows operating system. They differ from the prototype in that they share an entire application, and in their dependence on the Windows OS for sharing. The window sharing prototype does not share an entire application, only windows belonging to an application - this is an important difference, as it implies that while a shared application can pop up a dialog related to a window, this dialog will only pop up on the local computer's screen, not on any remote users' screens. Sharing a single window in this way can be considered either an inherent weakness or an inherent strength, depending on one's point of view.

For the X Window System, there exist a number of packages that allow one to share X11 applications. The packages usually do this either by window replication or window migration, and the most common implementation technique uses a pseudo-server sitting between the "host" X server and the X clients that are to be shared, although other possibilities exist. Sharing windows or applications on X at the protocol level presents a number of problems [7], as the X11 protocol wasn't written with multiple servers in mind. Most of these are related to different server characteristics (depth, resolution, etc.), and problems translating sequence numbers.

There have been many attempts at writing applications that do X multiplexing (in essence, application sharing for X11) [8]. Two examples here are Xplexer [9] and XTV [3]. These applications both attempt to replicate the windows on X11 across multiple different X servers. Xmove [2] is an application that uses the second approach of window migration. It uses a pseudo-server to record information that is later used when a request is received to migrate an application's windows from one X server to another. The approach works fairly well, but is limited in that it lacks compatibility with some X applications using more or less esoteric X extensions, and is failure prone when X servers supporting different extension sets are used. Xmove differs from the window sharing prototype in its dependence on X11, and the fact that it doesn't share, but migrates, windows. It doesn't provide the granularity that the prototype offers either, as an entire application is moved, not individual windows - a problem shared with most other X multiplexing solutions as well.

8 Limitations and future work

While the prototype implementation works well, it currently only allows sharing of windows from MacOS X. In order to be complete, implementations should be written for the X Window System and Microsoft Windows environments as well. The platform independent `WClient` works well on X11, but is lacking in ease of use and in that it doesn't have a `WShare` counterpart. Minor changes to the networking code will be needed in order to make the client run on Windows.

A rather big limitation with the current MacOS X implementation is that it does not allow for posting events to all windows - the set of "writable" windows is limited to applications written in Cocoa. While this is a problem, it is difficult to fix without knowing in detail how the window server sends events to applications. A complete implementation of this has been left for future work. A second limitation is that windows utilizing an overlay to have the video hardware composite their contents for them have a useless backbuffer filled with the overlay color, giving unexpected results on the receiving ends. This is especially a problem when sharing movies, as the technique is most commonly used in these applications.

9 Conclusions

A working window sharing prototype has been developed, that allows MacOS X users to share their windows with other users on different platforms. Users may interact with the shared windows, with some limitations, and the interface for sharing windows is intuitive and easy to use. Windows can both be pushed to a large shared surface, or pulled by users wanting to see the shared windows, depending on how the system has been deployed. The shared window does not need to be visible on the sharing user's screen in order for others to see and interact with it.

The performance of the system is limited by how often the sharing peers do change detection on the shared windows, the available processing power and on how capable the underlying network is. The user experience when operating a remote window is excellent, with response times generally in the 10-35 ms range, depending on the size of the shared window, and how often change detection is performed.

The window sharing system as it stands now works well, with an underlying architecture that has good room for further expansion, both for sharing windows and other resources. This should provide a good foundation for further enhancements and research.

10 Acknowledgements

Thanks to Kai Li and Grant Wallace, for their ideas, suggestions, discussions and support. Thanks also go out to Karl Gunnar Aarsæther for giving an outside perspective on the system and criticizing the implementation when it wasn't working properly.

References

[1] Microsoft Netmeeting. http://www.microsoft.com/windows/NetMeeting/.

[2] Ethan Solomita, James Kempf, and Dan Duchamp. XMove: A pseudoserver for X window movement. *The X Resource*, 11(1):143–170, 1994.

[3] H. Abdel-Wahab and M. Feit. XTV: A framework for sharing X window clients in remote synchrounous collaboration. IEEE Tricomm, April 1991.

[4] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1), January/February 1998.

[5] Simple directmedia layer. http://www.libsdl.org/.

[6] Klaus H. Wolf, Konrad Froitzheim, and Peter Schulthess. Multimedia application sharing in a heterogeneous environment. In *ACM Multimedia 95 - Electronic Proceedings*, June 5.-9. 1995. http://www-vs.informatik.uni-ulm.de:81/Papers/ACM95/QM.html.

[7] Hussein Abdel-Wahab and Kevin Jeffay. Issues, problems and solutions in sharing X clients on multiple displays.

[8] J.E. Baldeschwieler, T. Gutekunst, and B. Plattner. A survey of X protocol multiplexors. *ACM Computer Communication Review*, 23(1), April 1993.

[9] W. Minenko. The application sharing technology. *The X Advisory*, 1(1), June 1995.