Term Paper Spring 2004

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

Daniel Stødle
daniels@stud.cs.uit.no

5th October 2004

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 the sharing is built into the collaborative application, limiting their usefulness due to either sharing too much or too little. Platform dependence is another hurdle limiting some of these applications. In this paper, an architecture and prototype implementation of a system for sharing individual windows between different window systems is presented, allowing windows on MacOS X to be shared with computers running the X Window System or Windows. The windows are shared by sharing their pixel representation. We conclude by offering some performance benchmarks and suggestions for further research.

1 Introduction

In a collaborative environment, one often needs to share information with others in a group. There are many approaches to sharing this information, ranging from the simple to the highly complex. A simple way of sharing information would be to send an e-mail containing a document, picture or other data to the entire group. While simple, it is also cumbersome, and in some settings tend to inhibit the workflow rather than facilitate it.

A more complicated approach uses a remote desktop solution to share what one user sees on her desktop, with the rest of the group. This allows demonstrations to be performed, and may even enable the other members of the group to provide input to the desktop in question, and thus create a more "true" collaborative environment. Other approaches to collaboration range from instant messaging, to multi-peer A/V conferencing and "shared whiteboard" solutions.

This paper presents a middle-path between having an entire shared desktop and a shared whiteboard solution. A shared whiteboard can be great, but has limits in that it is a highly specialized solution with limited abilities. Sharing the entire display isn't always ideal either, for many reasons. Giving others access to ones own desktop is obviously not always desirable[1], and in addition the bandwidth requirements to share an entire display with a room full of computers

*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 may 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.

can put a severe stress on the local network. By using a large, shared surface, such as a display wall, the bandwidth requirements can be reduced, as it no longer is necessary for everyone to have their own view of the shared display. Instead of sharing the display with many computers, the user now only shares it with the display wall. Since the display wall by definition should be visible to everyone in the group, they are still able to see the information to be discussed. The granularity, however, is still limited to sharing either the entire desktop, or nothing.

This is where sharing individual windows comes into play. There are essentially two scenarios:

- 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, or
- sharing a window between multiple users, giving all the users the ability to interact with the window and modify its state, essentially putting up a view of the window on multiple different computers.

These two scenarios will be treated in this paper. Their requirements and limitations 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.

Section 2 describes the requirements of the system. Section 3 details the architecture, before section 4 explains the protocol used to share resources and windows. Section 5 describes the implementation of the prototype window sharing system, first detailing the base layer, before looking at the MacOS X implementation. In section 6, the prototype implementation is benchmarked and some results given, before section 7 goes on to look at the limitations in the current prototype. Some future work is also given, before section 8 examines related work. Section 9 concludes the paper.

2 Requirements

This section deals with the requirements of the window sharing system. The primary goals are to make the system non-invasive[2], easy to set up and use, make it scale well and provide the user with good performance. The system must also work across different window systems and computer platforms, and it should run without needing special privileges.

Sharing windows also presents a number of questions:

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?
3. What kind of access control should the shared windows support? Can many people provide input at once (if at all), and if so, how is this input stream best multiplexed between different users?

[2]Being non-invasive means that neither an approach that attempts to modify the window server, nor an approach that uses extensive patching to reach the goal of sharing windows, can be used.

The answers to these questions will vary depending on the setting where the window sharing system is to be used. For instance, pushing windows will be a natural way of dealing with the first sharing scenario (putting a window up on a shared surface), where the display wall offers space and users "push" their windows to the 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.

2.1 Portability

Making a shared window portable requires examining the mechanism by which a window can be shared. The two approaches to sharing a window 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, depending on what other windows are obscuring the area occupied by the target window.

Using drawing operations is the approach taken by the X Window System, commonly referred to as X11. X11 was developed to allow applications to run on a mainframe, with their user interfaces exported over a network to a dumb terminal. As such, it sends drawing operations over the network, making the dumb terminal perform all the hard work of drawing the graphics, layering windows, clipping, etc. The idea, naturally, is that sending drawing operations requires a lot less bandwidth than sending the pixels that result from the drawing operations. The terminal has to do the work of drawing, while the mainframe can spend its cycles on running the actual program.

Unfortunately, it turns out that in order to be platform independent, the only feasible way to share windows is by exporting the actual pixels resulting from the drawing operations to the receiving clients. The reason for this is simple: 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. Since this isn't possible, the only remaining option is to share pixels. Finally, even if proxy servers *were* available, it would still be simpler to share pixels, as opposed to translating drawing operations between different window servers.

2.2 Access control

Controlling access to shared windows raises a number of issues. Who gets to see the window and who is allowed to interact with it? Can a shared window be re-shared[3], and what happens to the window when the sender no longer wishes to share it with the others? Are all the remote windows closed, or do they remain on their receiver's screen until the receiving user decides to close it? Naturally, it would no longer be possible to interact with the "stale" window, but does that automatically imply that the window must be closed?

Ideally, the user sharing the window will have full control over all these aspects. Unfortunately, there is one aspect that the sharing user can not control: It will never be possible to be

[3]That is, a receiving user decides to share the window she is receiving, thus possibly evading the access limitations imposed on the original window.

completely certain that all traces of a shared window are gone from the receiving users' computers after the window has been un-shared. The reason is simple - a receiving user can merely take a screenshot of the window at regular intervals, and by doing so keep a copy of the window long after it has been un-shared. The same problem applies to a rogue user re-sharing a shared window. While it won't be possible for the re-shared window to gain higher privileges (such as providing input to the original window) than the rogue user already has, the possibility is there that important or confidential information is shared beyond its originally intended recipients. No attempt is made to solve this problem - it is assumed that the group having access to the window is trusted.

Floor management, or floor control, is the second topic related to access control. Floor control deals with who gets to interact with the shared resource at any given time, and is important in cases where many people are trying to work together with a window that doesn't explicitly support multiple users. The most common floor management techniques are:

1. Token-based
2. Director
3. Slot
4. Anarchy

The token-based approach uses a token that can be passed around between the participating users. The user holding the token can pass it on when she feels like it, or it will be passed on automatically if the holding user is idle for a period of time. When a director-based approach is used, one person is designated as the director of the session. This person can grant other users write-access to the window, while revoking it from whoever currently has access. The slot-based approach simply lets each user interact for some amount of time, before letting the next user interact - a simplified, and not very convenient, version of the token-based approach. Finally, anarchy is just that; every user can provide input, with no explicit coordination.

3 Architecture

As part of this project, an architecture and prototype implementation for a window sharing system was developed. This section describes the architecture of the system, and the reasoning behind it.

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.

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, 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

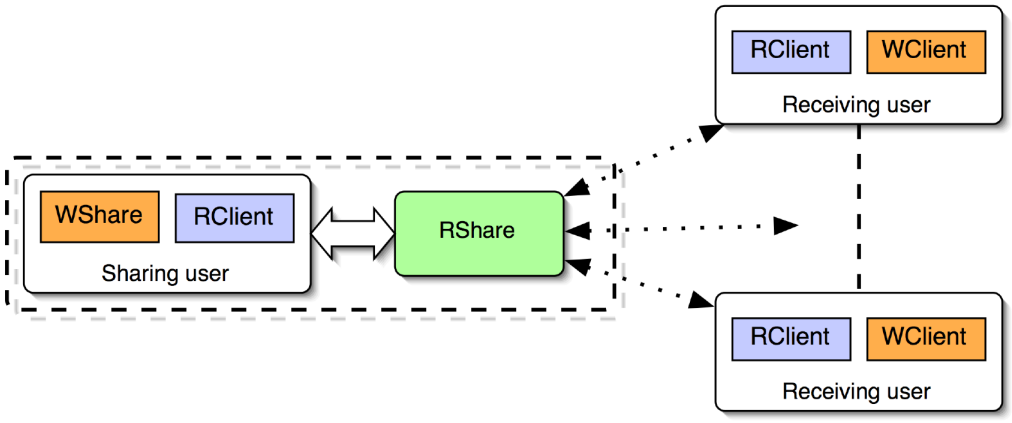

Figure 1: Overall architecture of the window sharing system

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 `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 Figures 2 and 3.

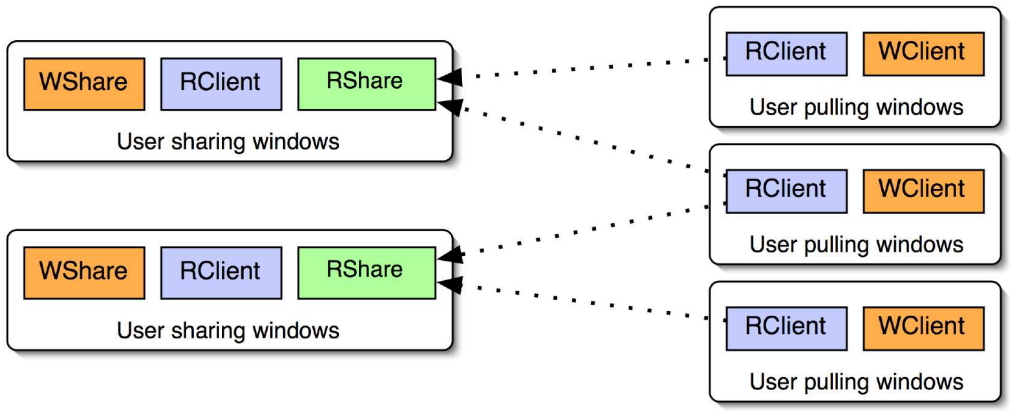

Figure 2: Pulling windows

`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

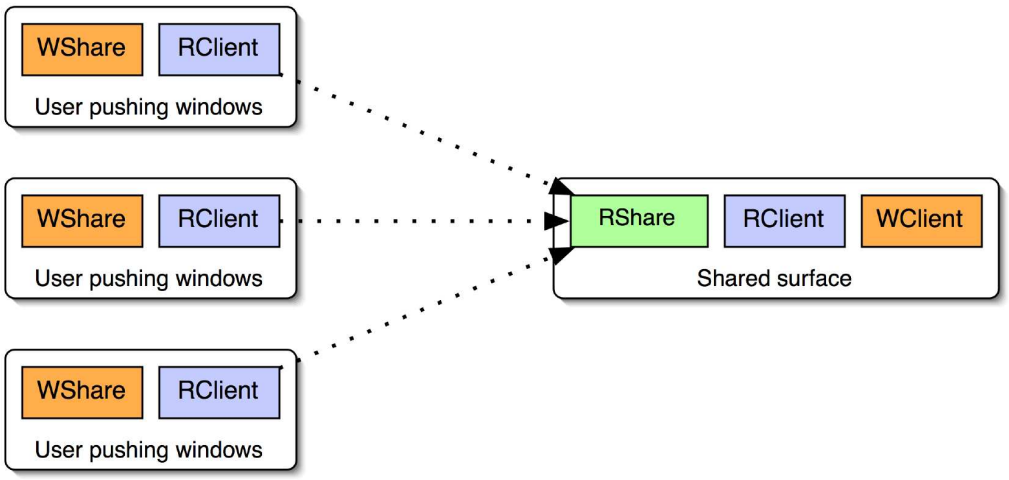

Figure 3: Pushing windows

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 (no interface is necessary for a "push-receiver," i.e. a client that responds to a newly shared window by immediately subscribing to it, and then displaying it on a shared surface).

4 Resource sharing protocol

This section details the workings of the resource sharing protocol, explaining first the operation of the base layer, before moving on to the window sharing protocol.

Once a resource server receives a connection, the resource server expects to discover what kind of connection it is dealing with. The remote peer will initiate communication by sending the connection type, along with the protocol version it is using. The version is needed to avoid problems when clients with differing versions attempt to communicate. There are three connection types:

1. Server
2. Resource
3. User

The Server connection type specifies that the connecting peer is another resource server. In these cases, the two resource servers will exchange information about their shared resources, essentially making their resources available to peers connected to the other server. This type of connection is useful in cases where there is a firewall protecting a number of computers sharing resources. With one resource server running on the outside of the firewall, the resources made available on the inside of the firewall will still be accessible from the outside (see Figure 4), assuming that the server running on the inside connects to the server running on the outside.

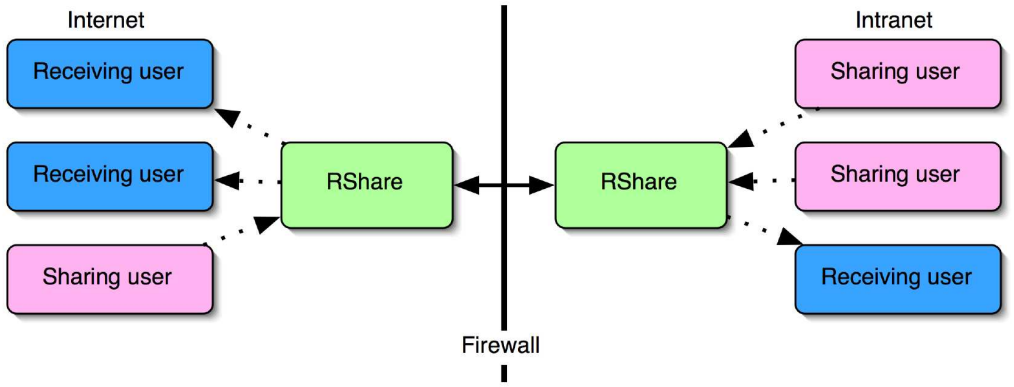

Figure 4: Example of two resource servers connecting to each other.

Server peers behave like both Resource and User peers at the same time. Once a server A has connected to another server B, it will send messages sharing the resources it has available to server B. Server B then does the same. These newly shared "server" resources are called shadow resources. When server A receives a request to subscribe to a shadow resource, it will start by subscribing to the resource from server B, then forward the data associated with the resource to the peer subscribing to it. An important point is that shadow resources are not shared beyond their first shadow server. That is, a resource shadowed by server A will not be shared with a second server C connecting to server A. Server B, however may share that resource with server C. The reason for this limitation is to prevent users from setting up a loop of connected resource servers, which will clearly lead to an infinite number of shadow resources within a very short time. This also prevents resources from inadvertently being shared with servers "far" from the original server, thus avoiding the situation where a user suddenly shares her window with many more people than she originally intended.

The Resource connection type specifies that the connecting peer will be sharing resources, whereas the User connection type indicates that the peer will be subscribing to a shared resource. The main distinction between resource and user peers is that a resource peer may share multiple resources using one connection to the resource server, while a user peer needs to open a new connection per resource it wishes to subscribe to (identifying as a user peer for each connection). The reason for this limitation is to ensure that messages related to a resource will be delivered promptly to every user peer subscribing to it, avoiding the situation where one large message related to a different resource clogs the connection to the user. The situation is illustrated in Figure 5. A second difference is that user peers may not share resources, and vice versa.

While waiting for the connection type, the server will continue accepting new connections and delivering messages. User peers may now do any of the following:

1. Request the list of shared resources, optionally specifying a resource type
2. Bind to a resource
3. Send a resource query
4. Send data to a resource

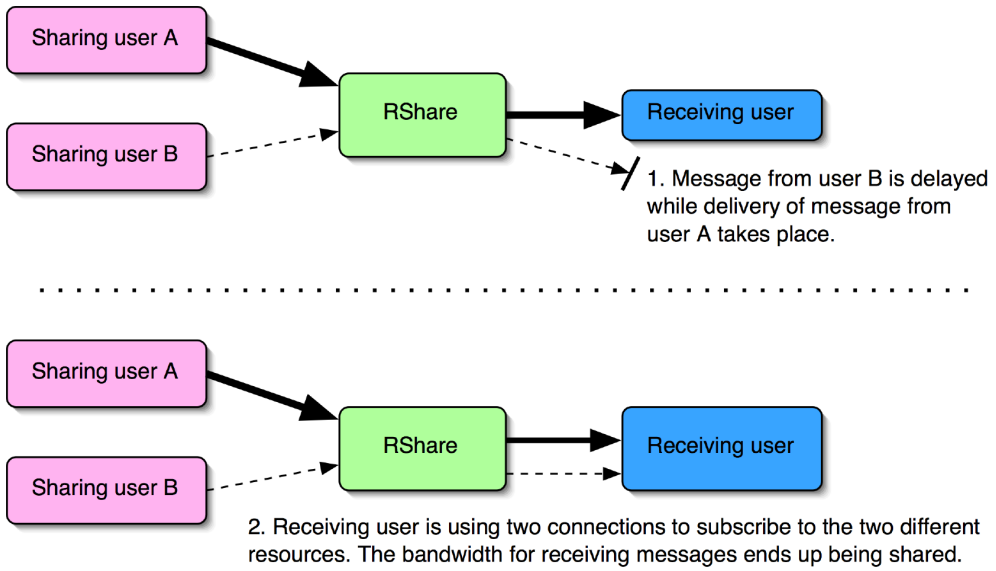

Figure 5: Why user peers need one connection per resource subscription.

A typical user peer will request the list of shared resources, and then send queries to get more information about each resource, before it will end up subscribing (binding) to one of the resources. At this point it may send further queries regarding that particular resource, or provide data to the resource. The difference between sending queries and sending data is that for queries, either the resource server or the peer owning the resource may send a reply to the query. Also, resource queries are always forwarded, even when dealing with read-only resources. Examples of standard queries are queries asking for the name of a resource, a preview of it or more information about the owner of a resource. A user peer may also unbind from the resource at any point, if it so desires. This will enable it to reuse the connection for subscribing to a different resource.

Resource peers can do the following:

1. Share one or more resources with specific flags
2. Invalidate one or more resources it has shared
3. Broadcast data to one of the shared resources
4. Respond to a resource query
5. Change the flags associated with a resource

A resource peer will usually share resources as the user controlling the peer makes different resources (such as windows) available. The flags specified when sharing the resource give hints to subscribing peers about what they are allowed to do with the resource, and also allows for some server side configuration; the currently supported flags are read-only, protected, no-execute and no-shadow. A read-only resource implies that data from user peers will be stopped by the server, and not forwarded to the sharing peer (this does not include query messages). A protected resource requires authentication before the user peer is allowed to successfully subscribe

to the resource. The interpretation of the no-execute flag depends on the type of resource, and is enforced by the sharing peer (i.e., the server only passes the flag along to subscribing peers). For windows, the no-execute flag means that subscribed users are not allowed to post events. The no-shadow flag indicates that the resource server should not share this resource with other connecting resource servers (i.e. server connections).

The messages the server can send fall into the following categories:

1. Replies (such as “resource shared” when a request to share a resource is received, or “resource bound” when a user peer asks to subscribe to a resource)
2. Notifications (a resource has been shared, invalidated or its flags have changed)
3. Errors (resource not found, invalid message, etc.)
4. Routed messages (a query message, for instance, or a broadcast message)

When resources are shared, the resource server will assign the resource a unique ID. This ID is only unique within each resource server. This also affects shadow resources: A shadow resource does not retain its ID on its shadow server(s). The implication of this is that subscribing and sharing peers should not embed specific resource IDs in their messages, or if they do, share the resource with the no-shadow flag set.

When a user peer binds to a resource, the server will inform the peer owning the resource that someone has bound to the resource. Likewise, when a peer unbinds itself from a resource, the owner will be notified. This is done in order to optimize for the fairly common case that a resource doesn’t have any subscribers. In this case, it would be a waste of bandwidth for the sharing user to send messages pertaining to that particular resource (for instance window updates). It also provides the sharing user with the ability to track the number of subscribers.

In the same manner, all connected user clients will be notified whenever a new resource is shared, a resource is invalidated or the flags pertaining to a resource change. This, among other things, makes it possible to create an application that merely subscribes to new resources as they become available, enabling support for pushing windows.

Authentication is accomplished using a very simple challenge-response based protocol. If a sharing peer marks one of its resources with the “protected” flag, the server will send a challenge request to the sharing peer specifying the resource in question. The sharing peer responds with the challenge, which is forwarded to the user peer attempting to bind to the protected resource. When the user peer responds to the challenge, the reply is routed back to the owner, who either accepts or declines the response. If the response is accepted, the server continues with binding the resource to the user peer. Note that the server does not concern itself with what the challenge messages contain, only whether the challenge is ultimately accepted or declined by the owner.

4.1 Window sharing protocol

After deliberating over the best way of sharing a window across platforms, the approach taken in the prototype implementation 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. One exception to this would be displaying a movie, where exporting drawing operations in essence would be equivalent to sharing pixels: “Copy these pixels to this rectangle.”

Sharing pixels also has a more subtle disadvantage: By sharing pixels, what is shared will be the pixels that are currently drawn on the sharing computer. This implies that when the window looses focus, window decorations will be updated to make the window appear "inactive." This in turn causes the window to appear inactive on all the "slave" displays (i.e., displays where the shared window is visible), whereas it will accept input as if it were active. This user interface inconsistency may cause confusion among users, but the disadvantage is not big enough to justify an attempt at writing a protocol-based sharing solution.

The window sharing protocol is similar to the one employed by VNC, 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 using the previously described resource sharing protocol. Once one or more users bind to the shared window, updates will begin being broadcast at some semi-fixed rate (usually at 1 frame per second or more). Each update packet contains encoding information, area covered by the update, and the (usually compressed) pixels. The sharing peer may send any of the following messages:

1. Set size
2. Refresh area
3. Move area
4. Frame marker

The set size message instructs the subscribing peers to resize their representation of the window, to fit the (possibly new) size of the window. The message is sent in response to the window being resized, or as a response to a subscribing peer's request. In most cases, this message will be followed by a refresh message containing pixel data for the newly uncovered area, assuming that the window grew in size. The move area message informs the subscribing peers that an area of the window has moved an integral number of pixels, and that the pixels representing that area can be moved to bring the remote representation of the window up-to-date. This has the potential to save much bandwidth in cases where the move can actually be detected[4]. Finally, the frame marker message allows the sharing peer to define the start and end of an update frame, as a frame may consist of multiple refresh area messages. The message is used to gauge performance on the remote end(s).

Subscribing peers can send the following messages:

1. Get window info
2. Get complete refresh
3. Post event

A subscribing peer will start its interaction by sending the get window info message. This instructs the sharing peer to return the size of the window, at which point the subscribing peer will request a complete refresh of the window. It should be noted that between the time that the peer starts subscribing to the window, and the time it has gotten a complete refresh, it may

[4]No attempt has been made to implement this kind of detection, but support for it has been added in case the system is ported to a platform that supports it natively.

have received many partial updates. Naturally, refresh messages that are received prior to the subscribing client knowing the size of the shared window will be ignored.

The most interesting message is the post event message, which allows a subscribing user to provide the shared window with input. Input comes in two forms: Keyboard input and mouse input. As the remote user clicks and enters text in the shared window, the client will send corresponding keyboard and mouse events as "resource data" messages to the window resource. If the shared window supports and allows input, the `WShare` component will post the events to the window in question. This will trigger updates in the window that the `WShare` component will detect, and then post as refresh messages to the window resource.

The window sharing protocol currently does not do any explicit floor management. Many users all providing input at the same time are thus likely to step on each others toes. Events will simply be merged by the sharing peer, and posted to the window in a FIFO manner. While the protocol doesn't support floor management, this does not prevent the implementation from adding it on top of the window sharing protocol.

5 Implementation

This section describes the prototype implementation of the window sharing system, first examining the message format and base layer implementation, before taking a look at the MacOS X implementation of the `WShare` and `WClient` components. The source code is reprinted in Appendix B, and also available with all supporting files on the accompanying CD-ROM.

5.1 The message format

RShare messages have a fairly simple format. All fields are assumed to be in network byte-order. Every message, except for the initial identification message containing connection type and protocol version, will start with 4 bytes describing length (24 bits) and message type (8 bits), as illustrated in Figure 6. Following this will, depending on the message type, either a resource ID or a request ID be located. Messages that use a request ID are generally messages that expect some sort of reply from someone *other* than the resource server, and always include a trailing resource ID. A typical example of such a message is the query message, which will be sent to the peer owning the referenced resource, when received by the server. When the owning peer responds, a routing mechanism is needed to send the reply back to the peer originally making the request. This is where the request ID is used.

The routing mechanism works by having the server record the incoming request ID, and then exchanging the request ID with a server-unique request ID. The modified message is then forwarded to the peer owning the resource. When the peer responds, the request ID it responds with will be the previously assigned, server-unique request ID. This ID is looked up in the server's internal routing table, and if found, the message is routed to the original peer, with the original request ID replaced for the server-unique one. Routes will expire after a server-defined time (usually about a minute).

For a complete list of the various messages, and their formats, please refer to the `rshare.h` file on page **??**.

5.2 The base layer

The base layer is written in C, utilizing standard BSD sockets and pthreads for network and threading support. The `RShare` component works by calling an init routine that spawns off the server thread. The server thread sets up a socket listening on a caller-defined port (usually

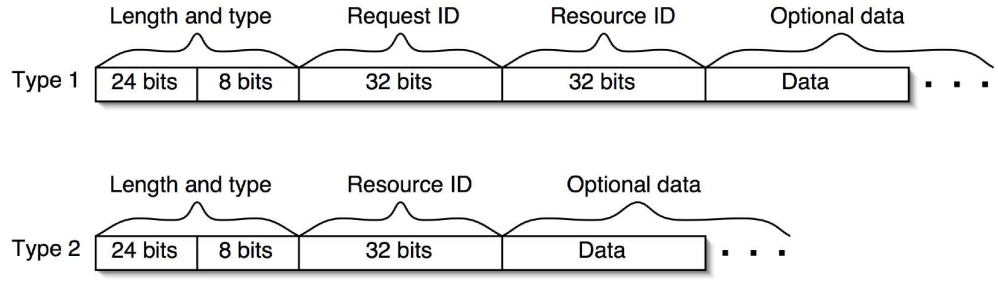

Figure 6: The two most common RShare message types.

in the 7000-7010 range), and if enabled, also opens a multicast socket used for RShare server discovery. Once setup is complete, the server starts accepting connections. The main loop runs through the available connections, reading data, processing and sending messages. The overview can be seen in Figure 7.

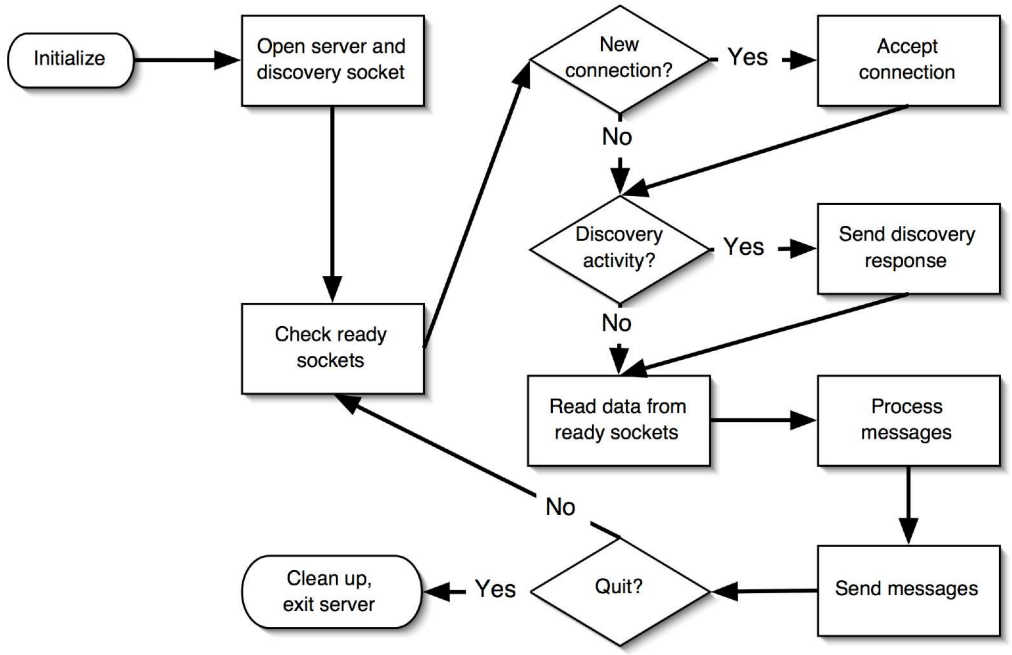

Figure 7: The server loop.

Misbehaving clients are handled by disconnecting them. Any resources associated with the peer (either shared or subscribed) will also be removed. This implies that any resources shared by a misbehaving peer will be invalidated when the peer's connection is terminated. Misbehaving user peers have no effect other than lower the subscription count for the resource they were subscribing to when they are terminated.

When a resource peer shares a resource, the server creates a resource structure and attaches it

to its internal representation of the peer. The resource is assigned an ID, and then entered into a hash table. This makes it easy to look up resources, and also simple to find the resource's owner, since the owner is also stored as part of the resource structure. Similarly, for user peers, once they subscribe to a resource, the resource's subscriber count is increased, and a reference to the resource is attached to the user peer. The resource structure also contains a list of subscribed peers, making broadcasting information to subscribed peers a simple matter to implement. The peer and resource structures can be seen in the `rshare.h` file, starting on page **??**.

5.2.1 RShare server discovery

Server discovery is an optional feature provided by the `RShare` component. The purpose of server discovery is to make setup simple for the users, alleviating the need for the user to input the name of the RShare server to which they want to connect. This allows the user to browse the available RShare servers without any specific directory being necessary. The feature is especially useful in the scenario where windows will be pushed to a shared surface. In this case, there will most likely be only one RShare server in use, and thus the details of connecting to the server can be completely automated.

Server discovery works by opening a multicast socket on port 7000. All messages received on this socket will come as one of two types: Server solicitations and server advertisements. The message format is simple - a magic number, followed by the message type, and then either two nil bytes or the port on which an RShare server is running, as illustrated in Figure 8. The message ends with a second magic number. The internet address, in case of a server advertisement, will be taken as the address from where the multicast packet originated. The magic numbers are used to identify the packet as a valid RShare advertisement or solicitation. All other messages will be ignored.

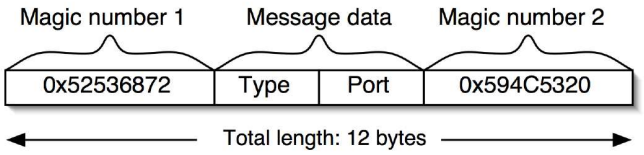

Figure 8: The format of an RShare solicitation or advertisement packet.

The server will periodically send advertisements to the multicast address associated with the discovery socket, with exponentially increasing re-send delays: 1 second, 2 seconds, 4 seconds, etc, up to a maximum delay of about one hour. An advertisement will also be sent in response to a discovery request.

5.2.2 The RClient interface

The `RClient` component operates in much the same way as the `RShare` code. An initialization routine is called, which spawns off a thread that connects the client to a server specified in the initialization call. If the server connection is successful, the client code will run a loop sending and receiving messages. The code using the component queues messages for transfer using a number of messaging functions. When messages are received, a user-supplied callback is called to inform of the event, and hand over processing to the user.

The `RClient` code also provides routines for discovering servers.

5.3 WShare and WClient on MacOS X

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. Both are written using Cocoa and Objective-C[5], integrating neatly with the `RShare` and `RClient` C implementations. Screenshots of the applications can be seen in Figures 9 and 10.

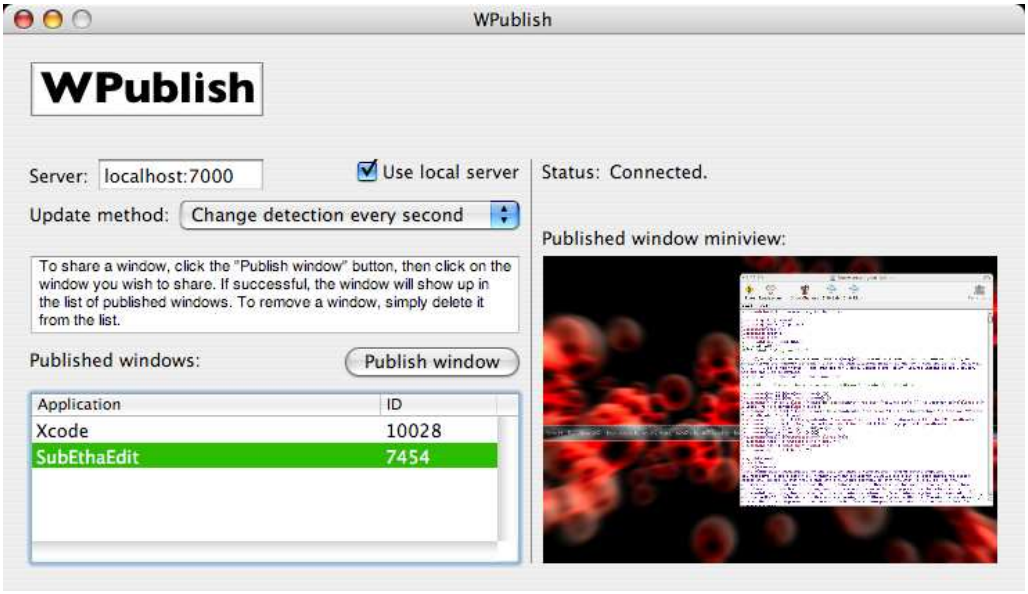

Figure 9: Sharing windows on MacOS X.

In order to understand the MacOS X implementation of `WShare` as embodied by the WPublish application, 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 one giant root window contains a number of smaller windows and the application's menubar, 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 (a common exception to this rule is games, who often handle their own backbuffers). 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,

[5]More information about Objective-C and Cocoa is available in Appendix A, in case the reader is unfamiliar with development on MacOS X.

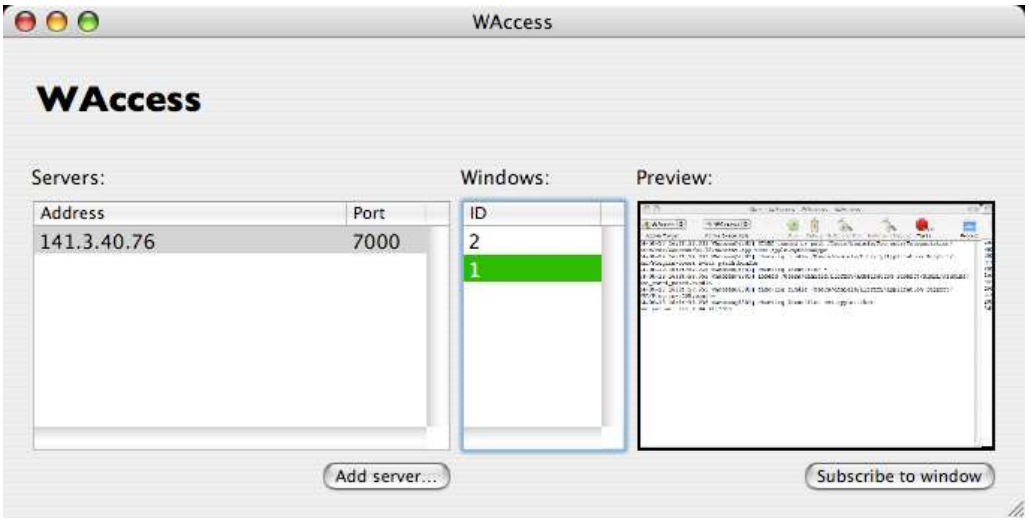

Figure 10: Accessing shared windows on MacOS X.

and also windows layered below in case the window being rendered is also translucent. 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 (in essence, an application linking against Xlib and friends). 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 allow posting of events to specific applications[6]. 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. The goal, however, is not to write yet another remote desktop client[7].

Luckily, the window server provides some SPIs[8] that allow applications to list all on- and offscreen windows, as well as gain access to the pixel contents making out these windows. What it doesn't provide is a way to post events to specific windows or applications. The prototype implementation solves this problem by inserting code into all Cocoa applications that allows WPublish to post events to windows specific to the applications in question. This is done in an OS sanctioned way, however it still qualifies as a patch and thus may not be suitable in all cases.

[6]This is not entirely true. By reverse engineering the window server, it should be possible to discover the format of the low-level Mach messages the window server uses to post events to applications, providing a possible event injection vector. This has not been done, however.

[7]As a spinoff from this project, the author of this paper has released a shareware remote desktop client for MacOS X called Desktop Transporter.

[8]Secret programming interfaces. These are interfaces that aren't officially supported by Apple, and thus are subject to change with OS revisions, or even disappear entirely. Binary compatibility between OS releases is thus unlikely achieved. They are completely undocumented, and their usage is usually discovered through reverse engineering. For this project the interface to the functions that provide pixel access to windows and allow identification of which window is at a particular point on screen were reverse engineered.

Also, it doesn't solve the problem of posting events to Carbon or X11 applications.

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 - that is, 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 at least once every second. 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 (say once every second), at the cost of 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. Since almost all of the movie window's contents will change numerous times every second, it makes good sense to skip the comparison step, and simply transfer everything.

The WAccess implementation is fairly straightforward, giving the user the ability to subscribe to windows from different resource servers. It uses the RShare server 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, which is accomplished by sending a preview query on the resource in question.

Both applications are implemented by having a controller class handling user input and messages from the `RClient` layer. When the user interacts with the applications, events are forwarded in the usual "first-responder" fashion that is common in Cocoa-based applications.

5.3.1 A platform independent WClient

In order to test the window sharing on more than one platform, a platform independent `WClient` was written that utilizes SDL[9] to do its input and display handling. It is a bare-bones, not-very-user-friendly 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, and thus allows testing of all parts of the system.

5.4 Pushing windows

Pushing windows was easily 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[10] an instance of the platform independent `WClient`, instructing it to connect to the local server and subscribe to the newly published window resource.

An example where this setup would work well is when the push-receiver is running on a computer running the X Window System, and whose `DISPLAY` points to a display wall. New windows would then pop up on the display wall once someone shared it to the push-receiver.

[9] Simple Direct-Media Layer. An open source, platform independent set of libraries aimed towards game development.

[10] Forking was a last resort here, but necessary due to SDL's inability to handle more than one window for every process.

5.5 Performance

In order to improve the performance of the prototype implementation, the window refresh messages have support for various compression methods (although only one has been implemented). 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. An RLE pair is indicated by setting the most significant bit (MSB) in both 16- and 32-bit varieties. This is not a problem for 32-bit pixels (which in reality only use the lower 24 of the available 32 bits), but can pose problems with 16-bit pixel values that use the MSB. Most platforms use this bit to indicate transparency, something that is mostly irrelevant for the purposes of window sharing, so the use of this bit in a shared window is silently ignored.

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 worst case performance of the RLE algorithm is no compression - the algorithm will thus never produce data that is bigger than its input[11].

In addition, the user can elect to share windows using thousands of colors (16-bit) instead of the (in most cases) native depth of millions (32-bit). This will cut bandwidth usage roughly in half, and also improve update latency due to the shorter length of the packets[12].

A second way to improve performance was to send scan-lines instead of large rectangles when sending the updates. This incurs a slight overhead due to the message headers, but improves the end-user responsiveness a great deal. The reason for this improvement is that the time required to send a short message is lower than the time required to send a long message, allowing the message to be processed earlier at the remote end. An illustration of this can be seen in Figure 11.

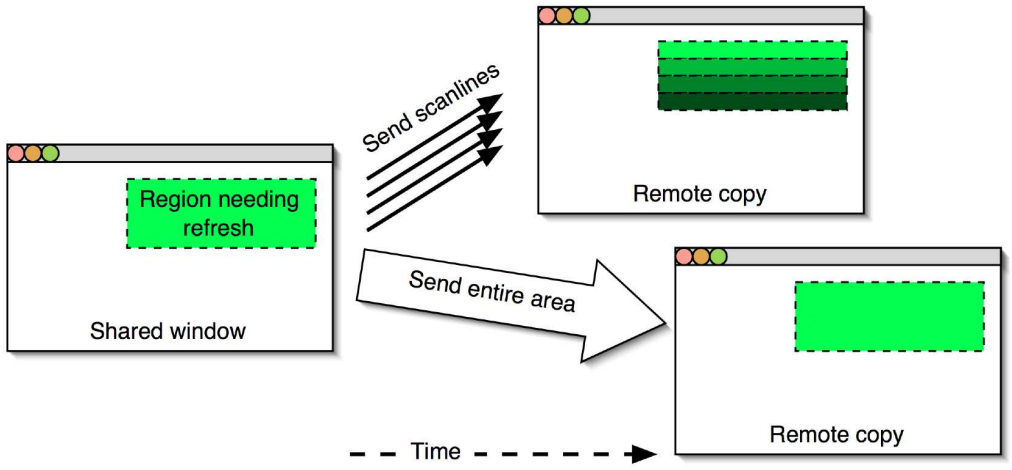

Figure 11: Sending scan-lines instead of entire areas to improve performance.

The best way to imagine this is to visualize sending a complete refresh. When not sending

[11] This is only because the algorithm defines one bit in both 16- and 32-bit color entries as unused, and uses this bit for marking whether the next data element is a pixel or a run-length pair.

[12] Note that the current implementation only allows the user to share using the native depth.

scan-lines, the entire refresh will be sent as one message. Before the remote end can start drawing the contents of the windows, it needs to receive the entire refresh message. Since the message is long, this might take a while, but once the message is received, drawing the update is practically instantaneous. On the other hand, when sending scan-lines, the user will see a window being drawn top-to-bottom, one line at a time. This is a good example of overlapping work with I/O. Finally, sending scan-lines allows the next optimization to work better, though it also has the effect of making it require more CPU time.

The third optimization, called the congestion-avoidance mechanism, attempts to remove redundant update messages from the sharing peer's output queue, before they are sent to the resource server. This is both an optimization and a necessary mechanism to avoid network congestion - with the network congested, packets will not be delivered promptly, and a large backlog of updates will pile up on the sharing side. The effect is that the remote users see the window as it was 10 seconds ago, instead of its current state. The mechanism works by examining the message that was last added to the output queue and extracting the rectangle that describes the area being updated. It then iterates over all the messages remaining in the output queue, comparing the extracted area to the area described by the current queue element. If the extracted area completely overlaps the area being examined, that refresh message is removed from the queue, since a more recent update is available further back. Unfortunately, the congestion-avoidance optimization can have a performance impact on the implementation (see 6.1), and it also conflicts with the local-mode optimization, described next.

The next optimization made to the base layer, called local-mode, 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 (which in reality would be the loopback interface). 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. For security reasons, 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, as mentioned when describing the MacOS X implementation of `WShare`, change detection is used to minimize the number of pixels that need to be sent over the network. This part of the implementation compares the pixels from the previous version of the window to the ones in the current version, and then determines which parts of the window have changed. In order to reduce the message overhead, the pixels lying adjacent to a changed pixel are also considered changed. This makes it possible to detect small, non-contiguous, regions and send them as one unified update. The change detection implementation builds a list of dirty rectangles that are decomposed into scan-lines and then sent as refresh messages.

6 Benchmarks

In order to have meaningful benchmarks, it is important to define a proper metric for gauging the performance of the window sharing system. For this purpose, frames per second (FPS) was decided upon as the best way of gauging this performance. This allows one to easily study how sharing many windows from one computer may degrade performance. Bandwidth usage and latency measures are also used to measure the performance.

The testing setup consisted of a PowerBook 1.25 GHz sharing windows to two PCs with PIII processors at 400 MHz and 800 MHz running SUSE Linux, all connected by the same local, 100 MBit ethernet network. The figures measured were number of FPS the sharing peer was capable

of delivering, and the observed number of FPS at each subscriber. For multiple receivers, the results are averaged. For all tests, the target FPS was set to 15[13].

The first benchmark shared 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 looping music video[14]. The pixel depth was 32 bits, and the tests lasted for approximately two minutes each.

When considering the results from this benchmark, it is important to keep in mind that it is performed with the constant background load produced by playing the movie. This background load is not insignificant - measurements indicated that QuickTime Player regularly consumed more than 50% of the available CPU time during sharing.

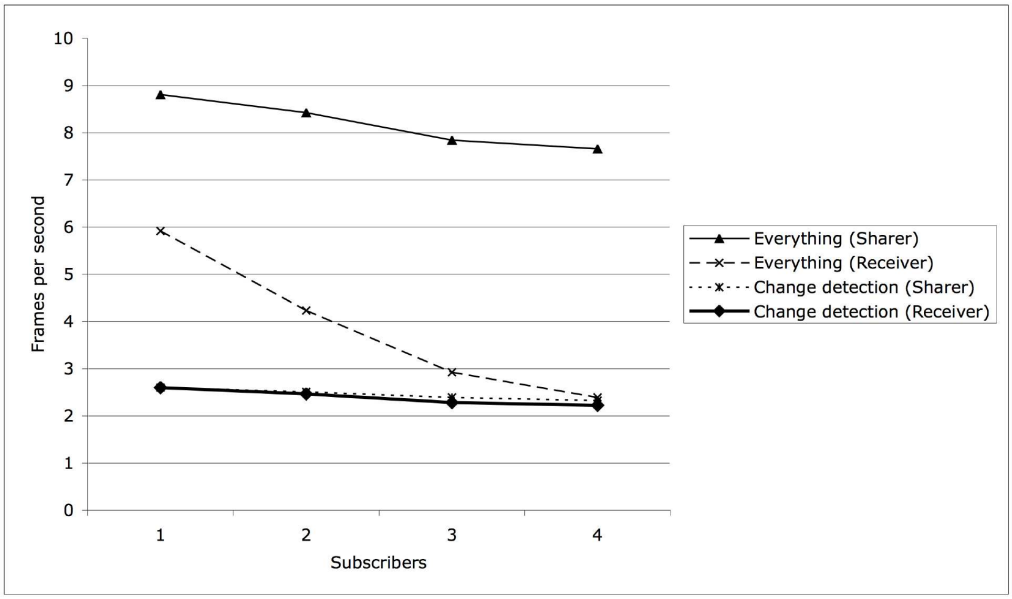

Figure 12: Graph showing how FPS varies with number of subscribers and update method.

The graph in Figure 12 shows how the sharing software performs when using either the change detection based update method, or the plain “send everything as often as possible” approach as the number of subscribers increases. For the change detection approach, the number of FPS the sharing peer is capable of delivering is quite limited, and remains fairly constant (the slight decline is likely due to the increased load of sending updates to more subscribers, combined with small variations in background load on the sharing computer). The subscribers have no problems keeping up in this case.

For the send-everything approach, the results are different. The number of FPS the sender is capable of delivering sees a slight but steady decline as more time is being spent transferring large amounts of data. This problem 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

[13]This setting causes a timer to fire at most 15 times per second in the WPublish application. If WPublish spends too much time performing things like change detection, some of these timer firings may be missed.

[14]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.

composed and broadcast. Interestingly, the subscriber-observed number of FPS is significantly lower than what the sender is capable of delivering. This has several reasons, but the most important is that the sender will remove frames from its output queue in cases where it suspects congestion may become a problem (the threshold is adjustable, but for the experiments it was set at a queue length of 1 MB). This means that even though a frame counts as delivered by the sender, it may never actually reach the network. Second, congestion *does* become a problem, which is why the number of FPS seen by the receiver eventually drops to come on par with what can be observed when using change detection. A screenshot from a frame of the QuickTime video can be seen in Figure 13. Note how the window has some quite significant borders that will be transferred only once by the change detection method.

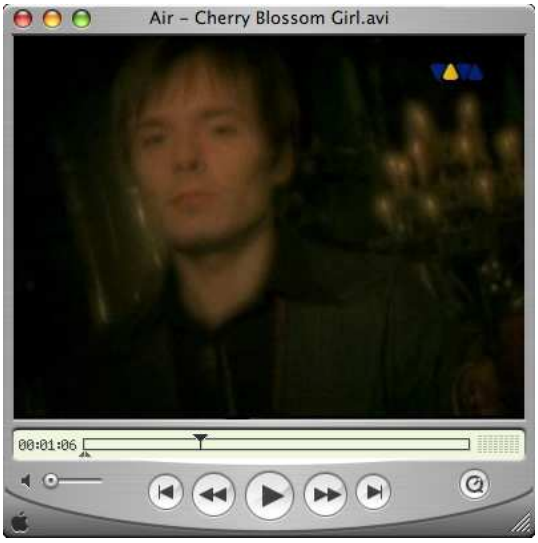

Figure 13: QuickTime window being shared.

This benchmark shows that using change detection clearly isn't a good idea in cases where CPU usage on the sharing computer is important, or when one knows that most of the contents of the shared window will change often.

The second benchmark examines how the performance scales when the PowerBook shares more than one window, ranging from one to four. Each window has one client, running on either of the two PCs. Two of the windows contain fairly static contents, whereas the other two windows are QuickTime movies. The two static windows are an Excel spreadsheet and a text editing document, sized at 740x603 and 713x646 pixels respectively. The first movie remains 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 the second, smaller movie window. During all the tests, the various static windows were interacted with, in order to provoke updates. Text was entered, cut, pasted, and the window contents were scrolled around.

The graph in Figure 14 shows how performance deteriorates as more windows are being shared by the sender. The first question that arises from looking at the graph is why change detection reaches such a high number of FPS when sharing a window that is bigger than the movie shared in the first benchmark. The answer has two parts: First, the movie window used in

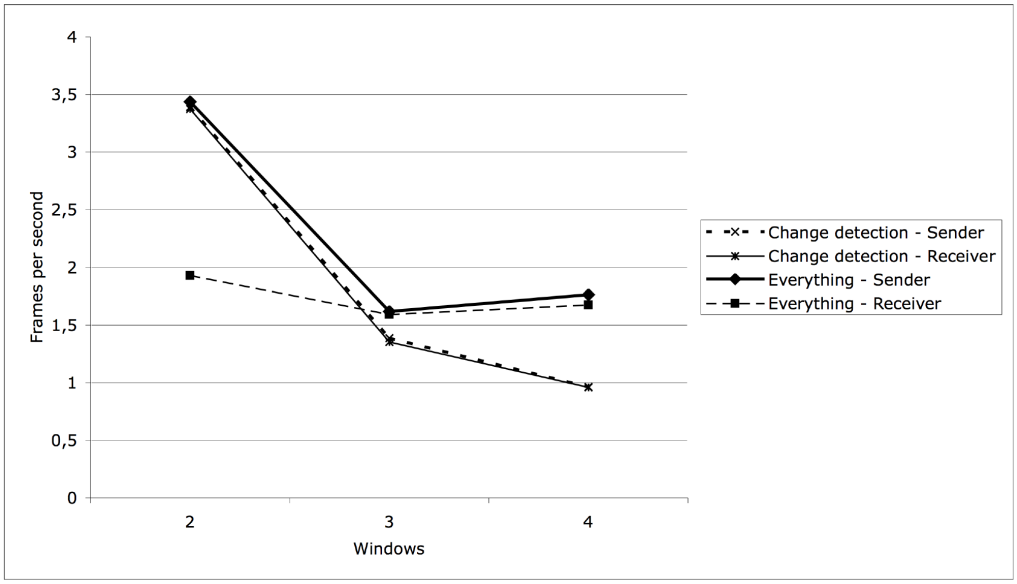

Figure 14: Graph showing how FPS varies with number of shared windows and update method.

the first benchmark was never static (except for maybe a handful pixels in addition to the static window decorations and widgets), so updates were constantly being sent. To contrast this, the text edit window rarely changed much, except when actions were performed in the window. The second part of the answer lies in background load. With a movie playing in the background, less CPU power is available to perform the change detection. To illustrate the impact of this, the movie window was shared a second time, using change detection, but this time *without* the movie playing. With one subscriber, both sharer and subscriber reached an FPS of about 13.7. During the test, the movie was being jumped through by dragging QuickTime's time-line slider to different points, producing changes in the window.

This explains how the FPS seen in the graph can be better than those observed in the first benchmark, and also illustrates how the contents of the window and the background load can affect performance. As the graph also shows, sharing a large window by constantly sending everything can be a recipe for bad performance. Delay between action in the local window and appearance in the remote window was also a problem in this case (see below).

As the number of shared windows increases, the performance of the system keeps dropping. Change detection doesn't do much for the movie windows, but at least manages to keep the bandwidth overhead from sharing the two mostly-static windows low. Unfortunately, at this point the change detection requires so much time to complete that it devastates the number of FPS the sharer is capable of delivering, while also being in competition with two playing QuickTime movies for processor time. A good way to improve on this would be to allow different update modes for different windows - technically not a problem, but the interface for WPublish does not give the user the ability to make this distinction yet.

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[15].

The next benchmark examines how the local-mode and sending scan-lines changes the performance of the window sharing system, again using the first playing movie as the shared window. The two tests were:

1. Local-mode disabled
2. Scan-line mode disabled

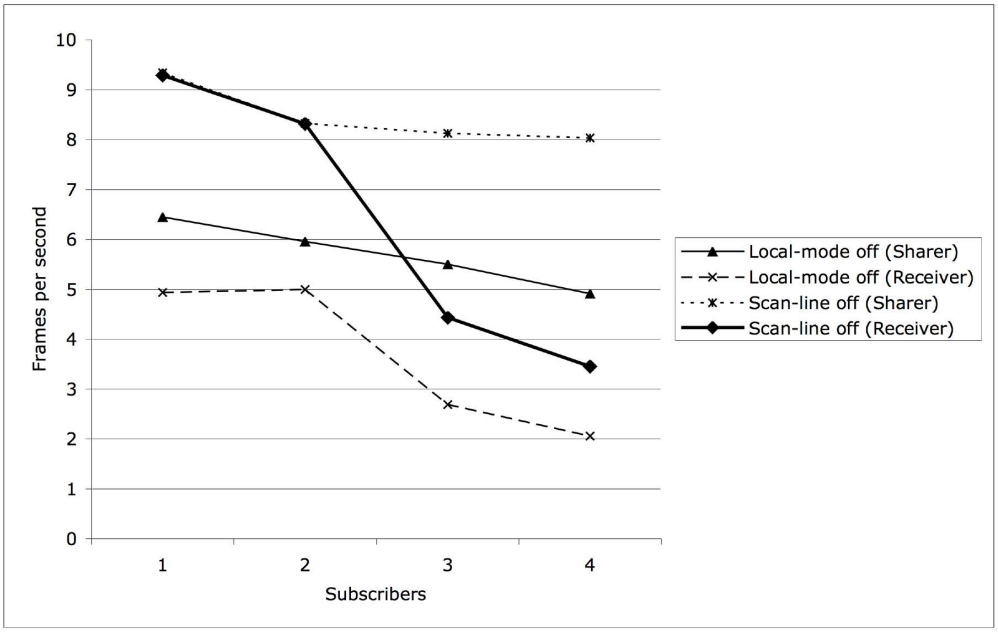

Figure 15: Graph showing how FPS varies the various optimizations disabled.

When local-mode is disabled, both sharer- and receiver-observed FPS drops, and quite considerably too (see Figure 15). What is not visible from these results, however, is how big the impact is on the "feel" of the shared windows. The problem with local-mode is that it gets rid of packets too quickly - so fast, that the congestion-avoidance optimization only works at irregular intervals when the network becomes heavily congested. The end result is that the subscribers see frequent freezes and time skips, whereas without local-mode, the frame rate drops slightly, but the perceived frame rate is much more constant. For one or two subscribers, local-mode is a win, but once more subscribers enter the loop, local-mode is no longer a good optimization for the task of window sharing.

When scan-lines were disabled and entire areas were sent instead, two things happened. First, the sharer- and receiver-observed FPS soared when one or two subscribers were connected. Performance was *better* than what could be observed when scan-lines were enabled, and surprisingly, out-performed all the other configurations. At three clients, however, things started looking

[15]The total area shared is 2205x2003 pixels, equating about 17 MB of data requiring transfer as often as possible, compared to 400x401 (times 4 subscribers) = 1600x1604 pixels, or about 10 MB. These amounts of data can clearly saturate a 100 MBit ethernet network. Note though that these figures do not factor in the gains from using RLE compression, which is always enabled.

down, although this is hard to observe just from looking at the graph. The number of FPS took a drop, but the performance compared with what was observed in the previous benchmarks was horrible - in some cases, the picture on screen lagged behind the actual window with more than 30 seconds, catching up from time to time before freezing. The situation didn't get better when the number of clients were increased to four, leading to even greater delays.

The conclusion that can be drawn from this is that while scan-lines may decrease performance in some cases, they aid in promoting fairness. Second, they are better suited for the congestion-avoidance optimization.

In order to investigate whether the congestion-avoidance optimization would work better with local-mode disabled, the scan-line test was performed once more without local-mode. These results were considerably better - with 4 subscribers easily supported at a steady 7.41 FPS. The average FPS observed by the receivers was 5.72, and most importantly, the update frequency at the subscriber end was steady, without the freezes observed in the previous experiment. While this is uplifting, the window still lagged 12 seconds behind the window on the sharing computer, once again reinforcing the merit of the scan-line approach.

These results make it interesting to examine exactly by how much the remote windows lag behind the real window, located on the sharing user's computer, when the remote user is interacting with the shared window. In order to get an approximate estimate of the latency from action on the remote computer, until the result is visible to the remote user(s), 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 setup can be seen in Figure 16.

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 (even though it makes good statistical sense to ignore or give less weight to samples that lie far away from the current measurement averages), 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.

The final benchmark simply aimed at measuring the bandwidth usage for sharing a window, when either using change detection, or sending everything. The scenario is almost the same as in the first benchmark, with number of subscribers ranging from one to four, but only sharing the window for one minute.

As is evident from Figure 17, the bandwidth usage is increasing linearly with the number of

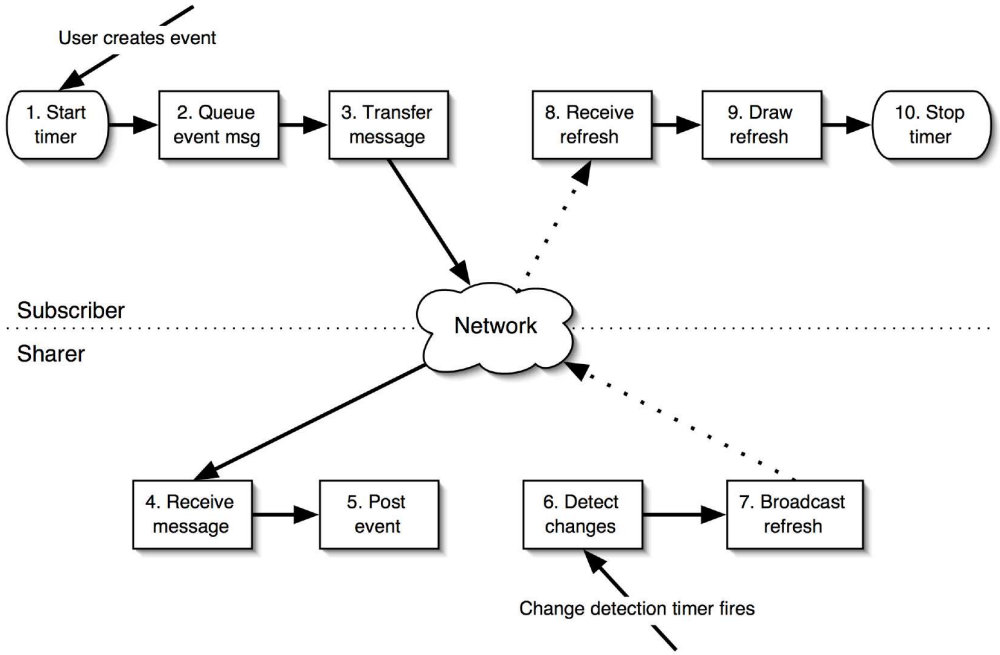

Figure 16: Setup for measuring remotely observed latency on posting events.

subscribers. The number of frames delivered doesn't scale as nicely, but this is likely due to the changing behaviour of the congestion-avoidance mechanism (it may decide to remove an entire frame from the output queue, which will lower the FPS count, but prevent congestion so that further updates will go through without dropping scan-lines).

A second figure of interest related to bandwidth, is the approximate bandwidth savings provided by the change detection update method. One frame using the send-everything approach averaged 0.52 MB[16], while a frame using change detection averaged 0.39 MB. These savings must be seen in light of the radically lower FPS provided by the change detection, however.

It is difficult to determine why the available bandwidth isn't further utilized when the send-everything test reaches 4 subscribers. One possible explanation is background traffic on the network, preventing WPublish from reaching the network's peak bandwidth, though in repeated tests this did not appear to be the case. Another possibility is that WPublish isn't able to deliver the amount of data needed to saturate the network, and finally the most likely explanation is that the congestion-avoidance mechanism is acting prematurely, removing data from the output queue before it is strictly necessary.

6.1 Improving performance further

As these benchmarks were made using a fairly unoptimized version of both the WPublish application and the Linux `WClient`, there is still room for improvement, both algorithmically and in the implementation. To determine areas that could use optimization in WPublish, the

[16]The reason this is lower than the expected frame size of 0.61 MB (400x401 pixels at 4 bytes each) is partially due to RLE compression, but mostly due to the congestion-detection mechanism, which should account for most of the "missing" data.

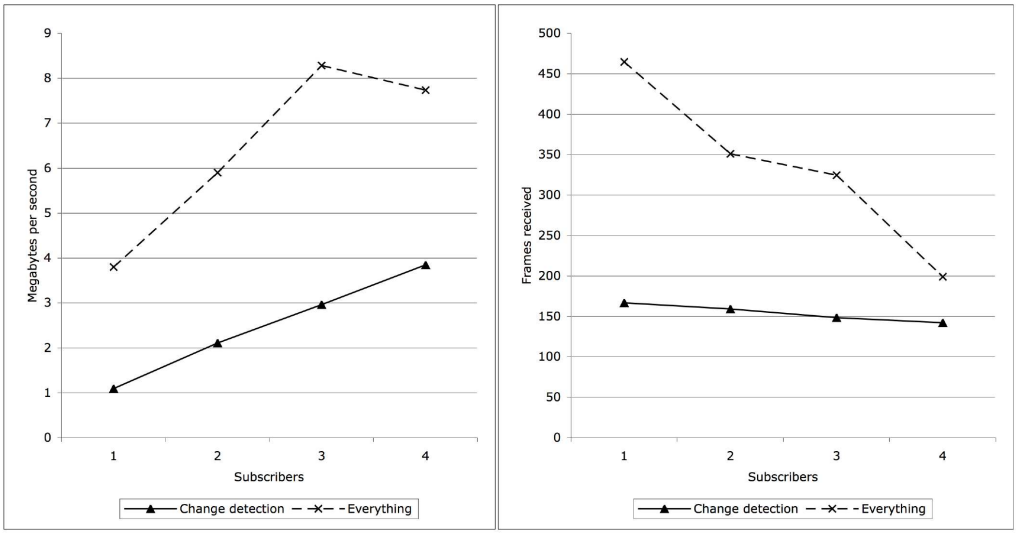

Figure 17: Graphs showing MB/s and number of frames received.

Apple-provided profiler Shark was used to determine potential hot spots in the WPublish implementation.

Shark pointed at a number of areas where performance could be enhanced. Not surprisingly, the congestion-avoidance function consumed much time in cases where the output queue was long. As the congestion-avoidance mechanism works by checking each submitted rectangle against all the other rectangles in the output queue, its algorithmic complexity at first glance appears to be linear. However, since we usually submit scan-lines, and the congestion-avoidance mechanism is invoked once for every rectangle submitted to the queue, it becomes clear that the *use* of the algorithm is anything but linear. This algorithm can be further tuned, and enhanced by using a different data structure, such as a tree, for the window refresh messages (though this would not integrate well with the current queue-based I/O mechanisms implemented).

A second area concerns the rectangle management routines, which work well for small numbers of rectangles, but are unsuitable when the managed rectangles are many and small. The main performance offenders are recursion, list traversals and the overhead from object messaging. Taking a different approach, such as the bitmap scheme used in VNC, could definitely improve performance in cases where change detection is used on a surface that changes a lot very frequently.

Much time is also spent fetching a copy of the shared windows' pixels. Although it's not possible to optimize the functions provided by the operating system, an attempt to optimize their use further can be made. For instance, it is not clear whether the image handle needs to be refreshed every time an attempt to access a window's pixels is made, or if it is possible to use the one already available. Figuring this out would require taking a second look at the SPI that provides the image handle.

For the platform independent `WClient`, the most eligible candidate for optimization is the drawing routine. Currently, the drawing is more functional than optimized for speed, something that is very visible when sending scan-lines instead of entire rectangles. Once a refresh message has been received, it will be decompressed and immediately drawn to screen. A better approach would be to defer the actual screen update, potentially accumulating more scan-lines that can all

be drawn in one go.

On a higher level, it should be investigated if there is a middle-path between sending scanlines, and sending entire areas, and if so, determine where the peak performance between these two approaches lie. In such an endeavour, there are many factors to consider, such as message overhead, delay between message send and the time it is ready to be drawn remotely, how the solution integrates with the congestion-avoidance code, and lastly if it works well with both sending everything, and doing change detection.

The ultimate speed-limiting factor, however, is how fast the publisher is able to detect and send updates. If the publisher is able to saturate the network, it can be assumed that updates are being delivered as fast as possible. At this point, other bottlenecks should be easier to detect, and optimize or work around.

7 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.

Coordinating multiple users' input to a single shared window (floor control) also needs further work, and in trying to look at possible solutions, a minor design flaw in the resource sharing base layer was uncovered. The flaw resides in the fact that it is impossible for the sharing peer to send messages to specific subscribers, unless the message is a reply to a message initiated by one of the subscribers. This in turn implies that peers should have some abstract identifier associated with them, to allow for more involved message exchanges routed by the resource server.

A rather big limitation with the current MacOS X implementation is that it does not allow for posting events to all windows (or any windows, depending on the whether the user will allow the installation of a small patch or not) - 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. The main use of sharing information with others still makes the MacOS X implementation useful, however. 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[17].

Finally, the current implementation of the `RShare` server does not support Server-Server connections (see Figure 4).

8 Related work

There already exist a number of solutions for sharing either an entire desktop or individual applications with other users. VNC 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 - while some implementations allow sharing only a part of the display, it is still only the pixels that are actually visible that will be shared. This contrasts with the window sharing prototype in that the prototype does not require that the window is actually visible in order to share it with other users, thus allowing the sharing user

[17] In Apple's DVD Player, it's even being "abused" to prevent screenshots being made from the movie.

to keep working with her entire display, without having to worry about moving sensitive data into some "shared region." Other solutions include Apple and Microsoft's own OS-dependent implementations, sharing only the desktop with other computers.

QuiX (later renamed to MaX) [1] is an application that works with older versions of MacOS (presumably System 7.5 and earlier) 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[18] drawing operations into X protocol drawing operations. This is clearly different from the approach taken by the prototype, which shares pixels, not drawing operations.

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. In addition, to use Messenger, users are required to "sign-up" with Microsoft in order to get a "passport," a significant disadvantage for the privacy-concerned. 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 peers' 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 [2], as the X11 protocol wasn't written with multiple clients 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) [3]. Two examples here are Xplexer [4] and XTV [5]. These applications both attempt to replicate the windows on X11 across multiple different X servers. Unfortunately, these and other applications replicating windows on X11 tend to be out-of-date and lack support for newer X11 servers or clients. Many of them are also riddled with bugs, making them unusable in practice. Finally, the X servers often need to be configured similarly in order for the replication to succeed acceptably.

Xmove [6] 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 actually share windows, only migrate them. 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.

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.

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

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 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 system should scale well, up to the point where the sharing user is unable to send updates to all the subscribers, or no longer has enough processing power locally to handle change detection on many shared windows. Depending on the content being shared, this limit can be high or low - using change detection on a mostly static window will allow the window to be shared with many users, whereas a constantly changing window being completely updated as often as possible will limit the number of subscribers to at most 5 or 6. Another limiting factor is the size of the window.

The protocol that was developed for sharing resources works well, apart from the weakness in its lack of directly addressable peers. This weakness currently only prevents the development of proper floor control for the window sharing aspect of resource sharing, and does not substantially detract from the overall working of the system. The window sharing protocol also performs well, and offers good room for future improvements, both in terms of how frames are encoded, and how updates are sent (scan-lines, blocks, etc.).

The window sharing system as it stands now is fully usable for sharing windows from computers running MacOS X to other computers, supporting auto-discovery of other users to make finding shared windows as easy as possible. The underlying architecture has good room for further expansion, both for sharing windows and other resources, providing a good foundation for further enhancements and research.

10 Acknowledgements

Thanks to Otto J. Anshus, John Markus Bjørndalen, 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] 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.

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

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

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

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

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

[7] *The Objective-C Programming Language*. Apple Computer, Inc., February 2004. http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf.

A A brief introduction to Cocoa and Objective-C

Cococa is the preferred API to target when writing applications for MacOS X, and allows the use of either Objective-C or Java as the programming language. This appendix is intended to give a brief insight into how a MacOS X Cocoa application is developed, with the aim of making the code for WPublish and WAccess easier to understand.

A.1 Objective-C

This section deals with the Objective-C programming language, and is in no way intended to be comprehensive. For a more detailed introduction, see [7]. The Objective-C language evolved as a fusion between C and concepts from Smalltalk. It makes a small number of syntactical changes to the C language, allowing one to create object-oriented applications. Objective-C revolves around the object and messaging concepts. An object is just a collection of methods and instance variables, possibly inheriting from a class higher up in the class hierarchy. Objects communicate by sending messages to each other - roughly equivalent to function invocation, but implemented as "true" message sends, yielding much greater flexibility.

The syntax used to send a message to an object is:

```
[receiver message];
```

Where the `message` part consists of the method name, and any variables. Objective-C supports named variables, though their use isn't mandatory. An example from the WPublish code:

```
[scr_view connect:servers[srv_idx] res_id:resources[rsrc_idx]];
```

This sends a message to the object `scr_view`, invoking the method `connect:res_id:` with the parameters `servers[srv_idx]` and `resources[rsrc_idx]`. An object can send a message to itself by sending a message to the object `self`, which is similar to the `this` pointer in C++.

Objective-C supports dynamic typing, which among other things allows sending of messages to objects whose type isn't known. The message sending mechanism will inspect the object, and determine if the selector[19] being used is supported, and if so, forward the message to it. This makes it possible for the following line to work, regardless of what kind of object is contained in the `data` pointer (assuming that the object implements the `process_msg:` method):

```
rshare_result   message_avail_callback(void *data) {
    [(id)data process_msg:0];
    return kRS_Success;
}
```

The cast to `id`, which is the generic Objective-C object type, allows one to make an attempt at sending the message to the `data` object (assuming it is an object, of course). If the method doesn't exist, the runtime will raise an exception, but usually allow the program to continue executing.

Objective-C has many other features, but these will not be delved further into here.

[19] A selector can be thought of as the "signature" of a method, consisting of the method name, its named variables and their types in a form defined by the compiler.

A.2 Cocoa and Interface Builder

Cocoa is designed around the Model-View-Controller (MVC) paradigm, where (at least conceptually) every object has one model (providing the data), a view (showing one of potentially many different views of the data) and a controller (which handles things like adding data to the model, user interaction, etc).

The user interface for most Cocoa applications are created using Apple's Interface Builder tool. The tool allows the developer to construct the interface visually, using objects from Cocoa as its building blocks. The interface is stored in a `.nib` file, which in reality contains flattened Cocoa objects. When the application starts, the objects are "revived," and receive an `awakeFromNib` message from the Cocoa runtime.

As Interface Builder allows the developer to construct and import classes of her own, these classes can also be used in constructing the interface. By instantiating controller objects[20], connections can be made between objects in the interface, and objects controlling the interface. For instance, a button can be connected to an action method in a controller class - this is done in WPublish, when the Add server button is clicked. The button sends a message to its "target," specifying itself as the sender. WPublish receives the message, and deals with it accordingly by allowing the user to add a server.

In addition, connections can be made the other way, by specifying "outlets." Outlets are just a convenient way of hooking a class up with objects in the user interface, allowing the controlling class to perform additional initialization when it receives the `awakeFromNib` message. This can be thought of as assigning an object to an instance variable in the controller's implementation. A common outlet would be a text field that contains some sort of status message. The interface for both connecting actions and outlets in Interface Builder is simply to control-drag from one object to another, specifying whether the connection is an action or an outlet.

B Source code

This appendix contains all the source code developed as part of the window sharing system. Please note that while some effort has been made to make lines fit within the "standard" page margins, this style has not been used everywhere, as the author prefers source code with longer lines. For best viewing, the digital copies should be studied, with the tab length set to 4 spaces.

Note: Source listing has been omitted for the online version of this paper.

[20] Any object can be instantiated. The use of "controller" here is only an example - the WPublish `.nib` file contains both instantiated controller objects, and instantiated view objects.