1 Introduction to proxy servers

A *proxy server* is a process that acts as an intermediary between a client (web browser) and (web) server. The client sends requests to the proxy instead of the server; the proxy then forwards those requests to the server. The server responds to the proxy with a reply; the proxy forwards the reply back to the client.

In its simplest form, a proxy merely forwards messages back and forth. However, there is no requirement that the proxy forward a request to a server. Some proxies implement *caching*: if the requested data is in the cache, the proxy can reply with that data instead of forwarding the request to the server and waiting for a reply. A proxy can also handle some requests internally. For example, IBM's WBI proxy handles special URLs such as "`http://_wbi/setup`" that are meant for the proxy, not a web server. A proxy could also refuse to forward requests to certain sites, [1] such as `doubleclick.net`.

A proxy can also transform data in a reply from a server. (Figure **??** shows this configuration.) For example, the server may return a web page that contains advertisements, and the proxy could remove them. Another manipulation of web pages is to translate text into a different language. For example, there is a proxy named EtoJ that translates English into Japanese.[2] A proxy could also refuse to forward certain *types* of data (such as music files or animated GIFs), or data with particular *content* (such as obscenities or HTML frames) back to the client. Proxies are also used to implement firewalls, which monitor network traffic between a set of "trusted" computers and all other ("untrusted") computers. Firewalls monitor all messages and sometimes refuse certain requests.

2 Proxy to modify Java classes

I am using a proxy to modify Java class files (in Java bytecode format) sent by a web server, in an attempt to put restrictions on applets. The basic approach is to substitute references to one class with references to a more restrictive class, as shown in figure **??**. A class such as `Window` can be

[1] Netscape and CyberPatrol both sell such a proxy that rejects requests to certain sites, to allow parents to control sites that their children visit. CyberPatrol also has a proxy that blocks access to "sports and leisure" sites, to allow employers to control what their employees visit.

[2] EtoJ information is available at `http://zip-translator.pharm.kyoto-u.ac.jp:10081/What-is-EtoJ-E.html`

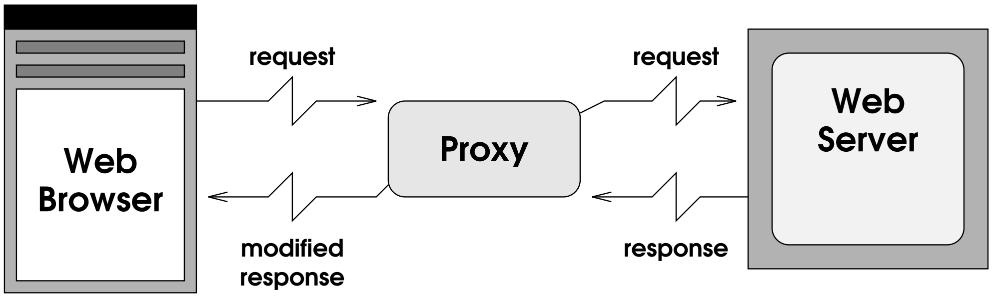

Figure 1: A proxy sits between the client and server

replaced with a subclass of `Window` (which will be called `Safe$Window`[3] in this example) that performs more security and sanity checks. For example, `Safe$Window's` `setSize()` method can limit the size of a window to half of the screen size. Since `Safe$Window` is a subclass of `Window`, the applet should not notice the change, unless it attempts to create a very large window. The proxy server can restrict applets without requiring any changes in either the web server or the Java Virtual Machine in the web browser.

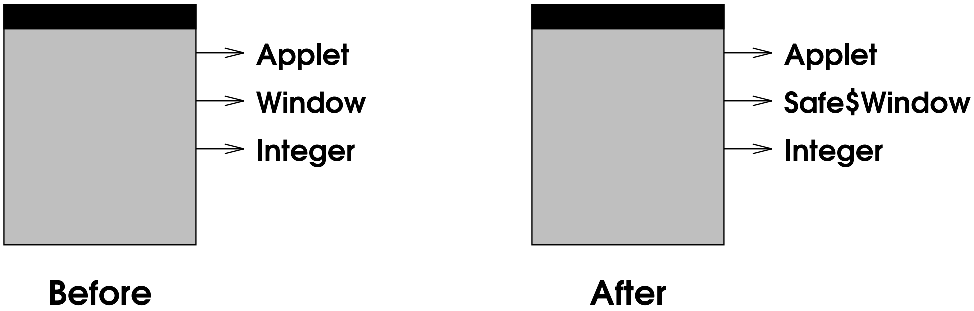

Figure 2: The proxy can modify a Java applet

Some of the things that might be restricted are:

- Window size
- Window position
- Number of windows
- High thread priorities

[3]Note that since `Safe$Window` is not part of the original set of classes used by the applet, it is not available on the web server, and instead must be delivered by the proxy.

- Threads that continue to run too long
- Access to network resources

3 Problems with this approach

This section contains technical details about the proxy project. There are a number of things that may be difficult to handle with this approach. An alternative approach would be to write a new Java `ClassLoader`, but at present, this is not possible in most (perhaps all) web browsers.

1. Java class files have no standard MIME type. Web servers commonly use `text/plain` or `application/octet-stream` for class files. This makes detection of Java class files more difficult. My proxy server assumes that a URL ending in `.class` is a Java class file, but it is not clear that this is guaranteed to be the case. Another possible test would be to scan the beginning of each file sent by the server to see if it contains the "magic numbers" present at the beginning of each class file.

2. Class files might be contained inside a ZIP or JAR container file. It is possible but impractical to examine every ZIP file to see if it contains class files. Furthermore, it would not be correct to assume that a ZIP file containing a class file is being used in an applet; it may be a file transfer (that happens to contain Java classes) instead.

3. Although unlikely, it is possible that someone is using the name `Safe$Window` (or other safe-class name) already. Name classes could be avoided by scanning the applet to see which names are already in use, and then picking a new name, but it is possible that the new name may later be used by the applet.

4. Performing a replacement of class references involves changing constant strings in the class file. It is possible that someone is using the string "Window" in a context other than as a class name; it would be incorrect to replace such an instance of the string.

5. Java includes a class named `Class` that gives programmers the ability to create instances of a class by name. It also allows programmers to query the class of an object. To prevent knowledge of our protected classes (such as `Safe$Window`, and also to prevent a program from

creating an instance of the unsafe window class, access to `Class` must be intercepted and replaced by `Safe$Class`.

6. Java bytecode includes the possibility of creating a string at run-time and then creating an instance of the class named by that string. Performing a string replacement during load-time would not catch these cases. An alternative to replacing strings at load-time is to replace the bytecode equivalent of `new x` with the bytecode equivalent of `new safe$translate(x)`: before each instantiation, the class name is replaced (at run-time) by its safe version.