Checkpoints of GUI-based Applications

Victor C. Zandy and Barton P. Miller
Computer Sciences Department
University of Wisconsin – Madison, USA
`{zandy,bart}@cs.wisc.edu`

Abstract

We describe a new system, called *guievict*, that enables the graphical user interface (GUI) of any application to be transparently migrated to or replicated on another display without premeditative steps such as re-linking the application program binary or re-directing the application process's window system communication through a proxy. Guievict is based on a small X window server extension that enables an application to retrieve its *window session*, a transportable representation of its GUI, from the window server and a library of GUI migration functionality that is injected in the application process at run time. We discuss the underlying technical issues: controlling and synchronizing the communication between the application and the window system, identifying and retrieving the GUI resources that form the window session, regenerating the window session in a new window system, and maintaining application transparency. We have implemented guievict for the XFree86 implementation of the X window system. The GUI migration performance of guievict is measurably but not perceptibly worse than that of a proxy-based system.

1 Introduction

Application mobility is the ability for an application in execution to follow its user, whether the user moves from one computer to another or moves with their computer around the Internet. This mobility includes moving the executing code, network connections, access to the graphical user interface (GUI) or the GUI itself, and access to files. We are developing a *roaming applications* system, called *evict*, that addresses these issues, providing application mobility that is transparent to users and applications while requiring no modification to system code.

This paper describes *guievict*, the part of the evict system that migrates an application's GUI. An application's GUI might migrate as the application itself migrates, or independently of the application. Both types of migration require the ability to capture the application's *window session*, a transportable representation of its GUI. Window sessions are difficult to capture because not all of the state they represent is resident in the application. Portions of an application's GUI state reside in the window server, a separate system that handles the display of all GUI-based applications on the same host. Most window servers do not provide a way to extract the state of an individual application's resources.

Guievict has been implemented for the X window system [19] and has the following characteristics:

- ❏ Migration occurs at application granularity; users can select and move the GUI of individual applications from their desktop, leaving other application GUIs behind or free to move elsewhere.
- ❏ Any application program, including those based on legacy toolkits, can be migrated without modifications such as re-programming, re-compiling, or re-linking.
- ❏ Migration can be unpremeditated; users do not need to run their applications in a special way, such as by redirecting their GUI communication through a proxy.
- ❏ No modifications to window system code are required; our functionality is contained in the window server extension, which does not require the server to be re-linked, and a library that is loaded in the application at run-time.
- ❏ The guievict functionality can also be used to replicate the GUI of individual applications on demand on multiple desktop hosts, enabling multiple users to interact with a single instance of an application for collaborative work or to support remote service.

There are several essential elements to a GUI-based application (see Figure 1). The application runs in one or more processes on the *execution host* and the user interacts with it from a possibly different *desktop host*. The desktop comprises a display, keyboard, and mouse managed by a window system that multiplexes the desk-

This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, Lawrence Livermore National Lab grant B504964, NSF grants CDA-9623632 and EIA-9870684, and VERITAS Software. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

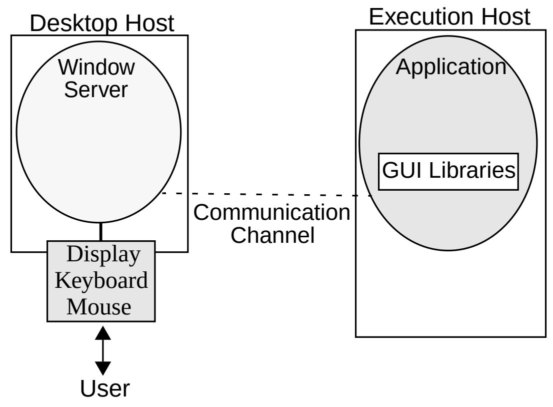

Figure 1: The elements of a GUI-based application. *The user interacts with the application through hardware managed by the window server on the desktop host. The application process executes on a possibly different execution host, exchanging GUI-related messages with the window server over a communication channel.*

top for all applications that interact with the user at that host. The window system responds to requests for GUI services (such as creating a window) sent by the application and passes notification of desktop events (such as a mouse click) to the application over a communication channel such as a network connection or (when the execution and desktop hosts are the same) shared memory. The state of the application GUI is distributed between the window system and a library (often called a toolkit) in the application process.

Our focus is on the migration of a single application from one desktop host to another. Our abstraction encapsulates precisely the GUI state of the *application* — not individual windows of the application, nor the windows of all applications that are being served for the user on the desktop host. Many previous systems have studied techniques for *session migration*, in which a user's entire desktop is migrated to another machine [7,14,16,17,23]. Our ability to migrate at application granularity has several advantages over session migration. First, it is flexible: it gives users the freedom to migrate only the applications they need at their new location, saving migration costs, and it enables users to work with multiple desktops at the same time. Second, session migration systems generally involve a virtualization layer, such as a virtual machine [7,14] or nested window server [16,17], a sort of GUI prison to which the user must redirect their applications in advance. In addition to the inherent overhead of virtualization, these layers often emulate basic and generic display hardware, preventing applications from taking advantage of enhancements or acceleration features present in the real underlying hardware. Third, a mechanism for application migration enables other useful operations, such as a *GUI attach* that allows other users to dynamically attach to and interact with the GUI of another user's application.

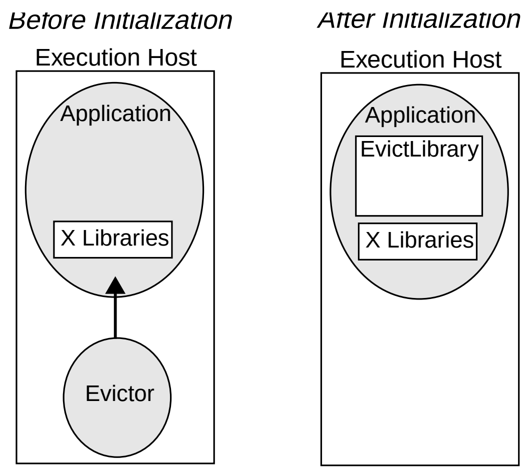

Figure 2: Initializing the application process. *The evictor stops the application process, forces it to load the evict client library, and exits.*

Guievict offers two major advantages over previous systems that perform GUI migration at application granularity, most notably xmove [22]. First, it enables users to migrate their applications without the premeditative step of redirecting the application to the xmove proxy. Although the details of the redirection can be hidden from the user in a launch script, the user still must use the script to start applications that they anticipate moving. In contrast, guievict offers application mobility features without asking users to develop new habits and foresight. Second, it simplifies process migration of GUI-based applications. Although migration was not one of its original design goals, xmove can be combined with process checkpointing to migrate a GUI-based application. However, in addition to the application process, the xmove proxy and the state of the communication channel between the proxy and the application must be checkpointed and migrated. With guievict, the window session and mechanisms that manage its migration reside in the address space of the application process, where it can be migrated along with application code and data.

To migrate the GUIs of ordinary, unmodified applications from one display host to another, we have overcome four main challenges:

❏ Dynamically taking control of the window system communication of a running program: We inject code into the application process that discovers the process's communication channel to the window server and synchronizes its communication with the server.

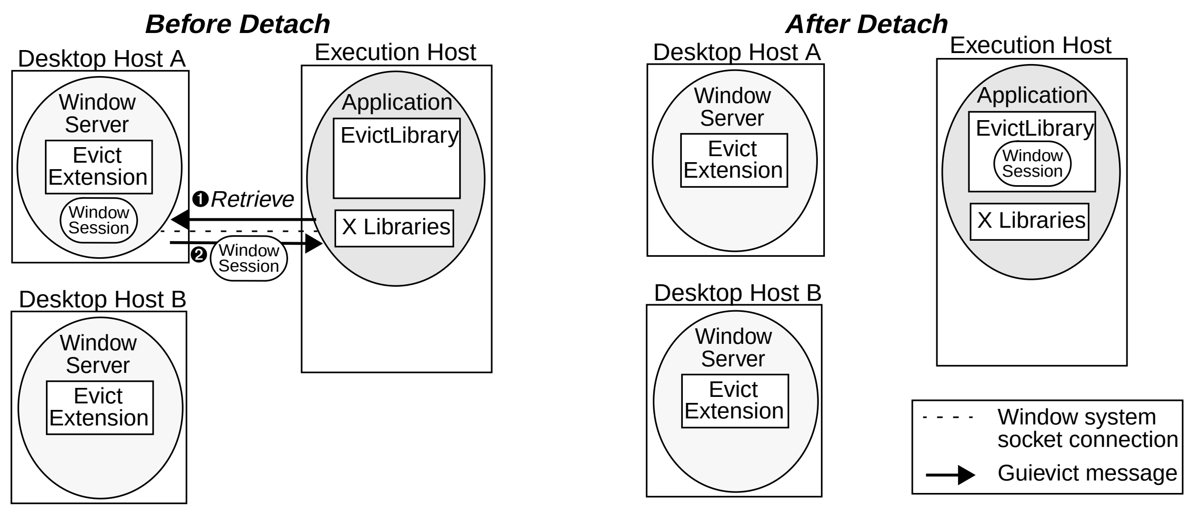

Figure 3: Detaching a GUI from Desktop A.
The evict client library requests the window session from the evict server extension at Desktop A, then closes connection.

- ❏ Retrieving the GUI resources of an application: We have developed an extension to the X window server that enables an application, at any time, to determine the identifiers of its GUI resources and the dependencies among these resources, and to extract these resources in a form from which they can be regenerated on a new desktop.
- ❏ Regenerating an application's resources in another desktop host: We use standard X protocol operations to regenerate GUI resources in the new window server.
- ❏ Ensuring GUI migration is transparent to the application: Applications whose GUIs have been migrated can be confused by the resulting changes to resource identifiers, message sequence numbers, and display characteristics such as pixel depth. We interpose a filter (called the *guimux*) on the communication between the application that provides the mapping necessary to maintain transparency. The guimux also serves as a multiplexor for GUI replication.

The main limitations of guievict are that (1) it requires the user to install our X window server extension on their desktop hosts, (2) it requires the availability of symbols for the window protocol stubs used by the application, and (3) it has a large (20 second) overhead in checkpointing font state. We discuss implications and possible workarounds of the last two issues in Section 3.

The remainder of this paper is organized as follows. Section 2 presents the architecture of guievict. Section 3 describes its implementation. Section 4 presents our evaluation of guievict. Section 5 identifies the security issues raised by guievict. Section 6 presents related work.

2 System Overview

The major steps in the operation of guievict are: initializing the system, migrating an application's GUI from one desktop host to another, replicating an application's GUI on multiple desktops, and migrating an application process along with its GUI. These operations involve four system components. The *evictor* is a program that the user runs to control the evict system. The user runs it on the same host as the application process. The *evict client library* is loaded by the evictor at run time into the application process and implements application-side GUI migration operations. The *evict server extension* is the corresponding server-side component. The *guimux* is a daemon, started by the client library, that runs on the execution host and ensures that GUI migration and replication is transparent to the application.

In the remainder of this section we describe the guievict operations in detail. In Section 3, we discuss our solutions to the technical problems underlying these operations.

2.1 Initialization

The user prepares a running application process for the evict system by invoking the evictor's *initialize* operation, passing the application process id as an argument. The evictor *hijacks* [25] the application process: it stops the process and forces it to load and initialize the evict client library (see Figure 2). During its initialization the client library establishes a communication channel for future interaction with the user. Hijacking is transparent

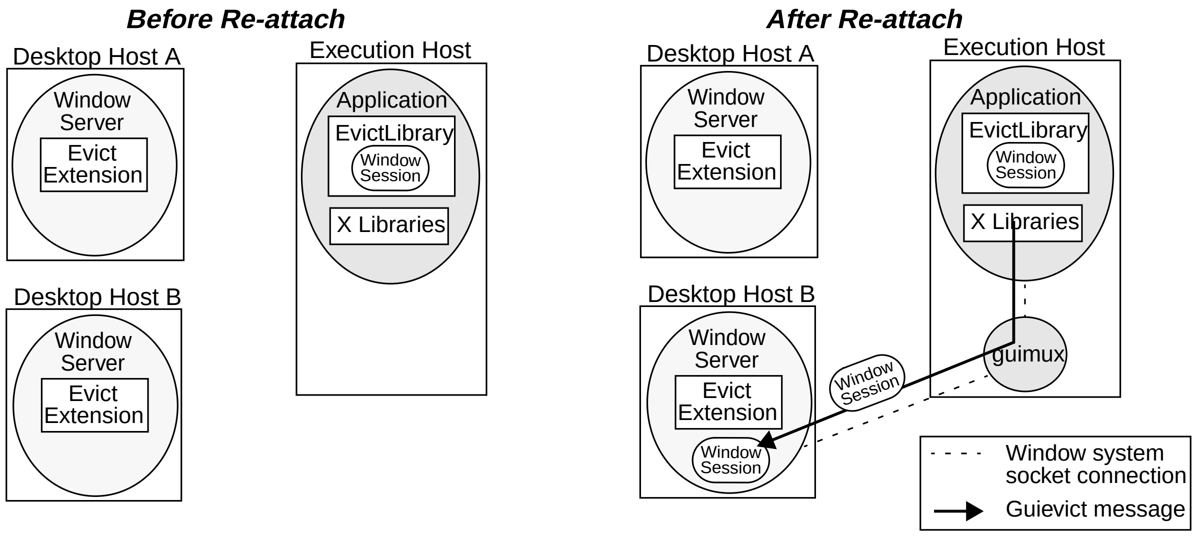

Figure 4: Re-attaching a GUI to Desktop B.
The evict client library establishes a new connection to Desktop B through the guimux, which forwards all window communication between the application and Desktop B, starting with the request to regenerate the window session.

to the application; afterward the evictor resumes the application process, allowing it to continue normally. At any later time the user can run the evictor again to request an evict operation. The evictor interrupts the application process and sends commands to the client library over the communication channel.

The evict server extension must be loaded and initialized in the window server before the user's first request for an evict operation. The XFree86 window server does not support run-time extension loading, so in our implementation the server must be configured to load the evict extension before it is started, but this is not necessary in general.

2.2 GUI Migration

GUI migration is broken down into two steps.

First, the user requests the application (with an evictor command) to *detach* its GUI from the window server. In response the evict client library (see Figure 3):

a. Synchronizes the application's communication with the window server and blocks the application from further communication;

b. Retrieves the window session from the server;

c. Closes the connection to the window server.

Second, the user requests the application to *re-attach* its GUI to a new window server (see Figure 4). The evict client library starts a guimux process, replaces the application's socket to the window server with a full-duplex pipe to the guimux process, and then transfers control to the guimux process. Then the guimux process:

a. Opens a connection to the new window server;

b. Regenerates the state of the window session;

c. Signals the evict client library to resume the application.

The re-attach operation may come an arbitrary period of time after the detach. In the meantime, the client library suspends the execution of application code to prevent the temporary absence of a window server connection from affecting the application. For the common case of a user who wishes to detach from one desktop and attach to another in one logical operation, the evictor provides a combined detach and attach command.

2.3 GUI Replication

GUI replication is a simple variation of GUI migration. The user requests (with the evictor) the application to *replicate* its GUI on another window server. The evict client library performs all but the final step of the detach operation. That is, it acquires the current state of the window session, but does not close the connection to the window server. It then performs a normal attach operation to connect the GUI to the additional window server (see Figure 5). If this is the first time an attach operation has been performed, it also redirects the original window server connection through the guimux process.

2.4 GUI+Process Migration

Guievict supports the simultaneous migration of the application process and its GUI. Figure 6 shows a typical scenario for this type of mobility in which the execution and desktop hosts are the same (such as a laptop) and the user wants to migrate their application process

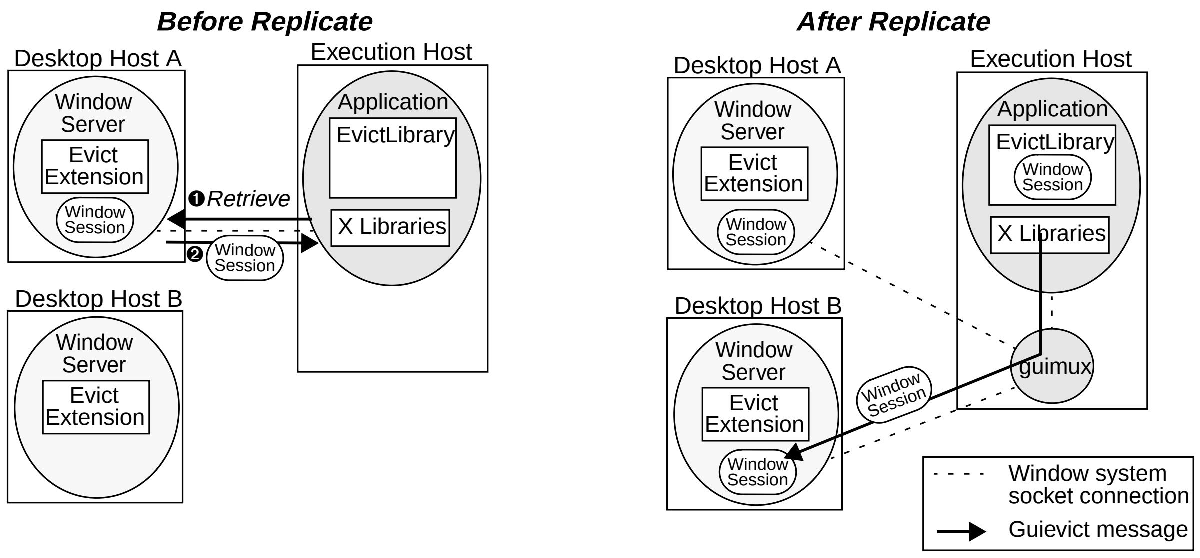

Figure 5: Replicating a GUI on Desktop B.
Replication is similar to migration, except that the connection to Desktop A is preserved and multiplexed by guimux with the connections to other desktop hosts.

and its GUI to another host (such as the computer on their desk). In this scenario, the user uses evictor to request the application to *migrate*, providing two arguments: the name of the new X window server and the name of the new execution host. Evict then:

a. Detaches the application's GUI from its window server;

b. Terminates the guimux daemon (if one is running);

c. Checkpoints the application process, producing a *checkpoint file* [15] containing the state of the application process, including its window session;

d. Exits the application process.

At this point the user must transport the checkpoint file to the new execution host and invoke the evictor to *restart* the application process. To complete the migration, evict:

e. Restores all the state of the application process except for its GUI;

f. Attaches the application process to the new window server.

Checkpointing and restarting the application process is transparently performed by a user-level process checkpoint library that is linked with the evict client library. The details of this library have been described previously [15,25] and are outside the scope of this paper.

3 Implementation

We have implemented evict on x86 Linux using the XFree86 implementation of the X window system. We describe the major technical issues and how we solve them in our implementation: hijacking the application, find the application's connection to the window server, synchronizing its connection, retrieving and restoring GUI resources, and ensuring that GUI migration is transparent to the application.

3.1 Hijacking the Application

Process hijacking can be safely implemented with basic dynamic instrumentation mechanisms, such as those provided by the Dyninst API [6]. These include stopping the process, forcing the process to execute code to load and initialize the evict client library, and then resuming the process. The evictor contains its own implementation of these mechanisms to avoid requiring users to install additional software like the Dyninst API (which contains much more functionality than evict requires).

The evictor forces the application process to load the evict client library by injecting code that calls the run-time library loading feature (usually named dlopen) of the process's dynamic loader. This technique does not directly work with statically-linked programs, since there is no dynamic loader in processes based on such programs. Supporting statically-linked programs is important because many GUI-based applications are distributed statically to avoid forcing users to have the necessary GUI library dependencies. We have developed hijacking functionality to cope with statically linked programs. The idea is to map and initialize a copy of the dynamic loader into the process's address space, essentially by reproducing the initial steps the

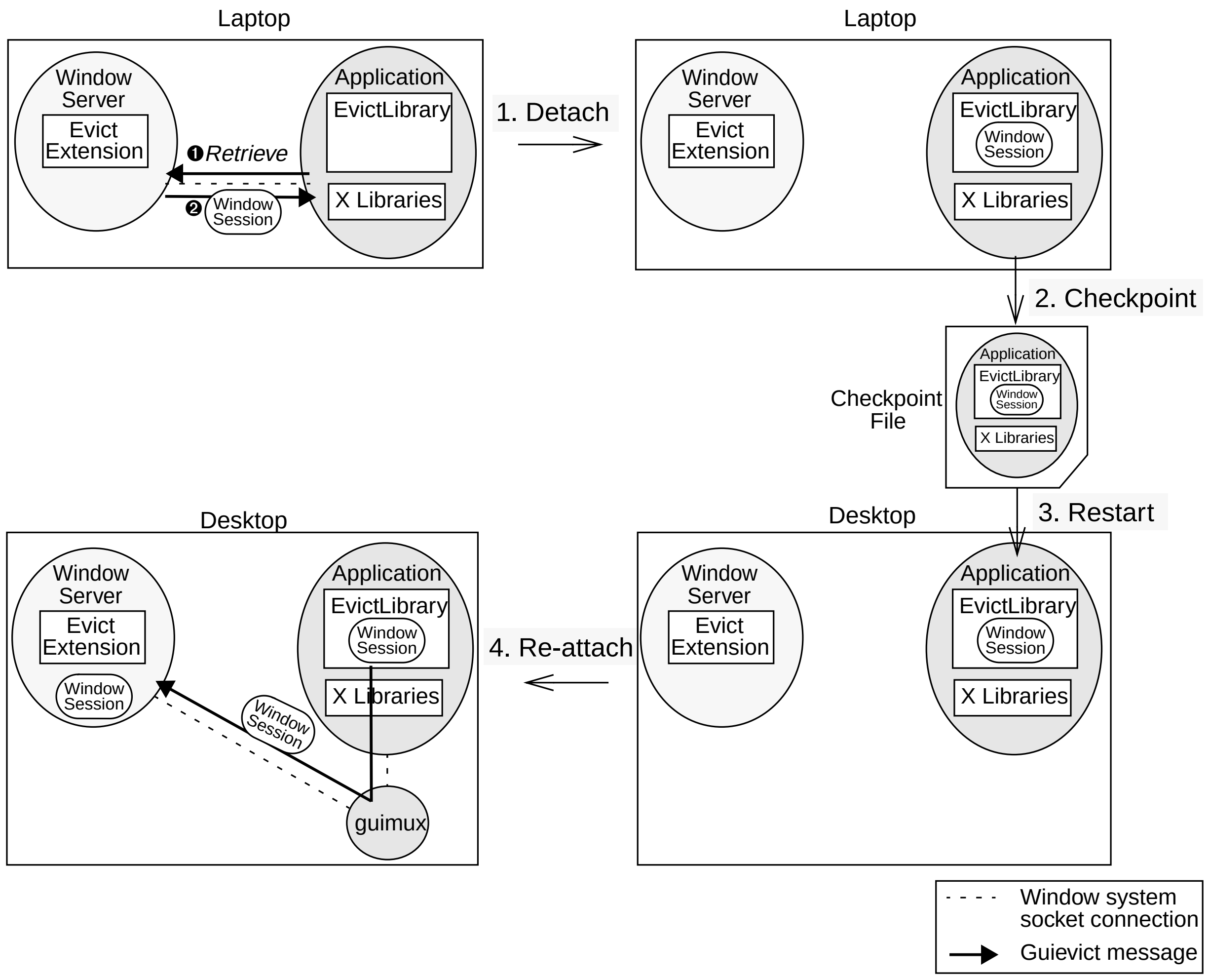

Figure 6: Migrating an application process and its GUI from a laptop to a desktop computer.
(1) The evict client library detaches the GUI from the window server; (2) Evict checkpoints the application process, producing a checkpoint file containing the entire application state including the window session; (3) After transporting the checkpoint file to the desktop host, evict restarts the application process; (4) Evict re-attaches the GUI to the desktop window server.

operating system takes when loading a dynamic-linked program. This instance of the dynamic loader does not control the original code in the process, but rather serves only to provide an implementation of dlopen that we can call as we do for dynamically-linked programs.

The evict client library creates a named Unix domain socket for subsequent communication with the evictor. The library deletes the socket when the application exits normally. The socket name is based on the application's process and user ids to avoid conflicts with other independently running instances of evict and to allow stale sockets left behind by abnormal termination to be cleaned up by the user who discovers them. The evictor gets the attention of the evict client library by writing a message the socket, which causes the application process to receive a signal that is handled by the library.

3.2 Finding the Window Server

X windows applications communicate with the window server over a Unix domain or TCP socket. Unlike proxy-based systems such as xmove, guievict may be invoked after the creation of the connection to the window server. It must search the file descriptors of the process for sockets connected to a window server. Most operating systems provide a way, such as a /proc entry on Linux, to list the open file descriptors of a process; on those that do not, guievict can test each possible file descriptor with the fstat system call. Guievict looks for a file descriptor that (1) refers to a socket inode (as reported by the fstat system call), and (2) is connected to a window server.

The second condition is difficult to check. In many cases, the getpeername system call, which returns the address of the socket on the other side of a connection, is sufficient: we check that the socket peer address is one of the well-known X window server TCP ports or Unix domain socket names. This test can fail if the application is tunnelled to the window server through a proxy, since the peer address of its socket will be the proxy's address. In many common proxy configuration, such as ssh tunnels [24] and firewall port forwarding rules [26], the difference between the proxy address and a normal window server address is a small positive offset in the port number, which is easy to recognize.

In the unlikely event that getpeername does not reveal an obvious window server connection, guievict checks whether the peer address of each socket leads to a window server. It creates a new socket, attempts to connect it to the peer address, and, if the connection succeeds, performs the first round of the standard X windows handshake. If the server gives the expected response to the handshake, guievict concludes that it has found a socket connected to an X server. If the probe fails on all sockets, guievict gives up control of the application.

The use of this probe raises two concerns. First, the probe may succeed on non X window servers that happen to respond to the handshake like a X windows server. In practice, the response is distinguishable from that seen in common protocols such as ssh, telnet, ftp, and http, however unfortunately it is not so distinctive to presume that conflicts will not occur in less common protocols. In the event of a false positive, guievict should eventually receive nonsensical messages from the server, after which it will abort. Second, the probe may have a negative effect on probed servers. Although server implementations should be robust to spurious connections, not all existing ones are, particularly those that have not been hardened for Internet deployment. For users who cannot risk using the probe, the evictor accepts a command line argument to identify the file descriptor or peer address of the window system connection.

3.3 Synchronizing Communication

Guievict must ensure that the state of the window session does not change while it is being retrieved. Changes to window state are caused by messages exchanged between the application and the window server. Guievict synchronizes the communication by finding a point in the message stream where there are no partially sent or unanswered requests, and then blocks further communication.

The synchronization occurs in two steps. First, guievict forces the application process to reach a message boundary in the stream of messages from the application to the server. It examines the application's process stack before starting an operation, searching for X library functions that are stubs for X protocol requests. If such a function appears on the stack, which indicates that the application process may be in the middle of sending a message, guievict sets a timer, resumes the application code for a short period of time, and then re-examines the stack. This procedure repeats until the stack is free of potentially unsafe functions.

Second, guievict sends an X protocol request containing an illegal resource identifier to the server. The only effect of this request is that it elicits an error message from the server. Guievict reads and scans the stream of messages from the server until it recognizes the error, at which point the client has no unanswered requests and the connection is synchronized. The messages read before the error are buffered and re-sent to the application from the guimux when the application is allowed to resume.

Detecting the presence of X protocol stubs on the application process stack depends on the presence of the symbols for these functions in the application process. This is not an issue for dynamically linked applications because the symbols must be present to facilitate linkage. However, symbols may be stripped from statically linked executables. The evictor refuses to work with stripped static applications. To remedy this limitation we are investigating alternative approaches to synchronizing the communication that do not depend on stack traces, including inferring message boundaries by analyzing messages as they are sent over the socket.

3.4 Retrieving and Regenerating GUI Resources

X windows applications create, modify, and destroy GUI resources through the exchange of X protocol messages with the X window server. GUI resources reside in the window server and are indirectly manipulated by the application by 32-bit *resource identifiers*, which are drawn from a namespace that is global across all clients of a window server. Clients choose the low-order bits of the identifier for each resource that they create, but they must set the high-order bits to a fixed *client id* chosen by the server when the application connects to it.

It is generally impossible to locate the resource identifiers in the application process's code and data, so we must get them from the X window server. The window server manages a per-client table of allocated resources but, unfortunately, it does not provide external access to the information in this table. Since no previously reported server extension has addressed this limitation, our server extension provides the missing interface. Using our *GetResources* request, an application can request the server to return an enumeration of all the resources that the application has created. For most types of resources, the resource identifier is sufficient for the application to retrieve the state of the corresponding resource with standard X protocol requests, but there are four exceptions: windows, graphics contexts, cursors, and fonts. The *GetResources* reply for windows includes the background pixel value and the window's cursor identifier. The replies for graphics contexts and cursors include their entire state: for a graphics context, a small array of flags, and for a cursor, its bitmap and geometry.

Fonts are more complicated. X windows fonts are stored at the server. Clients acquire the use of a font by sending a request to the server containing a font resource identifier and the name of the font. The server loads and binds the font to the identifier if it has the font, and otherwise returns an error. Applications can request detailed geometric information about the font associated with a font identifier, but unfortunately there is no request to map a font identifier to the name of the font. Strangely, the server discards the font name after loading a font, so the mapping is not possible even within our server extension. During the detach operation, the evict client library performs a search to map each font identifier to a font name. It requests the server to list of all of its stored fonts (a standard X protocol request), and then searches this list for a font whose geometry matches that of each font identifier. Usually, the font name suffices to regenerate the font resource on a new window server, but sometimes the new server does not have a font with that name. In those cases, guievict searches the font list on the new server and selects the font with the closest matching geometry, using a least-squares font matching algorithm similar to that used by HP Shared X [9]. This complicated and expensive approach to migrating font resources could be eliminated by switching to client-managed fonts, a recently proposed architectural change to the X window system [10]. In the meantime, we eliminate the overhead by caching in the application's file system the font names and geometry of each server we use regularly.

Sometimes it is not possible to regenerate pixel-based resources identically to their previous instances. Displays can vary by the number of bits per pixel (depth) and the method by which pixels are mapped to colors (visual type), both of which cause the meaning of pixel values to change. Xmove provided an additional translation operation that mapped pixel values from their previous depth and visual to that available on the current server. Recent developments for the X server, however, promise to eliminate the need for such translation. In particular, the R&R extension and shadow framebuffers [11] are server-based mechanisms for virtualizing depth and visual that have been designed with the goal of providing heterogeneity support for migration and replication systems. Our extension complements these developments; together they combine to produce a system for GUI migration that is transparent to display heterogeneity.

3.5 Maintaining Transparency

The main role of the guimux daemon is to make GUI migration transparent to the application process by translating resource identifiers and sequence numbers that appear in the messages exchanged between the application and the window server. The resource identifier mapping is initialized during the regeneration of GUI resources. The evict client library regenerates a resource by issuing an ordinary X protocol resource creation request containing the original identifier. As guimux forwards these requests, it replaces the identifier with an unused identifier that is valid in the current window server. For subsequent messages from the application to the server, the guimux maps references to resources to their current identifier; it performs a similar reverse mapping on messages from the server to the application. As the application destroys resources, they are removed from the map.

Sequence numbers occur in messages sent from the window server to the application and represent the number of messages the server has processed for the application; they do not occur in messages from the application. The guimux replaces the sequence number of a message with the next sequence number expected by the application process. This procedure is initialized when guievict synchronizes the communication to the window server. At the point, the next sequence number expected by the application process is the sequence number contained in the sentinel error reply.

Replication of windows on multiple displays extends the role of the guimux process. While managing a replicated GUI, the guimux maintains a separate translation map for each window server connection. Messages from the application are translated and sent to each window server. Messages from the window servers are reverse translated and forwarded in series to the application. To control the behavior of replicated GUIs, guimux accepts a set of commands, sent by the evictor, that act as primitives for setting replication policy. For example, the user can suppress the forwarding of keyboard, mouse, and window modification events from selected desktop hosts to allow users seated at those desktops to observe but not modify the state of the GUI; more sophisticated policies for managing collaborative work [2,12] can be built over these primitives.

4 Evaluation

We have evaluated the performance of guievict's GUI migration functionality. As a point of reference, we compared its performance to the proxy-based xmove system. We measured the time to detach and re-attach a GUI and the impact on interactive response. We performed our measurements on a 700 MHz Pentium-III laptop running XFree86 4.2.0 on Linux 2.4.18, and we used the most recent release of xmove [21]. Overall, the results are not surprising. Guievict takes somewhat longer than xmove to detach a GUI from a window server, but re-attaches in comparable time. Xmove and guievict (after re-attach) both increase the latency of the communication between the application process and the window server, but not enough to be perceptible to users.

4.1 Detach and Re-attach Latency

We measured the latency of detaching a GUI from a window server and then re-attaching it to the same window server for several applications. The guievict detach latency is the elapsed time from when the evict client library receives the detach command to just after it closes the connection to the window server. The guievict re-attach latency is the elapsed time from when the evict library receives the re-attach command to just after it allows the application process to continue. The xmove latencies are analogous. We ran both the application process and the window server on the laptop and we detached the application's GUI after its initial windows were drawn but before any user interaction with the GUI. We report average measurements over five runs. Our results are reported in Table 1.

Guievict has a more expensive detach operation than xmove. Table 2 breaks down the average guimux detach time for one application. The most expensive stage is mapping the font identifiers to fonts names, during which most of the time is spent waiting for the server to return the complete list of the fonts. More generally, guievict takes longer to detach because it retrieves the GUI state when it receives the detach request, while xmove collects the GUI state as it is created. To reduce our detach latency, we plan to enable the evict client library to incrementally fetch the font list during idle periods of the application process's execution, but we have not implemented this optimization yet. Guievict and xmove have similar re-attach performance, which is to be expected because they perform similar tasks during this stage.

Application	Guievict Latency (msec)		Xmove Latency (msec)	
	Detach	Re-attach	Detach	Re-attach
Xterm	21,042	46	32	134
Xmame	21,100	55	857	96
Emacs	21,198	148	45	230
Ghostview	21,655	379	315	307
Netscape	21,655	667	362	432

Table 1: Average detach and re-attach latency.

Stage	Time (usec)
Font List	21,052,039
Pixmaps	562,205
Windows	31,824
Fonts	6,986
Graphics Contexts	1,977
Cursors	113
Colormaps	63

Table 2: Breakdown of detach latency for Netscape.

4.2 Interactive Overhead

To measure the impact of guimux and xmove on interactive response we created a small application that repeatedly sends a request of minimal size (8 bytes) to the window server and waits for a reply of minimal size (32 bytes). We measured the average time for 1000 round trips for the application by itself, after it has been detached and re-attached with guievict, and through xmove. Our results are reported in Table 3.

Guievict and xmove have a measurable impact on the round trip time, caused by the overhead of redirecting the window system communication through a proxy. Since the overhead is less than a millisecond, it should not ever be perceptible to users.

System	Latency (usec)
None	70
Guievict	107
Xmove	133

Table 3: Average round trip time for a minimal X protocol request and reply.

5 Security

Our system introduces three issues related to the security of X window applications and servers.

First, the owner of the application process must be able to control who is able to migrate or replicate its GUI. Our policy is to allow only the owner of a process to perform guievict operations on the process. This policy is enforced by two mechanisms. First, because the standard operating system protection prevents one user from modifying a process of another user, a process can only be hijacked by its owner. Second, the evict library authenticates the messages it receives from the evictor. It uses the credential passing mechanism of Unix domain sockets to ensure that the sender of a message is the same user who owns the application process. These mechanisms suffice to protect an application process from ordinary users, but not, of course, from the superuser of the execution host.

Second, the guievict server extension should not weaken the security of the X window system, a goal we believe we have met. Since the GetResources request only returns information about the resources of the application that issues the request, it cannot be used to learn about the resources of another application. Although a man-in-the-middle attack could be staged to inject a GetResource request in another application's connection to the window system, the information that would be revealed could also be obtained from passive eavesdropping on the connection, an old X windows vulnerability [8]. The defense, then and now, is to encrypt the connection.

Third, the owner of a desktop host must authorize guievict to re-attach a GUI to their display. Most X server access control mechanisms (such as MIT-MAGIC-COOKIE) require an authorized application to possess a server-generated capability that it can present to the server when it establishes a connection. This capability gives its possessor complete access to the X server. The desktop owner must have a secure way to transfer the capability to the guievict user, and they must trust the guievict user not to abuse the access to the server. Guievict does not provide capability transfer mechanisms and it does not change the access control policies of the X server. These issues are trivial in migration scenarios, where the desktop user and the guievict user are usually the same, since the user can transfer the capability when they log in to the execution host to run the evictor. However, these issues must be faced by a GUI replication system built over guievict's replication mechanism.

6 Related Work

Guievict most closely resembles xmove [22] in that both systems share the goal of migrating GUIs on a per-application basis from one desktop host to another. Unlike guievict, xmove requires the user to redirect in advance their application's window system connection through a proxy that tracks the state of its resources. In addition, xmove does not support the migration of application *processes*: it lacks a way to restore the communication between the application and proxy processes after the application process has migrated, as well as a way to migrate the proxy process or its state.

Other systems, including VNC [18], Teleporting [17,23], and Slim [20], provide remote access to a *session*, a collection of GUIs for remotely executing applications. Unlike xmove, these systems enable the user to migrate the display of all applications in the session as a single unit, a convenience for users who want remote access to their entire desktop, but a hindrance for users who want independent movement of their GUIs. Like xmove, these systems depend on a level of indirection that must be established when applications are started, and they do not support application process migration.

Recent systems have extended the session migration concept to include application process migration. Users of the Internet Suspend/Resume system [14] run their entire computing environment, from the operating system to their applications, inside a virtual machine whose state can be saved and regenerated on another machine. This system's mobility model is much coarser than evict's: all applications (and their operating system) migrate as a single unit and GUI migration cannot be done separately from application process migration. The Zap system [16] provides a finer degree of mobility by allowing users to run their application processes in session abstractions that can be independently migrated to new hosts. However, Zap provides no support for GUI migration; its users must use systems like VNC to migrate their GUIs, adding another level of indirection.

Many systems have been developed to replicate the GUI of unmodified X windows applications [1,2,4,5,3,13] on multiple desktops. Like xmove, most of these systems require applications to be redirected to a proxy when they are started. One interesting exception is the HP Shared X [9] system, which performs replication through the use of an X server extension. Unlike the guievict extension, instead of providing a way to retrieve window session state from the server, the extended Shared X server itself acts a proxy that regenerates the GUI on the new displays and forwards messages between the application process and the new displays. The extension is thus unsuitable for GUI migration because it does not allow the application to detach from its original X server.

7 Conclusion

Guievict enables the GUI of an ordinary X windows application to be migrated to another desktop host or replicated on multiple desktop hosts without premeditative steps such as redirecting the application process's communication through a proxy or relinking the application program binary. We have shown that server functionality necessary to retrieve a window session, a transportable representation of an application's GUI, is small and can be encapsulated in a window server extension without server recompilation, and that ordinary X windows applications can be hijacked at run time to retrieve their window session and perform GUI migration or replication.

We have implemented guievict for x86-based versions of Linux running the XFree86 window system. The code is freely available at http://www.cs.wisc.edu/~zandy/guievict.

References

[1] H.M. Abdel-Wahab and M.A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration. *IEEE TriCom '91: Communications for Distributed Applications and Systems*. Chapel Hill, NC, USA, April 1991, pp. 159-167.

[2] H. Abdel-Wahab and K. Jeffay. Issues, Problems and Solutions in Sharing X Clients on Multiple Displays. *Internetworking – Research and Practice* **5**, 1, March 1994, pp. 1-15.

[3] J.E. Baldeschwieler, T. Gutekunst, B. Plattner. A Survey of X Protocol Multiplexors. *ACM SIGCOMM Computer Communication Review* **23**, 2, April 1993.

[4] J. Bazik. XMX – An X Protocol Multiplexor. http://www.cs.brown.edu/software/xmx.

[5] C. Bormann and G. Hoffmann. Xmc and Xy – Scalable Window Sharing and Mobility. *8th Annual X Technical Conference*, January 1994.

[6] B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications* **14**, 4 , Winter 2000, pp. 317-329.

[7] P. Chen and B. Noble. When Virtual is Better Than Real. *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, May 2001.

[8] S. Garfinkel and G. Spafford. **Practical UNIX & Internet Security**, 2nd Edition. O'Reilly and Associates, Sebastopol, CA, April 1996.

[9] D. Garfinkel, B.C. Welti, T.W. Yip. HP SharedX: A Tool for Real-Time Collaboration. *Hewlett-Packard Journal* **45**, 2, April 1994, pp. 23-36.

[10] J. Gettys. The Future is Coming: Where the X Window System Should Go. *2002 Usenix Annual Techical Conference (Freenix Track)*, Monterey, CA, June 2002, pp. 63-69.

[11] J. Gettys and K. Packard. The X Resize and Rotate Extension —RandR. *2001 Usenix Annual Technical Conference (Freenix Track)*, Boston, MA, June 2001.

[12] T. Gutekunst, D. Bauer, G. Caronni, Hasan, and B. Plattner. A Distributed and Policy-Free General-Purpose Shared Window System. *IEEE/ACM Transactions on Networking* **3**, 1, Februrary 1995.

[13] O. Jones. Multidisplay Software in X: A Survey of Architectures. *The X Resource*, Issue 6, O'Reilly & Associates, Jan 1993, pp. 97-113.

[14] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002)*, Callicoon, NY, June 2002, pp. 40-46.

[15] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report #1346, Computer Sciences Department, University of Wisconsin, April 1997.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.

[17] T. Richardson, F. Bennett, G. Mapp, and A. Hopper. Teleporting in an X Window System Environment. *The X Resource*, Issue 13, O'Reilly & Associates, Jan 1995, pp. 133-140.

[18] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A. Hopper. Virtual Network Computing. *IEEE Internet Computing* **2**, 1, January/February 1998, pp. 33-38.

[19] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics* **5**, 2, April 1986, pp. 79-109.

[20] B.K. Schmidt, M.S. Lam, J.D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-client Architecture. *17th ACM Symposium on Operating Systems Principles (SOSP '99)*. Kiawah Island, South Carolina, December 1999.

[21] E. Solomita. Xmove Version 2.0 Beta 2. ftp://ftp.cs.columbia.edu/pub/xmove, November, 1997.

[22] E. Solomita, J. Kempf and D. Duchamp. Xmove: A Pseudoserver for X Window Movement. *The X Resource*, Issue 11, July 1994, pp. 143-170.

[23] K. Wood, T. Richardson, F. Bennett, A. Harter, and A. Hopper. Global Teleoporting with Java: Towards Ubiquitous Personalized Computing. *Nomadics '96*, San Jose, March 1996.

[24] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, and S. Lehtinen. SSH Protocol Architecture. *Internet Engineering Task Force Internet-Draft* draft-ietf-secsh-architecture-13, September 2002.

[25] V.C. Zandy, B.P. Miller, and M. Livny. Process Hijacking. *Eighth International Symposium on High Performance Distributed Computing (HPDC '99)*, Redondo Beach, CA, August 1999, pp. 177-184.

[26] E.D. Zwicky, S. Cooper, and D.B. Chapman. **Building Internet Firewalls**, 2nd Edition. O'Reilly and Associates, Sebastopol, CA, June 2000.