Visual Proxy: Exploiting OS Customizations without Application Source Code

M. Satyanarayanan, Jason Flinn, Kevin R. Walker
School of Computer Science
Carnegie Mellon University

1. Introduction

Performance and functionality enhancements are often made available to applications through extensions to an operating system's API. Recent examples include API extensions for search latency reduction [13], application-specific conflict resolution [5], mobility [4, 9], informed prefetching [10], consistency management [14], and low-overhead transactions [7]. Unfortunately, source code to some of the most popular and important applications is often not available under acceptable licensing terms. How then can such applications benefit from the new performance or functionality enhancements?

In this paper, we describe a solution to this problem that appears promising for a broad class of interactive applications that rely on graphical user interfaces (GUIs). We call our solution a *visual proxy*: "proxy" because it involves redirection through an interposing layer of code; and "visual" because this proxy is located at the front end of a GUI-based application and modifies its visual appearance.

The concept of a proxy is a well-established idea in Web browsers [8]. Visual proxies differ from Web proxies in two important ways. First, Web proxies have historically been used to extend the back-end of browsers. In contrast, visual proxies augment the front-end of an application and enable extensions that may not be feasible at the back-end. Second, Web proxies are easy to implement because browsers already provide for their existence — interposing a Web proxy merely involves specifying its IP address to a browser. In contrast, visual proxies are more difficult to implement because current GUI-based applications are not written with them in mind. Another approach that bears a modicum of resemblance to the visual proxy idea is the use of an interposition toolkit to extend an operating system interface [3]. However, as in the case of Web proxies, interposition agents based on such a toolkit only extend the back-end of applications.

We describe our technique in more detail in the next section. Although we are still early in our exploration of the visual proxy approach, we have evidence of its viability for two quite different applications. Section 3 describes our use of the technique in Netscape [2], and Section 4 describes its use in FrameMaker [1]. We conclude in Section 5 with a discussion of future work.

2. Implementing Visual Proxies

Two key insights provide the foundation for the visual proxy idea. First, applications such as Web browsers, word processors, and spreadsheets reflect much of their internal state in the GUI. Second, such applications modify GUI state only by communicating with their windowing system through a narrow, well-defined channel. A proxy interposed in this channel can snoop on changes to the GUI and mirror the application's internal state. This mirrored state allows the proxy to correctly determine when and how to invoke system API extensions on behalf of the application. Of course, it is neither possible nor necessary to precisely mirror the entire state of an application in its proxy. Only those components of the state relevant to the API extensions being exploited need to be mirrored.

Extending an application requires the proxy to address several complex issues. First, the proxy must determine how to monitor changes to the user interface without significantly degrading application performance. Second, the proxy must modify the behavior of the application to ensure that it benefits from the API extensions. Finally, the proxy must augment the GUI to reflect information from both the application and the extended API.

Figure 1 shows the key components and interactions in the implementation of a visual proxy. Our initial

This research was supported by the Air Force Materiel Command (AFMC) and DARPA under contract number F19628-96-C-0061. Additional support was provided by the Intel Corporation and the Novell Corporation. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, Intel, Novell, Carnegie Mellon or the U.S. Government.

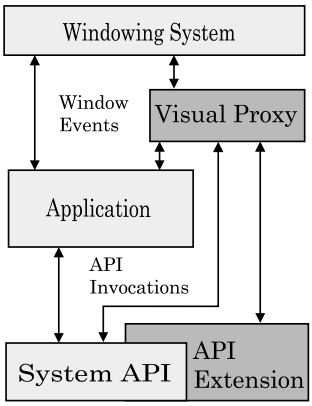


Figure 1: Visual Proxy Architecture

realization of this concept is in the context of X Windows [11]. Applications communicate with the X sever through a socket-based channel, using a well-defined windowing protocol. Redirecting the entire traffic stream through a proxy, though simple, would have a significant detrimental effect on performance. Therefore, the proxy takes a minimalist approach, monitoring only key messages that reveal important information about application state. The vast majority of events are sent directly to the application without the intervention of the proxy.

By monitoring these events, the proxy can correctly invoke system API extensions on behalf of the application. To ensure that the application benefits from these invocations, the proxy often needs to alter the behavior of the application. This is made possible by the proxy's position in the channel between the application and the windowing system. Since the proxy can generate synthetic window events simulating user input, it has access to all functionality available through the GUI and can thus stimulate the desired internal state changes in the application. By manipulating both visual appearance and internal application state, a visual proxy provides a powerful technique for augmenting application behavior.

Often, enhancing an application to make use of an extended system API increases the range of interactions and information available to the end-user. However, the application's existing user interface, designed without knowledge of the new system extensions, is typically insufficient to express such capabilities. Therefore, the GUI needs to be extended. A visual proxy first creates a separate GUI with widgets reflecting the additional capabilities provided by the API extension. The proxy then seamlessly integrates this interface with the application using a technique called *reparenting.* In this well-known window management technique, the application's original interface is inserted as a child window inside the extended interface, in much the same way that a window manager adds decorations such as title bars to an application window. The visual effect is that of combining the application's interface with one reflecting the new capabilities provided by the system API extension.

3. Dynamic Sets in Netscape

The dynamic sets API [12] extends the interface of a distributed file system by allowing applications to create unordered collections of the objects they intend to access. A subset of this API is shown in Figure 2. As an application iterates through the members of a dynamic set, the system takes advantage of non-determinism and asynchrony to reduce aggregate I/O latency. Use of dynamic sets in a modified version of NCSA Mosaic 2.6, a source-available browser, has been shown to reduce the latency of Web searches by up to 90% [13].

`setOpen`	creates a new set
`setClose`	terminates use of a set
`setIterate`	non-deterministically yields a new set member
`setDigest`	returns a summary of the set

The above calls represent the most important elements of the dynamic sets API. Calls that are infrequently used are omitted for brevity.

Figure 2: Dynamic Sets API

We were curious to see if the benefits of dynamic sets could be made available to Netscape Navigator, a widely-used proprietary browser for which source code was not available[1]. Integrating dynamic sets into Netscape was the original motivation for the development of the visual proxy concept.

Figure 3 shows how a visual proxy extends the Netscape GUI. The proxy inserts an additional menu-bar above the existing Netscape menu-bar, granting the user the ability to directly invoke dynamic sets actions. The proxy also displays a set dialog box representing

[1]Although source code to Netscape is now publicly available, we did not have access to it when we did the work described here.

(a) Original GUI

(b) Extended GUI

This figure illustrates how we have extended Netscape with a visual proxy to exploit dynamic sets. Part (a) shows the result of a normal Web search using the AltaVista search engine. Part (b) shows the corresponding result when the output of the search is treated as a dynamic set. Three components of part (b) are generated by a visual proxy: the menu-bar with buttons labelled "Set", "Actions", etc.; the dialog box labelled "Open Set Window"; and the message at the bottom saying "The iterator has been reinitialized".

Figure 3: Extending Netscape to Use Dynamic Sets

the dynamic set that contains the pages returned by a search engine. Finally, the proxy uses the window in the bottom left corner of the Netscape GUI to display transient information about the set. Because Netscape displays transient information such as the status of loading the current page in this window, the proxy's use of the window for similar tasks helps to transparently blend the GUIs of Netscape and the dynamic sets extension.

Although the menu-bar provides a convenient way to access the dynamic sets API, the utility of a browser interface lies in hypertext-based point-and-click navigation. Therefore, we also extended Netscape's interface to include the ability to create new sets by clicking on a hypertext link. For instance, a mouse click on a hypertext link with the control key pressed opens a dynamic set whose members are the sub-pages referenced by the specified link. The I/O latency of iterating through this set can be much smaller than if each page were accessed through the normal hypertext interface.

The following example illustrates how one might use the Netscape visual proxy. When started by the user, the proxy locates all Netscape windows currently being displayed and extends them by adding the dynamic sets menu bar shown in Figure 3. To provide a smooth transition, the proxy reparents the Netscape window into one with identical size and location. When the user moves or resizes the parent window, the proxy ensures that the Netscape window is appropriately reconfigured.

Subsequently, the user searches for the latest baseball news by entering parameters for a favorite search engine and clicking on a link while pressing the control key. When the proxy intercepts the window event corresponding to the click, it deduces from the control key state that the event is sets-related. The proxy passes the event through to Netscape and monitors the result. Since the event causes Netscape to load a new page, the proxy calls `setOpen()` to create the dynamic set which will contain the results of the search. The proxy also creates a set dialog box which allows the user to navigate through the set. The user iterates through the set by clicking the Iterate button in the set dialog box. On each click, the proxy calls `setIterate()` which returns the name of a page that has been prefetched by the system. The proxy causes the browser to load this page using the Netscape remote-control interface; alternatively, it could cause the same action by sending window events to Netscape. When he has finished, the user clicks the Close button; this causes the proxy to invoke `setClose()`.

4. Application-Specific Resolution in FrameMaker

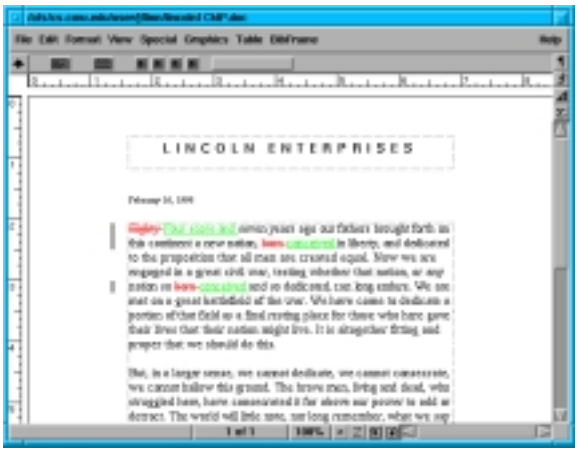
An *application-specific resolver (ASR)* is an important tool for handling update conflicts in an optimistically-replicated file system such as Coda [5, 6]. When connectivity is re-established after a network failure, the replicas of a file that was updated in multiple network partitions must be resolved to create a single resultant version. By exploiting application semantics unavailable at the level of the file system API, ASRs can substantially simplify the resolution of many update conflicts.

We are currently creating an interactive Coda ASR for FrameMaker, a commercial word processor for which we do not have source code. This ASR is implemented as a visual proxy, as shown in Figure 4.

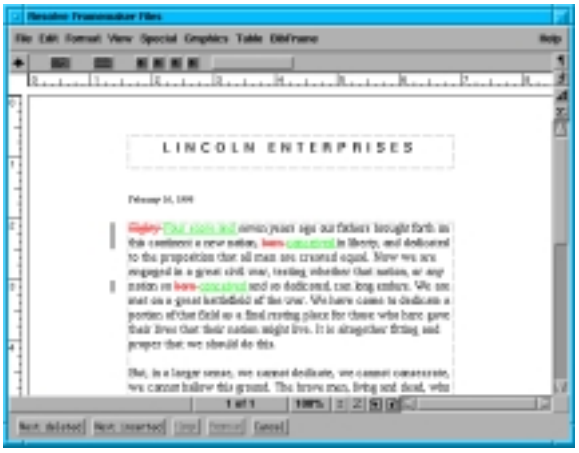
When a Coda client detects divergent replicas of a FrameMaker document, it invokes the ASR shown in Figure 4. This ASR consists of two components. The first is FrameMaker, which contains the application-specific knowledge necessary to detect and display conflicting updates within the document. The second component is a visual proxy, which extends the FrameMaker GUI to allow the user to merge updates and which invokes the Coda API to perform resolution.

Since the ASR is an extension of FrameMaker, the user has access to full editing capabilities. The extensions provided by the visual proxy are reflected in new buttons in the FrameMaker GUI, as shown at the bottom of Figure 4 (b). These buttons allow the user to search for conflicts, accept modifications, and delete modifications. In addition, the user may also perform the reconciliation via a point-and-click interface. In a manner similar to the Netscape proxy, the FrameMaker proxy will intercept mouse events and either accept or delete modifications if the user clicks on them in a specific manner. For instance, if the user clicks on a modification with the right mouse button, the proxy will remove the modification from the document.

Tasks such as deleting text and searching for modifications require the proxy to interact with FrameMaker. Since these functions are available through the word processor's GUI, the proxy sends artificial window events to FrameMaker to invoke the desired behavior. When it sends such events, the proxy sometimes manipulates the FrameMaker GUI to present a more pleasing interface to the user. For example, the proxy often prevents the display of pop-up windows which might otherwise distract the user. In these cases, detailed knowledge of the application state allows the proxy to achieve a more seamless interface.



(a) Original GUI



(b) Extended GUI

This figure illustrates how we have extended FrameMaker with a visual proxy to provide an interactive ASR. As part (a) shows, FrameMaker possesses the capability to highlight the differences between two versions of a document. Part (b) shows how this capability can extended through a visual proxy to allow interactive resolution of conflicts. The component of part (b) generated by a visual proxy is the row of buttons at the bottom labelled "Next deleted", "Next inserted", "Keep", "Remove" and "Cancel".

Figure 4: Extending FrameMaker

5. Conclusion

The growing number of applications for which source code is unavailable makes their integration with operating system extensions an important challenge. While much work lies ahead, our initial results give us

confidence that visual proxies have a major role to play in addressing this challenge.

In the short term, we plan to integrate our implementation of the FrameMaker visual proxy with the Coda File System. We also plan to explore the creation of visual proxies for a broader range of applications, and to carefully evaluate their performance overhead. A major task that awaits us is the demonstration of the visual proxy concept in Windows NT and Windows 95. In the long term, we would like to evolve design guidelines for the developers of applications and windowing systems to better support visual proxies.

References

[1] *Learning FrameMaker*
Frame Technology Corporation, 1990.
Part Number 41-00522-00.

[2] James, P.
Official Netscape Navigator 3.0 Book.
Netscape Press, 1996.

[3] Jones, M.B.
Interposition Agents: Transparently Interposing User Code at the System Interface.
In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Asheville, NC, December, 1993.

[4] Joseph, A.D., deLespinasse, A.F., Tauber, J.A, Gifford, D.K., Kaashoek, M.F.
Rover: A Toolkit for Mobile Information Access.
In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, December, 1995.

[5] Kumar, P. and Satyanarayanan, M.
Supporting Application-Specific Resolution in an Optimistically Replicated File System.
In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems*. Napa, CA, October, 1993.

[6] Kumar, P.
Mitigating the Effects of Optimistic Replication in a Distributed File System.
PhD thesis, School of Computer Science, Carnegie Mellon University, 1994.

[7] Lowell, D.E., Chen, P.M.
Free Transactions With Rio Vista.
In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. St. Malo, France, October, 1997.

[8] Luotonen, A., Altis, K.
World-Wide Web Proxies.
Computer Networks and ISDN Systems 27, September, 1994.

[9] Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R.
Agile, Application-Aware Adaptation for Mobility.
In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. St. Malo, France, October, 1997.

[10] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J.
Informed Prefetching and Caching.
In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, December, 1995.

[11] Quercia, V., O'Reilly, T.
X Window System User's Guide.
O'Reilly & Associates, Inc., 1996.

[12] Steere, D.C.
Using Dynamic Sets to Reduce the Aggregate Latency of Data Access.
PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[13] Steere, D.C.
Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency.
In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. St. Malo, France, October, 1997.

[14] Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.
Managing Update Conflicts in a Weakly Connected Replicated Storage System.
In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, December, 1995.