XMOVE

A PSEUDOSERVER FOR X WINDOW MOVEMENT

Ethan Solomita, James Kempf, and Dan Duchamp

ABSTRACT

We describe the design and implementation of *xmove*, a tool that supports the movement of an application's X11 interface from one X server to another. *xmove* is implemented as a pseudoserver. A pseudoserver is a program interposed between client and server that looks like a server to the client and looks like a client to the server. The *xmove* pseudoserver operates by spying on all X protocol messages and retaining any information needed to move a client from one server to another.

Compared to other pseudoservers that provide similar functionality, *xmove* has several advantages: an interface to provide application-specific mapping of properties and selections from server to server; superior, though limited, handling of color; a heuristic means of detecting and handling applications that open several connections to a server; suspended animation for clients without a server; and more modular and simpler deployment and use. Furthermore, performance analysis indicates that a pseudoserver does not introduce unacceptable overhead into common X protocol operations. However, there are inherent disadvantages to the pseudoserver approach, and we discuss the pros and cons of implementing window movement with a pseudoserver as opposed to other approaches, such as toolkit changes or server extensions.

I INTRODUCTION

The X Window System provides a client-server model in which clients can connect to a server that controls a display located on a remote platform. This connection is intended to last for the lifetime of the client, and there is nothing in the X protocol or in the X11R6 implementation that is intended to support the dynamic movement of clients between servers. Consequently, this capability—which we call "window movement"— is not common and must be implemented as an extension of existing X11 software.

As the workplace, and the world in general, becomes more display-rich, the desirability of window movement becomes evident. Requiring users to reinstantiate their user interface when they move to a new display makes work cumbersome and, in some cases, impossible. Growth in trends such as telecommuting and portable computers suggests that, in the future, users will be faced increasingly with the prospect of using several displays in the normal course of their work, and thus the utility of a practical window movement tool might reasonably be expected to grow over time.

Accordingly, we have set out to build a tool to provide window movement. In order to make it as widely usable as possible we decided that this tool should not require modifications to either clients or servers. We accomplished this by using a design known as a *pseudoserver* [Gust88]. As shown in Figure 1, a pseudoserver is an intermediary positioned between client and server. This design allows for the capture, interpretation, change, and redirection of the X protocol messages exchanged between client and server. Our pseudoserver for window movement is called *xmove*.

FIGURE 1: *PSEUDOSERVER ARCHITECTURE*

Specific goals we set for *xmove* include the following:

- There should be no changes to either the server or Xlib, and clients should not need to be recoded or relinked.
- The new server need not be specified when the client starts; it should be selectable just before movement.
- The user should be able to move individual clients without moving the entire display.
- Features controlled by the graphical user interface (GUI) model, such as top-level window frames or drag-and-drop, should work on the new server.
- Common display configurations, such as monochrome and 8-bit color, should be supported without excluding extension to less common configurations.

Implementing window movement without modifying the client implies that the client must be shielded from all environment changes. All server-dependent information must be converted between the client's expectations and the current server's expectations. Server-dependent information includes resource IDs, colors, and atoms, all of which are represented by identifiers chosen by the server. A new server is likely to choose different IDs, so all messages between client and server must be checked, and identifiers mapped.

Other issues that make window movement a thorny problem include:

- Clients that act on resources owned by other clients. The most frequent cause of this is an application that maintains multiple connections with the server. Each connection appears to be an unrelated client to *xmove* and the server, yet the connections may manipulate resources created by each other.
- Interclient communication (ICCCM [Rosenthal91]) that uses message formats not specified by the X protocol. The contents may contain server-dependent information that needs to be mapped.
- Displays with different characteristics, such as color resolution.

Examination of these and related issues form the majority of the paper.

II Design Options

Deployment of a pseudoserver is not the obvious choice for a means to support window movement. Implementations can be constructed at any level within the client or server, as well as between client and server as found in a pseudoserver. Thus, support could be provided by the client itself, a toolkit used by the client, Xlib, or the server.

The possibilities for having several entities working together to provide mobility are nearly unlimited. Without intending to limit the available design options, we discuss below some of the alternatives.

Client-Implemented Mobility

The most straightforward approach is to have the client capable of moving itself. This would require no translation of messages at lower levels and allow the client maximum control over its environment. Only the client itself can optimally adjust to a new server with more limited resources. Additionally, the client could discover that a new server has additional features not found on its prior server.

A likely implementation would involve the client restarting its connection on a new server, just as if it had originally been started there. It would then have to recreate its windows just as the user left them on the prior server. Although this involves reuse of code that already exists within the client, it requires the code to be utilized in ways not originally intended and thus is likely to involve a significant amount of reworking.

The individual application authors are the ones responsible for implementing mobility within the client. Although this could yield the best results, it is unlikely to be considered a priority for most authors, particularly when support can be provided at other levels not requiring their knowledge.

Server Extensions

A server can be extended to forward a client's messages to a new server. In this position, the server performs the role of an intermediary. It would be required to translate server-dependent information between the client and its current server, as well as to compensate for functionality not present in a new server.

This approach is similar to a pseudoserver in a number of respects. It requires every message between a moved client and its current server to be forwarded before reaching its destination. It must also discern the client's intentions at the protocol level, which makes it difficult to render the client properly on a new server with fewer resources. Server extensions are also far from standard, making compatibility across platforms very difficult. Many servers are also proprietary, rendering modifications undistributable.

The primary advantage of server extensions over a pseudoserver is that an unmoved client needs no more processing than a client using a traditional server. A secondary advantage is that, because the server can process all of the client's messages and maintain the client's bitmaps, a client that is moved to a display with limited colors can be returned with no loss of image quality.

An alternative method of extending the server could involve extensions that allow a server to dump a client's complete state and to load such a dump. A server, upon loading a dump, would then be responsible for translating the client's requests before processing them. This extension could be combined with minor changes in Xlib that would respond to an event telling the client to move. Xlib would then retrieve the dump from the server, close its connection with the server, open a connection to the new server, and transmit the dump.

This technique has the advantage of not requiring an intermediary between client and server after the client has been moved, since the connection with the original server is closed after the dump is retrieved. The primary disadvantage is that it requires modifications to Xlib (see below). This method also has the disadvantage of never having been implemented, so the practicality of a dump has never been tested.

Toolkit Support

Given the difficulty of mobilizing large numbers of application programmers to implement window movement, a toolkit approach is attractive. Since toolkits may hold detailed and high-level information about the state of the client's interface, a toolkit has the next best understanding, after the client, about why and how resources are created and shared, and how they should be moved.

Because of the toolkit's knowledge of the purpose of its widgets, the toolkit can adapt those widgets to different display environments. Most toolkits also implement a number of look-and-feel issues, such as drag-and-drop, which may involve interclient communication. The toolkit could seamlessly handle the transition to a new server in such a situation.

Another important advantage of toolkits, besides their higher-level knowledge of a client's environment, is that they can be designed to allow those clients that wish to be mobility-aware to know that they have been moved, and adapt in any ways needed. Such an implementation has been developed [Jacobi92].

Unlike a pseudoserver implementation, the above can be implemented without having to resort to storing the client's state and modifying its communications. However, clients do not typically use a toolkit exclusively and often make calls directly to Xlib. Because of this, a toolkit implementation is limited to moving those entities it alone is responsible for modifying. In order to implement complete support, the Xlib layer would need to be modified as well.

Another disadvantage is that applications that are statically linked with their toolkits would not be able to use client mobility until their authors relinked with the modified toolkits. This violates our original goal of supporting as many clients as possible.

Finally, toolkits are moving targets in that they are subject to revisions by their developers. Unless the modifications for mobility are actually accepted by the toolkit authors, the modifications would have to be reintegrated into the toolkits for each new toolkit release.

Xlib Support

The Xlib library could be modified to store the client's state and recreate it on a new server. This implementation has much in common with pseudoserver and server-extension–based implementations in that it involves monitoring and modifying the client's messages at a level which parallels the protocol.

The primary advantage of an Xlib implementation is that it would not require an intermediary between client and server, because Xlib could shut down the connection and restart it elsewhere. This is assuming, however, that the client is not using the connection directly, such as in a `select()` function call.

Just as with toolkits, Xlib's disadvantages are that it is often statically linked into clients and that it is a moving target undergoing regular revisions. Unless the X Consortium adopted the modifications for mobility, they would have to be reintegrated into Xlib for each revision.

Pseudoserver

A pseudoserver listens for new clients as would a standard X server, using an otherwise unused display name such as `machine:1` (as opposed to `machine:0`). When a pseudoserver receives a connection from a client, it opens a new connection to the server. It then serves as a pipeline, forwarding and possibly translating messages between client and server. Since a pseudoserver gets to see all protocol messages, it can record and/or modify the information necessary for implementing window movement. Pseudoservers have been used to add a variety of new features to X applications, from collaborative work to an audio interface on existing applications for the blind [Edwards93].

The primary advantage of a pseudoserver is that it meets the goal of not requiring modification to existing executables. Its primary disadvantage is that many aspects of client operation must be deduced at the protocol level—something that is in many cases difficult and in some cases impossible. Another disadvantage of any pseudoserver architecture is that it adds overhead. In a round trip from client to server, it represents a third process, introducing additional context switches and protocol processing.

Despite these potential drawbacks, we decided to use a pseudoserver implementation due to its unparalleled compatibility with existing applications and servers. As shown in the performance section, most delays due to protocol interpretation are unnoticeable. The low-level nature of the X protocol does preclude implementing some advanced features—for example, automatically adjusting colors in adjacent windows that end up being the same on a new server that has fewer colors. However, the information contained in the X protocol is sufficient to efficiently implement most common cases.

III Related Work

There have been a few implementations of window movement using the toolkit approach. Trestle [Nelson91] is a toolkit that can move shell windows from one display to another. XTk [Jacobi92] is a window toolkit that allows window widgets to move from one server to another. XTk introduces an additional step in widget creation: screen binding. Screen binding binds a widget to a screen without geometrical layout. This allows the widget's geometry negotiation to account for screen geometry should it be moved. An additional step is also introduced in widget shutdown, to maintain widget structures in case they should be moved to another screen.

Most related work has focused on the problem of collaborative work—that is, sharing a single client simultaneously among multiple users. A tool for this is called a MUX, since it multiplexes the client's output onto several servers. For MUX implementations, the pseudoserver approach has been in use for several years, with a number of academic projects [Abel-Wahab91] [Altenhofen90] and commercial products in existence. Hewlett-Packard's product, SharedX, is implemented with extensions to the server. Any client that connects through an HP server can be resent to any other server. Unfortunately, SharedX is proprietary and has not been released for non-HP workstations. Sun Microsystems has a new pseudoserver product called "ShowMe SharedApp." Sun promises to add support for many features in the future but does not currently support black-and-white displays, custom colormaps, or drag-and-drop. It is also a commercial product and available only on a limited number of systems.

Baldeschwieler et al. [Baldeschwieler93] describe their experience with four pseudoserver implementations of MUX tools. Two of these (XMX and Xmux [McFarlane91]) are unable to recreate the client's state on a new server and so are unable to move clients to new displays after the client has been started. Since this is key to *xmove*'s functionality, these two tools are not comparable.

The other two pseudoservers are XTV/2 [Abel-Wahab91] and ShX [Altenhofen90]. XTV/2 is comparable, but the tests by Baldeschwieler et al. revealed that XTV/2 suffers some serious limitations: it is unable to run FrameMaker (an application that opens two connections to the server), it fails during cut and paste, and it is unable to display simultaneously on both a color and a monochrome system.

ShX was originally implemented by modifying the X11R3 version of Xlib. Since its original release, ShX has been updated, but these updates have not kept pace with the X releases. ShX is currently based upon the X11R5 release. Even if ShX were to be updated to the current release of Xlib, this problem would recur on a regular basis. According to Baldeschwieler et al., ShX also seemed to suffer many of the same limitations as XTV/2. It had problems with cut and paste and was unable to move a client to a new server with limited color availability.

xmove has several advantages over these systems:

- *xmove* provides support for the correct mapping of property and selection information. Currently supported are ICCCM and OPEN LOOK [Sun89], including drag-and-drop [Marks91].
- *xmove* provides color support for clients moved to new servers with limited color availability.
- Applications that open multiple connections to the server, such as FrameMaker, are heuristically recognized and supported.
- Movement can be directed onto a specific screen of the new display. This implies that movement between screens on the same server is also possible.
- *xmove* is the first example of a pseudoserver designed exclusively for client mobility, as opposed to collaborative work. Therefore, by using *xmove*, the user is relieved from the need to "buy in" to a particular collaborative work philosophy and mechanism simply in order to get the capability for window movement. Furthermore, *xmove*'s smaller size provides a better starting point for continued development.

IV START-UP AND COMMUNICATION

xmove is started by a user and runs with user privilege. At start-up, the user gives *xmove* the name of a "default server," and *xmove* takes the name of an unused display on its platform. For example, if *xmove* runs on workstation "spruce" and the X server on that machine is `spruce:0`, then *xmove*'s default server might be `spruce:0` and *xmove*'s name might be `spruce:1`. Once *xmove* is running, the user specifies it as the "X server" for all clients (e.g., in UNIX, by setting the value of the DISPLAY environment variable to `spruce:1`). *xmove* listens for new connections on the X server port that corresponds to its name (e.g., if *xmove* is `spruce:1`, then the port is 6001).

A small but important performance optimization for any pseudoserver is to specify the TCP_NODELAY socket option [Stevens94] on all connections. A TCP optimization called the Nagle algorithm causes an application that has sent out a small packet to delay sending its next packet. This assists in buffering but hurts connections that require "real-time" response. Setting the TCP_NODELAY socket option turns off this optimization. We observed substantially improved interactive performance after specifying this option.

Upon receiving a new connection, *xmove* opens a socket connection to the default server. All client requests are sent through this socket, and any replies, events, or errors received from the connection are forwarded to the client. This is the standard implementation for a pseudoserver, but *xmove* presents a complication. *Xmove* occasionally finds itself in the position of needing more information before it can continue, requiring it to send requests to the server.

The reason that this presents a difficulty is because the X server maintains a sequence number which counts the number of requests which have arrived along a connection. If *xmove* were to simply insert its requests, the client would be confused by future sequence numbers. A simple solution would be to increment a signed integer after each insertion and subtract that integer from each sequence number on its way from server to client. However, since an event can be sent by the server to the client at any time, it is possible that the new sequence number was generated before the latest insertion was processed. This can lead to the modified sequence number being smaller than a previous one.

Initially *xmove* was designed to open two connections to the server for each incoming connection. If *xmove* were to receive a client request that it could not process without first sending its own private requests to the server, it did the following:

1. Flush the primary connection by sending a `GetInputFocus` request and wait for its reply.
2. Send the necessary requests on the secondary connection.
3. Send the client's request on the *second* connection.
4. Flush the second connection by sending a `GetInputFocus` request and waiting for its reply.

The `GetInputFocus` requests are necessary to synchronize the two connections. The X server provides no guarantees as to the order of execution between different connections. `GetInputFocus` provides a request which does nothing, and returns a reply. By waiting for the reply on a connection, *xmove* can be sure that all prior requests on that connection have been processed, and can safely issue requests on another connection which rely upon the execution of the first.

Using this method, the total number of requests sent to the server through the primary connection is preserved, thus avoiding problems with the sequence number count. The signed integer is only incremented after the reply is received. This method works, but has many problems when put into practice.

In particular, this requires *xmove* to perform two synchronizations with the X server. Were it not for *xmove*'s inserted requests, the communications would have likely been a continuous flow of data from client to server. The delays caused by the synchronizations can be very significant, and are discussed in more detail under Performance on page 18.

The secondary problem from the above scheme stems from the overhead of maintaining two connections per client. The infrastructure code within *xmove* that opens and closes server connections must all be written twice. A bug in earlier versions of the X server, including Sun's OpenWindows server, would cause the server to ignore all connections if a new connection has arrived but not yet sent its setup information. Since *xmove* has two connections to the server, a seemingly trivial change could easily lead to deadlock. If *xmove* should need to insert a request while the client has the server grabbed exclusively, the second connection will be blocked permanently. Finally, many pieces of code had to be implemented three ways: directly to the server; via the secondary connection; if the server is grabbed.

All of the above lead us to seek out a better alternative. One solution would be to remember all recent request insertions, so that any sequence number can be understood, but it was feared that the overhead from searching a table for every message from the server might be undesirable. Instead, a simpler approach was chosen which requires one connection, and one synchronization.

When *xmove* needs to insert a request to the server, first it flushes its outgoing buffer to the server, and then remembers the sequence number of the last request from the client that it has processed. Any requests may now be sent to the server, so long as the last request involves a reply from the server, that is, so long as a synchronization is performed at the end. This gives *xmove* a chance to clean up after itself by retrieving any replies or errors based on its requests. It also allows *xmove* to use the signed integer, described earlier, during normal processing, by only incrementing it after the synchronization is completed. Note that this is not necessarily an additional synchronization, because in some cases the last request xmove needs to insert involves a reply. Only if it does not involve a reply does xmove need to insert an additional spurious request in order to synchronize.

If an event, reply or error is received with a sequence number smaller than *xmove*'s first inserted request, it is forwarded to the client with the sequence number modified by the signed integer, as normal. If an event, reply or error is received with a larger sequence number, then it is either intended for *xmove*'s consumption, or it is forwarded on to the client with the sequence number set to the client's last real request, modified by the signed integer. When the synchronization is completed, the signed integer is incremented by the number of inserted requests.

V Mapping of Server-Dependent Information

Almost all messages sent between client and server contain some server-dependent information. When *xmove* receives a request from a client, it must translate all server-dependent information to meet the expectations of the current server. When *xmove* receives an event, reply, or error from the current server, it must translate all server-dependent information to meet the expectations of the client. By so doing, *xmove* is able to isolate the client from changes found in a new server.

Server-dependent information falls into four categories:

- Graphics resources maintained by the server on behalf of the client
- ICCCM requests implemented by the particular window manager or toolkit in support of a particular look and feel (for example, OPEN LOOK)
- Color
- Authorization information

How *xmove* deals with each of these is described in the following sections.

Resource Mapping

In X, graphical resources such as windows and pixmaps are referred to by 32-bit resource IDs. A 32-bit resource ID consists of two fields: a client ID and a field identifying the resource. The client ID is assigned to the client by the server when the client initially connects with the server. The remaining bits are chosen by the client when it creates the resource, with the only constraint being that the same resource ID must not be assigned twice.

Accordingly, when *xmove* moves the client to a new server, the resource bits need not be modified, but the client ID must be translated between the client's expectations (i.e., the client ID on the original server) and the new server's expectations (i.e., the client ID on the new server). This requires *xmove* to keep track of all client IDs and to perform resource ID mapping on all messages between client and server. Figure 2 shows an example in which the client originally created resource 7 with client ID 2, but when it is moved to the new server, the new client ID becomes 5.

FIGURE 2: *RESOURCE MAPPING*

It is common for the server to mention resources owned by other clients. For example, `GetInputFocus` tells the client which window is currently receiving keyboard input. This might be a window owned by another client. When the server shows such a resource ID to a client for the first time, *xmove* intercepts and changes the resource's client ID if the client would be confused by it. The client will be confused if it has already seen the ID in a different context, or if the new server uses different bits within the ID to hold the client's ID. Once the client has seen a client ID, however, *xmove* cannot change it again.

A significant difficulty arises when a client refers to a resource containing a client ID that has never been shown to the client by the server. This situation cannot be ignored, as it implies that the client is communicating with another client through a channel that *xmove* cannot observe. Were *xmove* to move one client without also moving the other, they could both fail when they tried to reference resources that were no longer on the same server. When this situation occurs, the two clients are usually two connections from the same application. Unfortunately, there is no way for *xmove* to verify that two of its connections are a single logical entity; neither the socket mechanism nor the X protocol provide enough identifying information. Yet it is essential that *xmove* understand when several connections are related. Since *xmove* is storing state for both connections, if one connection modifies resources owned by the other *xmove* must recognize this and update its state information.

In order to handle cases of multiple related connections, *xmove* uses a simple heuristic. When a client sends a request to *xmove* with a client ID that the client has not previously been shown, *xmove* checks to see if it holds both clients, and if so *xmove* assumes that the two clients are a single application. From then on, if one connection modifies resources created by the other, *xmove* will correctly make note of the changes. Also, any request to move one of the connections will move them both.

This heuristic fails if the two connections do not share any resources by the time *xmove* is requested to move one of them, or if the second connection is created after the first client has already been moved.

Examples of applications that need this special treatment include FrameMaker and *ghostview*. FrameMaker opens two connections to the server, the second of which is responsible for changing the pointer to an hourglass for the first connection's windows. *ghostview*, a PostScript previewer, opens a new connection each time it needs to draw into its primary windows, and then closes the connection once it is finished. Ghostview is a much more difficult case because, after the client has been moved, the new connection will still be sent to the original server. Handling this would require *xmove* to automatically move the connection to the new display on the spot. Although it is not impossible, this is not currently implemented.

ATOM MAPPING AND PROPERTY SUPPORT

ICCCM communication is a way for clients to communicate between themselves using the X server as an intermediary. The X protocol uses 32-bit handles called *atoms*, which clients request from a server by specifying an ASCII name. The name can be anything, but is typically descriptive of the atom's purpose. Examples are WINDOW, which refers to a window resource ID, WM_ICON_HINTS, which are hints that a client gives to the window manager regarding display of its icon, and _OL_WIN_ATTR, which is OPEN LOOK specific and is used to control window attributes. When a client is moved to a new server, it expects the same atoms to be available that were available on the old server.

xmove tracks all `InternAtom` requests from a client, and when a client is moved, the client's atoms are interned on the new server. However, handling atoms is more complicated than simply interning them on the new server. Every window can have several associated properties, each of which are identified by an atom. A *property* is a piece of data whose format is not specified by the protocol but rather by various policies utilized by clients.

As an example, most clients set the WM_NAME property on their main window to the name that they want the window manager to display in the window frame. In this case the property's contents are ASCII, which does not need to be mapped by *xmove*. However, in the case of the WM_HINTS property, the client can specify a window or pixmap ID to be used by the window manager as the application's icon. This property does contain server-dependent information and, if not mapped by *xmove* to the new server's expectations, will result in incorrect behavior on the new server. (See Figure 3.)

FIGURE 3: *ATOM AND PROPERTY MAPPING*

A similar situation exists with selections. A *selection* is similar to a property, but it allows the two applications to communicate about the format of the data. Again, because the contents of this communication are not part of the X protocol, *xmove* cannot know whether any of the selection's data fields needs translation.

The details of property and selection communication between applications is a policy issue, controlled largely by the toolkits and window managers. Toolkits use properties and selections to implement aspects of the toolkit look-and-feel (for example, drag-and-drop or window frame decoration). In order to provide flexible support for property and selection mapping, it is necessary to provide a mechanism for mapping that is independent of the semantics of property and selection use.

We accomplish this by defining an application programming interface (API) to *xmove* that provides hooks for policy-specific mapping functions. Look-and-feel implementors must supply code for implementing the property mapping functions. Using this API, it is possible to create shared libraries that perform the mapping functions for any look-and-feel policy. Shared libraries implementing look-and-feel policy support are put into a well-known directory, and *xmove* loads all such libraries when it begins execution. *xmove* currently has a library supporting most atoms defined by OPEN LOOK and the ICCCM.

Each atom name has a mapping function associated with it, and whenever the atom name is used—whether as a property, property type, selection target, or client message type—*xmove* calls the function, which takes whatever actions are necessary to translate data for the client.

xmove maintains a separation between the translation of properties themselves and the translation of property types. This separation allows unknown properties having a standard type, such as WINDOW, to be processed. The separation is implemented by putting all mapping functions into one of two lists:

- Functions for translation when an atom is used as a property
- Functions for translation when an atom is used as a property type, a selection target, or a client message type

Whenever a client first uses an atom, *xmove* searches for a mapping function from both lists. If it finds one or both, the results are stored in a per-client lookup table. Future use of the atom in protocol requests causes the functions to be retrieved from the lookup table and to be applied to the atom and its contents.

As an example of a fairly complex use of properties supported with this system, the OPEN LOOK drag-and-drop standard works after an OPEN LOOK client is moved to a new server. A client request to create a property listing its drop sites is intercepted by *xmove,* and a mapping function is called to map all the window resource IDs for the client requesting the drop site list property. During the actual drag-and-drop process, when the client sends or receives a `ClientMessage` to trigger the drop, a mapping function is called that translates the atom specifying the selection holding the data.

Additional support is required to handle temporary properties. After a `ConvertSelection` or `SelectionRequest` event, some properties are used temporarily to store data of a certain type. Later, these properties are destroyed. In addition, the standard ICCCM conversion target MULTIPLE can assign temporary meanings to several properties at once. Temporary properties are supported by a third mapping function in the client's lookup table. Unlike the other two functions, the temporary property mapping function is placed in the table at the time of the `ConvertSelection` or `SelectionRequest` event rather than when the atom is first used. Upon property access, the temporary mapping function is called to translate the temporary property. The temporary property-mapping function is automatically removed from the table after the client either deletes or changes the property. This heuristic, although imperfect, serves for most needs.

Color Support

Supporting color among heterogeneous displays is probably the most difficult task *xmove* must perform. There are so many potential combinations of displays, visuals, and client needs that *xmove* can not handle all eventualities. Our approach has been to start with the most common configurations and expand from there.

Moving Colors Correctly to a New Server

As with other resources, *xmove* tracks all X protocol traffic associated with colors and maintains information about client allocations. After a window has moved, color translation involves using a lookup table to translate color cells between the client and the new server. Colors need to be translated during access to a variety of graphical resources (graphics contexts, window and pixmap contents, and window attributes), as well as during color allocation requests and replies.

xmove maintains a translation table for all colors, including the RGB values for the color or the color name and whether the color is allocated as writable. Figure 4 shows how color mapping takes place. When a request deals with the entire contents of a window or pixmap (such as with

FIGURE 4: *COLOR MOVEMENT SUPPORT*

`GetImage` and `PutImage`), every pixel must be mapped using this table if the image is greater than 1-bit deep. This is very expensive.

When allocating the client's colors on a new server, the exact shades may not be available. In this case the server typically provides an approximation. Since *xmove* stores the color actually requested by the client, it can ensure that once the client is moved to a server with sufficient color availability, the originally intended colors will be restored.

Because *xmove* cannot inform a client that window movement has occurred, if a client is moved to a server in which fewer colors are available, the client's windows may be displayed with lower color resolution. For example, if a client using ten separate colors is moved to a server with only eight colors, then two colors that were originally distinct are now rendered identically. As a result, areas of the client's GUI could become indistinguishable. Also, any graphics stored by the client in pixmaps will still possess fewer colors even if the client is moved back to a higher-resolution machine.

This problem also occurs when moving to a black-and-white (1-bit) display from a color or gray-scale display. *xmove* must decide whether the color's intensity is closer to black or to white and then choose accordingly. This can lead to an application that cannot be understood, as text and imagery may be all black or all white. Typically this does not occur, however, as neighboring colors are usually of different intensities, even on color displays.

CLIENT COLOR ALLOCATIONS ON A NEW SERVER

When a client is moved to a new server and then allocates a color, *xmove* must first create a mapping between the returned color cell and one that the client can be shown. For example, if the client believes it owns color cell 4, which on this server is really color cell 5, and the client's new color allocation returns color 4, *xmove* must provide an unused number to show the client. Otherwise the client will believe it has allocated color cell 4 twice.

It is not uncommon for clients to intentionally allocate the same color several times, often because one part of the client does not know what another part has allocated. In these cases *xmove* must notice this and, instead of making an additional entry in its color-mapping table, note that the color has been allocated an additional time.

If the new server has limited color availability, it is possible for the client to allocate colors that the new server cannot accommodate. For example, if the client has allocated blue and now requests turquoise, but the limited server can provide only blue, *xmove* maps the returned color cell into an unused color number instead of into the one for blue. It would not be unacceptable for *xmove* to admit to the client that it could not obtain the new color. However, if *xmove* tells the client that it received a new color, the client may look better once it is moved to a new display with more color availability, because *xmove* can then render the colors distinctly.

In practice, none of the preceding problem areas have proven to be serious. Most standard window-based applications are designed with a particular look-and-feel, are coded with a toolkit supporting that look-and-feel, and are run with a window manager implementing that look-and-feel. As a result, the application designers constrain their use of color to what the look-and-feel guidelines recommend.

Naturally, applications that make more sophisticated use of color may exhibit reduced color quality when moved. The ability to support color allocation as we just described makes it possible, for example, to display an image on a color display, move it to a 1-bit display, and modify the color scheme. It also allows for IslandDraw to be started on a color display and moved to a gray-scale and 1-bit display without loss of functionality.

Custom colormaps

Another significant aspect of color support is the custom colormap. The colormap itself presents no difficulty, because its colors can be allocated using the techniques described above. The problem lies in identifying which colormap a pixmap or graphics context (GC) will be using.

Pixmaps and GCs are not directly associated with a colormap. Their data is interpreted only when it is applied to a window, because the window is associated with a specific colormap. This means that it is necessary to guess which colormap will be used with new pixmaps and GCs. *xmove* guesses a colormap by examining the drawable specified when creating the pixmap or GC and assuming its colormap. This guess is not always correct, particularly with clients that always specify the root window as the drawable.

If a pixmap or GC is used and its assumed colormap is found to be wrong, it is necessary for *xmove* to convert its contents to the appropriate colormap immediately, before allowing the client's request to go through. This requires *xmove* to synchronize with the server both before and after conversion. For a GC, the conversion involves modifying only a few color cell values. For a pixmap, however, it involves retrieving the pixmap's image, mapping every pixel, and storing the converted image. The synchronization aspect of the conversion is very time consuming, as reported in the performance section. Note that this problem should occur only upon initial use of a pixmap or graphics context, thus limiting its overall impact upon performance.

Limitations on the Client

xmove imposes three sets of broad limitations on color use for applications that want to move windows:

- TrueColor and DirectColor visuals (ie. 24 bit displays) are not directly supported.
- Applications can use only the default visual and cannot allocate color planes.
- Applications started on a 1-bit display can be moved only to other 1-bit displays.

The first two limitations were imposed as a matter of expedience; the third is a consequence of display dependencies in the X protocol and library interfaces.

Read-only visual types (`StaticGrey` and `StaticColor`) were not initially supported because some clients, under these visuals, simply read in every color cell and its color value when they initialize, assume the colors never change, and so never actually allocate any colors. Note that this problem only occurs when the client is started on a read-only visual, as opposed to being moved to such a visual, since the client is made by *xmove* to believe that it is always on its initial server.

The solution to this problem with read-only visuals was found by modifying processing of the reply to QueryColors. Any color which the client sees in this reply is treated as allocated. As well, even if a client allocates a color and later frees it, *xmove* continues to mark it as allocated, in case the client tries to use it again anyway.

`TrueColor` and `DirectColor` visuals are generally used on 24-bit displays. Instead of having a single colormap as is typical on 256-color displays, they use three separate colormaps: one each for red, green, and blue. Enabling *xmove* to support these visual types is possible but would require further research. An important distinction here is that the problem stems from the client believing that the visual is 24-bit. If the client is started elsewhere, and later moved to a 24-bit display, the client should continue to be functional.

The second restriction limits clients to the default visual. Very few clients utilize other visuals, and it is felt that there are other, more important areas of development for *xmove*. Multiple color planes are also rarely used, primarily by clients requiring animation abilities. Because of the extreme difficulties in supporting color planes across multiple servers, and because of their limited use, we do not foresee implementing support for allocated color planes.

The limitation that prevents moving an application from a monochrome to a color display might seem counterintuitive, in that it should be simple to transfer just black and white. Unfortunately, because of a detail in the X protocol, this is not so. There are some uses for pixmaps that require them to be 1-bit deep, regardless of the display's depth (e.g., stipples). On a color display, these pixmaps are easy to distinguish because they are 1 bit while all others are 8 bits. A client started on a 1-bit display, however, creates everything as 1 bit, making it impossible to know what must be kept at 1 bit on the new server and what should be mapped to 8 bits. Worse yet, a client started on a 1-bit display and moved to an 8-bit display could use a pixmap such that for some operations it would need to be 1 bit and for others it would need to be 8 bits.

In summary, *xmove* reliably moves clients between displays having equivalent availability of colors, and between 1-bit displays. Other movement is also supported, with potential degradation in quality.

Authorization

Unlike other server-dependent issues, authorization does not depend upon communication between client and server. Rather, it depends upon the machine on which the client is running, and on the user who is running the client. *xmove* supports host-based (i.e., *xhost*) and MIT-MAGIC-COOKIE security mechanisms. Host-based authorization is the simplest of the mechanisms and is available to all users of X. MIT-MAGIC-COOKIE is a simple authentication protocol that is available to many users of X.

Under *xhost*, the server maintains a list of authorized hosts. Under MIT-MAGIC-COOKIE, the server adds a 128-bit key to a file readable only by the user who started the server. A new client is permitted access if it either comes from a host on the authorized list, or if it passes the appropriate key in its start-up message. Typical configurations for a server include allowing global access, allowing only local clients, and allowing only clients with the appropriate key. The first configuration provides no security, the second provides security but requires all clients to be local, and the third provides security but requires a server running MIT-MAGIC-COOKIE.

Upon invocation, *xmove* looks up the key for its default server, if any, in order that *xmove* may legitimately contact it. If one was found, *xmove* then adds a key for itself into the user's directory. By so doing, any client that the user starts and sends to *xmove* will need to present that key before it will be granted access.

Security is provided at two levels. For a new client, *xmove* decides itself whether the client is authorized. The first action *xmove* takes is to make a new connection to the server and issue a `ListHosts` request, which returns the list of all authorized hosts, or simply says that the server is not authenticating new clients. If the client is not connecting from a host on the authorized host list, and the server is authenticating new clients, then *xmove* waits to see if the client transmits a magic cookie, and if so compares it against the key that *xmove* placed into the user's home directory. If this check still fails, *xmove* rejects the client and closes the connection. Otherwise it sends the client through to the server with a key it knows the server will accept.

xmove is not as strict when moving a client to a new server. It relies on the new server to accept or reject the client. The reason *xmove* is not as strict is because *xmove* might be running on a host that is permitted access to the new server, yet the client itself might not be. It should be remembered that, even in such a case, the client had to have authorization to access *xmove* in order to request that the client be moved.

If the new server is using MIT-MAGIC-COOKIE, and it is still in its standard configuration of having an empty host list, then there can be no security holes during client movement. The host becomes irrelevant as the only access to the new server is via the correct key. The key sent to the new host is provided automatically by the user who issues the movement command to *xmove*, thus providing a key appropriate to the user issuing the request. This allows a second user with greater permissions to move a client to a server its original user could not.

A complication occurs when *xmove*'s default server crashes. When the server comes back up, assuming it uses MIT-MAGIC-COOKIE, it will have a new key. Since *xmove* may not know that the key has changed, it will continue to use the old key when it tries to connect to the server to issue the `ListHosts` request. When *xmove* finds its attempt to connect to the server has been rejected, it rechecks the key file to see if the server has changed keys. If *xmove* still fails to connect on its second attempt, it gives up, since it does not have access rights to the new server.

If a client tries to connect to *xmove* while the default server is unavailable, *xmove* will discover this when it tries to connect to issue the `ListHosts` request and the socket calls fail. If *xmove* were to simply reject the incoming client at this point, it would also reject requests from an *xmovectrl* client. This would block the user from moving clients that are on other servers, or are in suspended animation. So *xmove* must wait until the client identifies itself, at which point it will be rejected if it is anything except an *xmovectrl* client.

VI The Window Movement Process

xmove has two main tasks during normal operation:

- It translates between client and server expectations.
- It stores information about the client's state that cannot be retrieved from the server in the future.

The stored information regarding a client's state is not complete. When a client is to be moved, *xmove* goes out to the server and retrieves any state that it does not already have. The request to move a client is sent to *xmove* by using a utility program, *xmovectrl*, that communicates with *xmove* through a separate control protocol. *xmovectrl* provides the following services:

- It lists all the clients currently under the control of *xmove*.
- It moves one or more clients to a new display.
- It places one or more clients into suspended animation.
- It specifies a new default server for all future client connections.
- It requests that *xmove* close all connections and exit.

When *xmove* is asked to move a client to a new server, it retrieves state information from the old server, combines it with what it saved, and recreates everything on the new server. Recreating resources on the new server involves a series of five well-ordered stages that build upon each other. Each stage relies only on resources created by previous stages, allowing *xmove* to move multiple clients in unison if one uses resources created by another. Each stage is applied to all related clients (See "Resource Mapping" on page 9) before proceeding to the next. If an error occurs in any stage except the last, that stage can cancel the move and specify a text string to be presented to the user. The last stage places *xmove* into a state consistent with the new server.

The five stages are as follows:

Connecting to the New Server

The first stage involves establishing a connection to the new server and retrieving the server's defaults from the setup reply. A GC with a resource ID of zero is created, as well as a 1-bit deep pixmap and GC.

If the movement command from *xmovectrl* did not specify a display number (i.e., `machine` vs. `machine:0`), then *xmove* tries to find another *xmove* running at `machine:1`. If none is found, *xmove* uses the server at `machine:0`. If one is found, then a communication path is established that involves *two* instances of *xmove*; that is, protocol flows along the path client-

*xmove*1-*xmove*2-server and vice versa. The second *xmove* simply passes through all messages without processing them. All logic and processing described previously in this paper takes place in the first *xmove*.

The purpose of the second *xmove* is to improve the user's interface to *xmove*. By having all clients go through a second *xmove* process on the new server's host, the server's *xmove* accumulates a complete record of all the clients that have been moved onto it. This record includes the client's name, host, and first *xmove* process. The value of the record is mostly informational, permitting users to understand which clients are running on which machines and where the clients originated. If the second *xmove* is told to move ("deport") a client being forwarded from another *xmove*, it passes on the instruction to the first *xmove*, thus preventing a client from passing through chains of *xmoves*.

This stage can fail for any of the following reasons:

- The client has grabbed the server exclusively.
- The new server either does not exist or cannot be contacted.
- The new server rejected *xmove*'s connection.
- The user is trying to move a monochrome client to a multi-bit display.

RESOURCE CREATION

Stage two creates resources that have no dependencies: custom colormaps, color cells, atoms, fonts, and pixmaps. The resource mapping tables are updated to remove transient client information and to coordinate clients being moved in unison. All pixmaps are created using the root window as a drawable. One-bit deep pixmaps remain 1-bit deep on the new server, but pixmaps of greater depth are set to the new server's default depth. Any color allocation to which the server returns an error is allocated as either black or white, depending upon the color's intensity. If a font cannot be found it is left unallocated, and a warning is displayed to tell the user that the client may not be stable.

This stage can fail for any of the following reasons:

- The client has allocated custom colormaps and the new server refused to create them.

MORE RESOURCE CREATION

Stage three creates resources having dependencies on resources created during stage two: glyph cursors, windows, and pixmap images. When creating windows, all attributes are applied except the cursor, because the cursor may not have been created yet. All properties are copied onto the new window, with mapping functions called as necessary.

In order to move a pixmap image, *xmove* must first read the image from the original server. If it is a 1-bit–plane image, it is not translated, because the image may be intended as a simple bitmask (e.g., stipple) where the bits indicate on or off, not black or white. Otherwise, *xmove* creates a second image structure with the appropriate attributes for the new server and scans the pixels, retrieving each color value, mapping it into the appropriate value for the new value, and storing it into the new image.

Window Stacking Order, Creation of GCs and Cursors

Stage four uses `QueryTree` to retrieve the window stacking order from the original server, and `RaiseWindow` to set the stacking order on the new server. It also creates GCs and cursors. The graphics context with resource ID 0 created in stage one is deleted so that the real one, with the characteristics set by the client, can be created.

For all cursors stored by *xmove*, *xmove* also stores the bitmaps containing the cursor's image and mask. When *xmove* creates the cursor, it first stores those images to the new server in pixmaps, then creates the cursor, and then frees the pixmaps. The pixmaps originally used by the client to create the cursor cannot be used here by *xmove*, because the client may have since deleted them or changed them, thus *xmove* must remember their contents as of the time the cursor was created.

Cleanup

Stage five recreates the client's passive grabs on keys and mouse buttons, gets any selections the client already owns, changes the cursor attribute for windows, maps windows, and updates internal data structures so that the new server becomes the current server.

VII Suspended Animation

A desirable feature for client movement is the ability to move the client from its current server without immediately moving it onto another server. This feature, which we call suspended animation, places a client into limbo. This can be useful if a user is leaving a shared workstation and does not need their applications until later. It can also be used if the user has advance notice that their workstation will be brought down.

This has a significant impact upon the implementation of client movement described above. Normally, when *xmove* is to move a client to a new server, it must go to the current server in order to retrieve some of the client's state which *xmove* is not already holding. Because suspended animation involves severing the connection to the server, *xmove* must retrieve that state immediately, so that it will be available later when the user requests the client be moved to a new server.

This state falls into three categories. The simplest is verifying which selections the client currently owns. The second is the retrieval of the image from all pixmaps the client currently possesses. The third is the retrieval of all information about the client's windows, including attributes, geometry, stacking order and properties.

Suspending a client amounts to roughly one half of the steps involved with moving a client. It flushes the outgoing buffer to the current server, processes any pending replies, retrieves state information, and closes the connection to the server. Moving a suspended client involves establishing the connection to the new server, recreating the client's state, and storing the necessary configuration information about the new server such that *xmove* can continue to do its job.

Suspended animation was not available in the initial release of *xmove*. Its inclusion presented two principle challenges. The first was the need to restructure the client movement code such that all of the window's state could be retrieved at once and collected into a data structure. The code previously accumulated the state in a variety of places and times, whereas suspended animation needs a single function call which can do all of this at once. The code to move the client also had to be changed to check whether the client had been suspended, and if so, retrieve the state information from memory.

The second challenge was related to the infrastructure of *xmove*. Previously a client was always connected to a server, and as such it was always assumed that the client had a buffers, default processing functions, and a socket descriptor for its server. Code throughout *xmove* had to be changed to look before it leapt.

VIII PERFORMANCE

Interposing a pseudoserver between client and server has a number of effects on system performance. First, there may be a decrease in performance seen by the user. Second, there is increased load on the network if the client has been moved to a new server, thus requiring each message to be rebroadcast onto the network. Third, the machine running *xmove* experiences increased load. Finally, the server occasionally sees increased load, since *xmove* can add messages of its own in addition to forwarding all the messages sent by the client.

All of the above are inherent in the implementation of a pseudoserver. The purpose of this section is to discuss the extent to which the user of *xmove* will perceive these problems. Our results show that, for most actions whose performance are important to a user, the delays of *xmove* are negligible. During less time-critical periods, such as client initialization, delays can be more significant. A summary of our performance results can be found in Table 1.

TEST	WITHOUT XMOVE (SEC)	WITH XMOVE (SEC)	DELAY (%)	WITH NULL PSEUDOSERVER (SEC)	DELAY (%)
Frame (B&W) startup time	5.171603	5.233718	1	–	–
Frame (B&W) page redraw time	0.190847	0.190000	0	–	–
Frame (color) page redraw time	0.199122	0.207961	4	–	–
Wabi (color) startup time	3.538090	5.709424	61	4.300291	22
xv (B&W) startup time	3.047388	3.353680	10	3.209246	5
xv (color) startup time	3.761253	5.560801	48	4.704758	25

TABLE 1: *XMOVE PERFORMANCE RESULTS*

To analyze latency, we set up a testbed in which client, server, and *xmove* were each on separate hosts (Sun SparcStation-2s) connected by a common wire (Ethernet). Also on the same wire was a host running *tcpdump*, a tool that captures network packets and assigns each a time stamp that is accurate to within a few microseconds.

We measured both the startup and the page redraw performance of FrameMaker on monochrome and color systems. We also measured the startup performance of Wabi, the Microsoft Windows emulation software from Sun Microsystems, and *xv* with an 850x578 gif image. We performed these tests on a system without *xmove* and one with *xmove*, so that we could evaluate the time delay incurred by *xmove*. The results also include the effects of a null-pseudoserver, that is, a pseudoserver that does not process messages but simply forwards them to their recipients. This allows us to observe the impact of *xmove*'s need to process the messages separately from the delay inherent in the pseudoserver construct.

We discovered that the most important factor affecting performance is neither the volume of data nor the type of requests issued by the client, but rather the data flow patterns between client and server. *xmove* is effectively unnoticeable when communication is asynchronous (i.e., one party sends messages without waiting for an acknowledgment). The reason is that, in most realistic scenarios, a client exhibits substantial “think time” between consecutive asynchronous requests. Actions such as computing where to draw lines, kerning text, deciding where to place a graphic, or retrieving data from a file are common examples of when thinking may occur between messages. If *xmove* delays the processing of a message by an amount of time that is less than the think time, then the next message reaches an idle *xmove*, and the client perceives no slowdown. The total resultant delay over the course of several messages is thus constant.

A relevant example is drawing a page in FrameMaker. The server sends no events or replies back to FrameMaker until all of the requests involved in rendering the page have been received. This pattern is typical of the clients we have observed—while rendering images they do not often need information from the server. By observing packets sent from client to server while FrameMaker drew a page, we were able to determine common timings. Each request burst from FrameMaker consisted of between 2000 and 2500 bytes of data. The think time between bursts varied between 8 and 20 ms, while the packet transmission delay introduced by *xmove* varied between 2 and 4 ms only. Thus each burst from the client is received by the server before the client sends its next burst.

This analysis explains why the canonical X server performance measurement tool, *x11perf*, is inappropriate for measuring interactive performance. *x11perf* was designed to measure server throughput; it does this by bombarding the server with requests as rapidly as possible. Because it does not pause between requests, *x11perf* represents the communication profile of few clients, mostly those that perform rapid animation (which typically either precompute their data or have hardware support). The behavior of *x11perf* also exposes TCP window management algorithms, since *x11perf* must write requests faster than the server can process them. This leads to the server's TCP window continually being filled and emptied, requiring the client to pause transmission while waiting for acknowledgments.

The analysis presented above addresses the situation where the data is traveling primarily in one direction. In contrast, *xmove* can introduce noticeable delay when communication between client and server is synchronous. In this case, the entire delay of *xmove* is felt by the client.

An example is provided by the start-up of Wabi. During initialization, Wabi attempts to allocate a set of 256 standard colors that match the colors typical of a Microsoft Windows environment. Wabi allocates one color at a time, waiting to verify that the prior color was allocated successfully before attempting to allocate the next. When Wabi runs without *xmove*, the roundtrip time for a color allocation is typically slightly under 2 ms, with a think time between receiving a reply and issuing the next request of 1 to 2 ms. When *xmove* is introduced, it adds approximately 2 ms of delay for copying data: 1 ms for copying the request to the server and 1 ms for copying the reply to the client. Table 2 shows the data collected by *tcpdump* for a setup that does not use *xmove*, while Table 3 contains data from a setup that does. In this case, Wabi is running on `ober`, the server is on `nama`, and *xmove* is running on `cedar`; server requests are 16 bytes and events are 32 bytes.

ELAPSED TIME (SEC)	SENDER.PORT	RECEIVER.PORT	BYTES
0.000000	ober.47859	nama.X11	861:877
0.002235	nama.X11	ober.47859	7013:7045
0.003453	ober.47859	nama.X11	877:893
0.005420	nama.X11	ober.47859	7045:7077
0.006607	ober.47859	nama.X11	893:909
0.008444	nama.X11	ober.47859	7077:7109

TABLE 2: *WABI PERFORMANCE WITHOUT XMOVE*

ELAPSED TIME (SEC)	SENDER.PORT	RECEIVER.PORT	BYTES
0.000000	ober.47953	cedar.6001	861:877
0.001003	cedar.4688	nama.X11	861:877
0.002844	nama.X11	cedar.4688	7013:7045
0.003996	cedar.6001	ober.47953	7013:7045
0.005325	ober.47953	cedar.6001	877:893
0.006341	cedar.4688	nama.X11	877:893
0.008267	nama.X11	cedar.4688	7045:7077
0.009435	cedar.6001	ober.47953	7045:7077
0.010855	ober.47953	cedar.6001	893:909
0.012108	cedar.4688	nama.X11	893:909
0.013944	nama.X11	cedar.4688	7077:7109
0.015148	cedar.6001	ober.47953	7077:7109

TABLE 3: *WABI PERFORMANCE WITH XMOVE*

The average roundtrip time is just over 3 ms without *xmove* and 5.5 ms with *xmove*. This delay is significant as it accumulates over time. Fortunately, this scenario is not common. Clients do not often communicate this way, and when they do it is often during start-up or some other point when the user is expecting a delay and is thus not in a time-critical period.

Another example of synchronous communication is presented in our results from invoking *xv* with an 850x578 gif image. When displaying in black and white, the performance delay for using *xmove* was only 10 percent. When displaying in color, however, the delay increased to 48 percent. The reason was a significant number of synchronous sequences that did not exist in black and white. As with Wabi above, many colors had to be allocated. Color allocation requires a moderate amount of processing time for *xmove*, which accounts for the performance gap between a null pseudoserver and *xmove*.

A seemingly similar scenario occurs during user feedback, such as while the user is dragging an item across the screen. Here the server is initiating action by sending cursor motion events to the client. Typically the client responds to each event by sending a request to the server to resize or move some item on the screen. Although this follows the pattern of messages alternating between client and server, it is not synchronous. The server does not wait for the request from the client before sending the next event, so the server's events continue to be produced at the same interval, regardless of *xmove*'s presence.

This usage pattern is illustrated by an example from IslandDraw, an application for designing graphics. In order to draw an oval, the user moves the pointer to specify the oval's size and shape. As the pointer is moved, IslandDraw updates an image of the oval matching the pointer's current position. By watching packets on the network as the pointer was moved, we were able to observe the traffic patterns of a typical user feedback situation. Table 4 shows the data collected by *tcpdump* for a setup that does not use *xmove*, while Table 5 contains data from a setup that does. IslandDraw is running on `chesnut`, the server is on `spruce`, and xmove is running on `cedar`; server requests are 308 bytes and events are 32 bytes.

ELAPSED TIME (SEC)	SENDER.PORT	RECEIVER.PORT	BYTES
0.000000	`spruce.X11`	`chestnut.3445`	736:768
0.004150	`chestnut.3445`	`spruce.X11`	5861:6169
0.041578	`spruce.X11`	`chestnut.3445`	768:800
0.045785	`chestnut.3445`	`spruce.X11`	6169:6477
0.083148	`spruce.X11`	`chestnut.3445`	800:832
0.087469	`chestnut.3445`	`spruce.X11`	6477:6785

TABLE 4: *ISLANDDRAW PERFORMANCE WITHOUT XMOVE*

ELAPSED TIME (SEC)	SENDER.PORT	RECEIVER.PORT	BYTES
0.000000	`spruce.X11`	`cedar.1306`	288:320
0.000819	`cedar.6001`	`chestnut.3457`	288:320
0.005087	`chestnut.3457`	`cedar.6001`	2741:3049
0.006408	`cedar.1306`	`spruce.X11`	2741:3049
0.042006	`spruce.X11`	`cedar.1306`	320:352
0.043003	`cedar.6001`	`chestnut.3457`	320:352
0.047249	`chestnut.3457`	`cedar.6001`	3049:3357
0.048660	`cedar.1306`	`spruce.X11`	3049:3357

TABLE 5: *ISLANDDRAW PERFORMANCE WITH XMOVE*

ELAPSED TIME (SEC)	SENDER.PORT	RECEIVER.PORT	BYTES
0.083476	`spruce.X11`	`cedar.1306`	352:384
0.084372	`cedar.6001`	`chestnut.3457`	352:384
0.088698	`chestnut.3457`	`cedar.6001`	3357:3665
0.090040	`cedar.1306`	`spruce.X11`	3357:3665

TABLE 5: *ISLANDDRAW PERFORMANCE WITH XMOVE (CONTINUED)*

Note that the interval between pointer events from the server is nearly constant at just over 41 ms; this number is not affected by the speed or direction of the pointer.

After receiving the motion event from the server, the client responds with a 308-byte request in just over 4 ms. With the addition of *xmove*, the time between the event and the server's receipt of the client's request increases to 6.5 ms. Considering that the delay due to *xmove* is only about 2 ms, it is highly unlikely that it will be noticed by the user. As with the asynchronous client bursts described above, this delay does not build up over time. Instead, the delay is between each event and the time the server receives the client's request to update the display. Two ms is a small time period compared to a monitor refresh rate (e.g., 72 Hz), so despite the delay, the updated display will probably appear on the next screen refresh anyway.

Now we can turn to delays introduced by *xmove*'s need to process the messages it forwards. In most cases, this source of delay is not cumbersome. However, there are two situations where the delay can be significant: if *xmove* sees an atom it does not recognize or if *xmove* realizes it needs to get the image of a pixmap. In these cases, *xmove* must break the normal message flow between client and server and insert additional requests. The case where *xmove* sees an unknown atom is unusual; however, the need to get a pixmap typically happens once for each non-glyph cursor created, and far more often if the client uses a custom colormap. In these cases, *xmove* must send requests to the server for more information, so *xmove* must use the techniques for request insertion described in Start-up and Communication on page 7.

The main conclusion of our performance analysis is that the delay an *xmove* user experiences is usually unnoticeable, and the delay is only rarely proportional to the time *xmove* spends processing data. Another conclusion is that performance analysis tools such as *x11perf* give misleading measures of pseudoserver performance, as they do not simulate the usage patterns of real clients. *xmove* can cause sizeable delays in some cases, but these are usually part of longer actions, such as initialization sequences, that are not time-critical.

IX FUTURE WORK

In addition to the color limitations discussed earlier, decisions made during the initial determination of requirements imposed additional limitations on *xmove*. These were imposed in the interest of expediency, since we wanted to get a usable prototype working quickly. The following is a list of areas representing limitations (including those involved in color limitations) and, in some cases, an analysis of the prospects for overcoming them:

- A client started on a monochrome display can be moved only to other monochrome displays. Unfortunately, display dependencies in the X protocol make this problem difficult to solve.

- Differences between displays due to display resolution and pixel density are not resolved by *xmove*.
- Color planes (overlay planes) are not supported. Color planes are difficult to support and rarely used.
- X protocol extensions are not currently supported. A method for supporting extensions has been envisioned, similar to the way that properties and selections are supported.
- Authorization protocols other than *xhost* and MIT-MAGIC-COOKIE are not supported.
- Clients that make multiple connections to the server are not properly moved unless the connections refer to each other before the client is first moved.
- New servers must have the same fonts used by clients moving to them. This is not typically a problem in homogeneous environments but could be troublesome in other environments.
- Although not a limitation per se, if *xmove* fails for any reason, all clients using that *xmove* also fails, which is exactly the case with the X server itself. Additionally, because *xmove* is single-threaded, when *xmove* moves a client, it cannot process other clients until it is finished. There are a number of possible solutions to this problem, but none is trivial, the better ones are nonportable, and the priority is not high.

X Future Directions for Client Mobility

Now that *xmove* is in use, we have begun to review our decision to use a pseudoserver and consider whether this was the proper choice. Given our goal of requiring no modifications to existing executables, there was no alternative. In its form as a pseudoserver, *xmove* is able to deal successfully with most applications and displays that we personally use.

The cases that *xmove* cannot handle lead us to the question of how better to implement window movement. The only two better options lie in the two layers that can be aware of the purpose of and interrelationship between various X11 operations—namely, the application itself and, to a lesser extent, the toolkit. An implementation based in Xlib only is not attractive. Since the semantics of Xlib directly reflect the X protocol itself, Xlib's operations are too low level to handle such tasks as sophisticated color mapping of widgets. There are very few limitations that can be overcome by implementing window movement in Xlib as opposed to a pseudoserver.

If window movement is supported by the applications themselves, then we are guaranteed that the client can be correctly moved to any server on which it could have been started originally. The application will be able to deal with the disappearance of functionality on a new server, in addition to taking advantage of new functionality not present on the previous server.

The client would have to shut down its current server connection and establish a new connection exactly as it did when it was first invoked. The main changes to the application's code come from the need to reopen all windows exactly as the client left them on the old server. This implementation will produce the optimal result in all cases.

However, we think it is unlikely that converting applications will develop into a trend. Most "larger" applications will find it difficult to reopen all windows as the client left them, while most "smaller" applications will not want to add any additional code for window movement, especially considering that most of their window movement needs can be satisfied by a pseudoserver.

With regard to the toolkits, most current toolkits possess enough knowledge to correctly handle the standard aspects of the user interface (e.g., menus, buttons, and scroll bars) when moved to a server with different display characteristics. Also, a toolkit implementation could correctly handle many property and selection issues, such as window attributes and drag-and-drop. However, because toolkits do not provide all of the functionality needed by clients, most clients should drop down to Xlib at some point. Consequently, the Xlib layer, as well as the toolkit layer, will have to be mobility-aware.

Most current toolkits provide only a very limited abstraction between the client and the X protocol. The client is still required to be intimately familiar with the details of its display. If a toolkit were designed with an interface that was abstracted both from the display and from the X11 protocol, then it should also be able to seamlessly relocate a client. Even if such a toolkit were devised and its flexibility was sufficient for both simple and sophisticated clients, it would require application authors to completely redesign their code. Given the current application base, it is unrealistic to expect application authors to switch to a new toolkit and rewrite their applications just to accommodate window movement.

This leaves us with no clear answer in our search for a better client mobility implementation than a pseudoserver. Modifying clients would require significant support from software vendors, modifying toolkits would require significant support from toolkit vendors, and creating a new toolkit would require significant support from everybody.

We believe that with the currently limited demand for client mobility, the present form of *xmove* is sufficient. As the demand for window movement increases over time and as trends in hardware and software become clearer, sufficient motivation may develop to add more complete support for window movement.

XI Acknowledgments

Ari Shamash and Peter Skopp implemented the first versions of *xmove*. Chris Kantarjiev volunteered a substantial amount of his time to read and criticize earlier drafts. His comments greatly aided our knowledge and thinking and improved the presentation.

Thanks also to Richard Stevens for his book *TCP/IP Illustrated* [Stevens94], which solved most of our performance problems in one fell swoop by explaining Nagle's algorithm, its effect, and the TCP_NODELAY workaround.

Funding at Sun Microsystems for this public domain project was arranged by Dick Sillman. At Columbia, support was provided by Office of Naval Research grant number N00014-93-1-0315.

XII Availability

The source code for *xmove* may be obtained by anonymous FTP from *ftp.cs.columbia.edu:pub/xmove.*

XIII References

[Abel-Wahab91] Abel-Wahab, H. and M. Feit. "XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration," *Proceedings of the IEEE Tricom 91*, Chapel Hill, 1991.

[Altenhofen90] Altenhofen, M. “Erweiterung eines Fenstersystems fuer Tutoring-Funktionen,” Diploma Thesis at Universitaet Karlsruhe, Karlsruhe, Germany, 1990.

[Baldeschwieler93] Baldeschwieler, J. E., T. Gutekunst, and B. Plattner. “A Survey of X Protocol Multiplexors,” *Computer Communication Review*, 23(2), pp. 16–24, July 1993.

[Bormann94] Bormann, C., Hoffmann, G. “Xmc and Xy — Scalable Window Sharing and Mobility,” Eighth Annual X Technical Conference, January 1994.

[Edwards93] Edwards, W. K., E. D. Mynatt, and T. Rodriguez. “The Mercator Project: A Nonvisual Interface to the X Window System,” *THE X RESOURCE*, 7, pp. 33–54, 1993.

[Gust88] Gust, P. “Shared X: X in a Distributed Group Work Environment,” unpublished work presented at Second Annual X Technical Conference, January 1988.

[Jacobi92] Jacobi, C. “Migrating Widgets,” *THE X RESOURCE*, 1, pp. 157–167, 1992.

[Marks91] Marks, Stuart W. “Drag-and-Drop Protocol Specification: OpenWindows Version 3”, Sun Internal Specifications, July 1991.

[McFarlane91] McFarlane, G. “Xmux: A system for computer supported collaborative work,” *Proceedings of 1st Australian Multi-Media Communications, Applications, and Technology Workshop*, Sydney, 1991.

[Nelson91] Nelson, G. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Richardson95] Richardson, T. “Teleporting Mobile X Sessions,” Ninth Annual X Technical Conference, January 1995.

[Rosenthal91] Rosenthal, D. “Inter-Client Communications Conventions Manual, Version 1.1, X Consortium Standard,” *X Consortium*, Cambridge, Mass., 1991

[Stevens94] Stevens, W. Richard. *TCP/IP Illustrated, Volume 1*. Addison-Wesley Publishing Company, 1994.

[Sun89] Sun Microsystems. *OpenLook Graphical User Interface Functional Specification*. Addison-Wesley, 1989.