Browsers for distributed systems: Universal paradigm or siren's song?

Robert C. Seacord and Scott A. Hissam

Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA
E-mail: {rcs;shissam}@sei.cmu.edu

Web-based browsers are quickly becoming ubiquitous in the workplace. Software development managers are quick to incorporate browsers into a broad range of software development projects, often inappropriately. The purpose of this paper is to examine the technical issues relevant to incorporating browsers as a component of a commercial off-the-shelf (COTS)-based solution. Issues examined include portability, performance, functionality, security, human factors, distribution, installation, upgrading, component-based development, runtime configuration management, and licensing.

1. Introduction

Not since the advent of software development toolkits has a technology captured the imagination of the software development community like browsers. Browsers are being used to provide the interfaces for an increasingly broad range of applications, from Internet sites intended largely for entertainment to large government systems. For example, one large government program is using a browser-based interface to access a database of technical drawings. While browsers have done a great deal to encourage the development of distributed systems, we must ask, at what cost? Has the rapid expansion of Internet browsers into all manner of distributed applications gone beyond the bounds of reasonableness?

An often-applied rationale for the use of browser-based interfaces in the development of distributed systems is that end users are familiar with their interfaces and have grown accustomed to using them. Other misconceptions about browsers that we hope to correct include the following:

- Browser applications are inherently "cross-everything" applications.
- Browsers simplify the development of any distributed system.
- Browser-based systems are always easier to install and upgrade.
- Browsers have robust and well-considered security models.
- Poor performance in browsers is a result of network latency.

In some cases a browser-based design may be optimal – for example, in the development of a distributed system that is principally hypertext. The problem is that the ubiquity and popularity of browsers is dangerously misleading and leads to a thoughtless, high-risk adoption of browser technology in inappropriate settings. By identifying both the benefits and risks of browser-based designs we hope to educate developers of distributed systems so that they may make informed decisions about the use of browsers in their systems.

2. Definitions

To examine the problems surrounding the development of browser-based systems, we must provide a characterization of browser development and runtime environments. We will also characterize an alternative approach to implementing distributed systems that does not require the use of a browser.

2.1. Browser-based design

In the runtime environment, the browser is merely the tip of the iceberg. Browsers are used to establish communications with one or more back-end servers, as shown in figure 1. Most commonly, the browser establishes communications with a hypertext transfer protocol (HTTP) server responding to requests on a well-advertised port (commonly, TCP/IP port 80). The HTTP server provides browser content, primarily in the form of hypertext markup language (HTML) pages [Raggett 1998]. The HTTP server may be

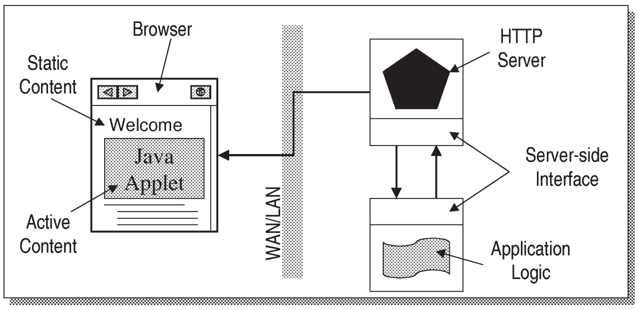

Figure 1. Browser-based design.

configured to communicate with other specialized servers, such as a directory server or certificate server, to provide services such as end-user authentication and authorization.

In the context of distributed system development, the combination of browsers and HTTP servers can be more readily classified as system frameworks in which system logic is inserted using well-defined interfaces. These well-defined interfaces exist both at the client side (i.e., browser) and server side (e.g., HTTP server). Browser capabilities can be extended using plug-ins or by downloading active content from the server. HTTP servers may be extended using a variety of server-side application programming interfaces (APIs). Generally, these APIs fall into one of two mutually exclusive categories: inter-process communication mechanisms and specialized intra-process communication mechanisms.

The communication mechanisms that underlie inter-process server-side interfaces have generally developed independently from any particular HTTP server implementation. Such mechanisms include the Common Gateway Interface (CGI), Windows^{tm} Common Gateway Interface (WinCGI), Common Object Request Broker Architecture (CORBA), Open Database Connectivity (ODBC), and Distributed Component Object Model (COM/DCOM). It is apparent that many of the application program interfaces (APIs) on this list are not unique to browser-based designs and, except for CGI and WinCGI, came into existence independently of browsers and HTTP servers. As such, these mechanisms are available to a broad range of client/server systems and not strictly limited to HTTP servers.

Intra-process communication mechanisms that function as server-side interfaces are typically specific to an HTML server product. Such mechanisms include Netscape Application Programming Interface (NSAPI), Internet Server Application Programming Interface (ISAPI), Active Server Pages (ASP), and Server-Side JavaScript (SSJS).

Regardless of the mechanism, much of the interaction between the end user and the system logic is channeled through the browser's runtime environment and the server-side interface. Application state and session management is encapsulated in the HTTP server. Active and static content in the browser controls presentation management as well as look and feel. Higher level functionality is performed by system logic in the server.

2.2. Non-browser design

Browser-based design is a relatively new paradigm. Prior to the advent of browsers, considerable research and development effort went into the development of client/server, three-tier, message-oriented, transaction processing and distributed object models for open distributed processing. Many of these models use the same inter-process communication mechanisms enumerated in the previous section for extending server-side functionality. Contrasting each of these models with a browser-based design would exceed the scope of this report. Instead, we will focus on distributed object systems as an alternative approach, based on recent interest in this model [Wallnau *et al.* 1997a].

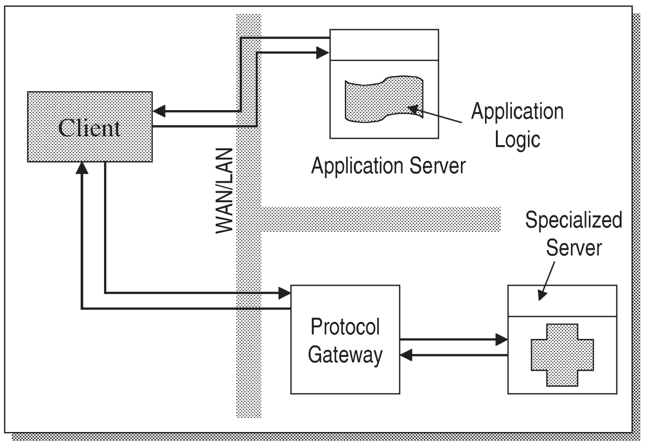

Figure 2. Distributed object system.

Distributed object systems often use a variety of servers and inter-process communication (IPC) mechanisms. Objects operate on a peer-to-peer level in a federated manner. Subordinate, hierarchical relationships are used to encapsulate functionality, act as a protocol gateway, or restrict access through a proxy.

Figure 2 shows an example of a distributed object system that uses an existing, specialized server connected by a protocol gateway. Examples of specialized servers include Lightweight Directory Access Protocol (LDAP) directory servers, mainframe database servers, and messaging servers. A single client object manages interactions with the end user and communications with one or more back-end servers.

We have now provided a characterization of both a browser and non-browser design for implementing distributed systems. In the following section we will consider how these approaches address quality-of-service issues.

3. Issues

In this section, we examine quality of service and other issues that influence the decision to select between browser and non-browser design solutions.

3.1. Portability (crossware)

At a time when technologies such as the Java programming language and CORBA are making the dream of a homogeneous computing environment a reality [Wallnau *et al.* 1997b], browsers are having the opposite effect of fragmenting the market. Competing browsers provide significantly different capabilities. The two principal browsers in use today, Microsoft Internet Explorer (IE) and Netscape Navigator, differ significantly in the manner in which they support key capabilities such as digital certificates, accessing platform resources outside of the browser, and client-

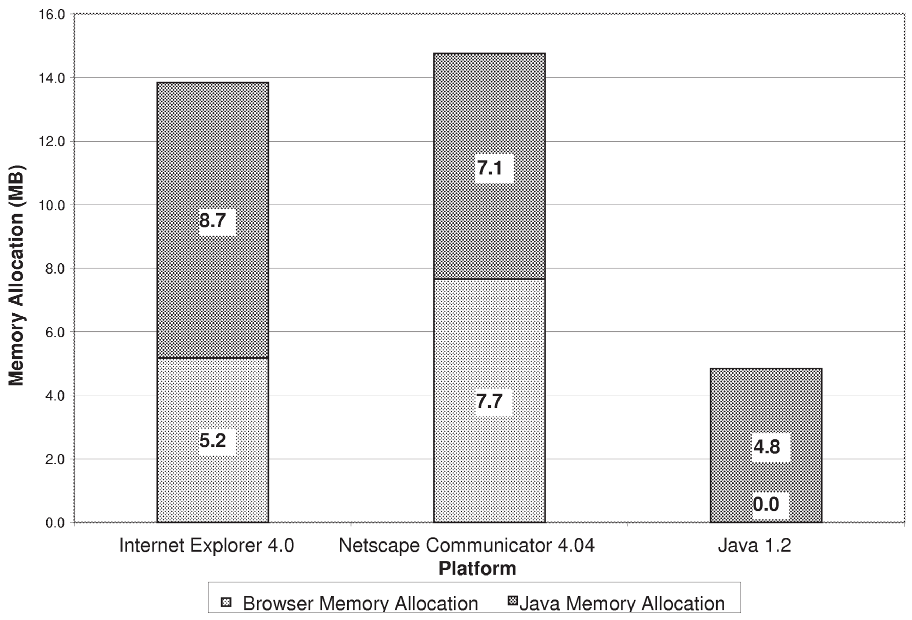

Figure 3. Memory usage.

side scripting. Disparate functionality needs to be accommodated in the design and implementation of browser-based software systems resulting in duplication of code and redundancy of effort.

Browser vendors such as Netscape argue that providing a browser environment that is (roughly) equivalent across most hardware platforms enables the development of crossware applications – applications that run "cross-everything." However, based on statistics gathered from the BrowserWatch web site [Browser], Netscape Navigator has only 54% of the browser market across all platforms (Internet Explorer has 31.8%). Meanwhile, 84% of desktops run Microsoft operating systems (while the remaining 16% are all other platforms). These numbers suggest that a broader market exists for desktop applications that run directly on Microsoft operating systems than "crossware" applications that are confined to a particular browser.

While it may be possible to develop "portable" active content that runs across a variety of different browsers, it may not always be easy – particularly if the active content needs to access local desktop resources. Alternatively, Java provides a standard, controlled environment for developing applications that run on a wide variety of platforms, including the various Microsoft operating systems.

Current standard activities at the World Wide Web Consortium (W3C) and elsewhere including HTML 4, ECMAScript [ECMA 1997], Cascading Style Sheets (CSS) [Lie 1997; Li 1998a, b], Extensible Markup Language (XML) [Browser] and the Document Object Model (DOM) [Wood 1998] provide a roadmap for extending browser capabilities and enabling the development of browser-independent applications.[1]

[1] Browser-independent applications denotes any hardware or software through which a user accesses Web content including different sized displays, printers, and speech synthesizers.

3.2. Performance

Browsers are resource-intensive software components that add an additional layer of functionality between the operating system and application. Figure 3 illustrates a measure of this overhead by showing the memory used by a Java applet under Internet Explorer and Navigator compared to a stand-alone Java application on a Windows NT 4.0 system. Both Internet Explorer and Navigator require considerably more memory than the stand-alone Java application. If the memory required by the browser was completely ignored, the amount of memory required by the browser's Java Virtual Machine (JVM) to run the downloaded applet is still greater than the memory required by the stand-alone Java application.

Browser-based applications are downloaded each time a user accesses the application's Uniform Resource Locator (URL). This constrains the design by severely restricting the size of applications that can be downloaded and run on the client desktop, because the download time – particularly, over modems and congested Internet service providers (ISPs) – can be prohibitive.

The development of Java "orblets" illustrates this problem. An *orblet* is a client of an object request broker (ORB) service. When developing an orblet, the vendor's ORB classes or jar file must be downloaded to the client platform.[2] For version 3.0 of Visigenics ORB, this jar file is 2.2 MB (2,301,416 bytes). Assuming this file is to be transferred over a dedicated 56 Kbps modem, we can use the

[2] Netscape tried to solve this problem by integrating the VisiBroker ORB into Netscape Communicator. Unfortunately, if the browser does not support the correct version of VisiBroker, or the orblet uses a different vendor's ORB; it may still be necessary to download these libraries.

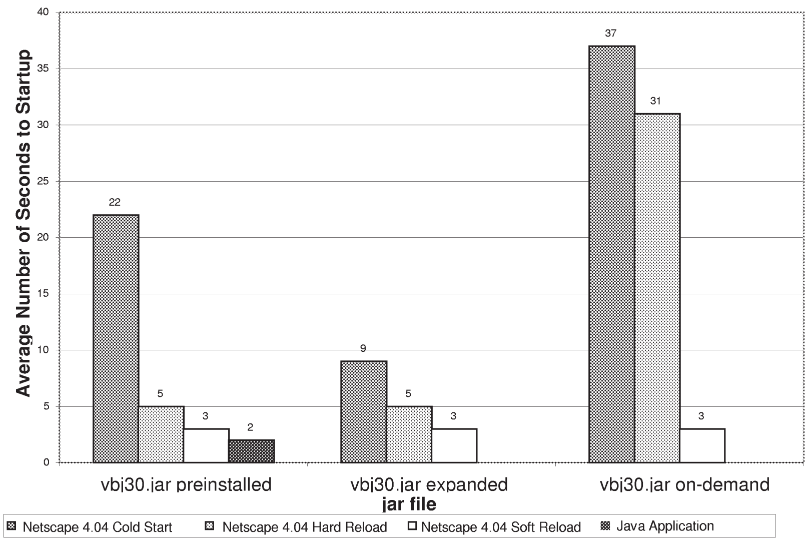

Figure 4. Average start-up time (Java applets vs. Java applications).

formula in equation (1) to predict download time:[3]

$$\begin{aligned}&57{,}344 \text{ bits per second}/10 \text{ bits per byte}\\ &\quad = 5{,}734 \text{ bytes/second},\\ &\text{Bytes transferred} \cdot \text{PPP Overhead}/(\text{bytes/second})\\ &\quad = \text{down load time (seconds)}, \qquad (1)\\ &2{,}301{,}416 \cdot 1{,}02/5{,}734\\ &\quad = 409 \text{ seconds} = \mathbf{6.82\ minutes}.\end{aligned}$$

Performance and network congestion problems caused by multiple users downloading large applications can be alleviated through the use of an *enterprise cache*. An enterprise cache consists of a series of strategically located traffic servers that can "cache" data to ensure that the user has access to the frequently accessed information. These traffic servers ensure that the data is only sent, eliminating redundant traffic over the Internet. However, enterprise caches do not reduce traffic over the local area network, as locally cached data still needs to be transferred to client workstations.

To illustrate performance issues on a typical Intranet, we examined the performance of a Java applet vs. a Java application running over a local 10 Mbps Ethernet-based local area network (LAN) using a Solaris 2.5.1 server and a Windows NT 4.0 desktop. Figure 4 shows the average start-up time (using the wall clock) for a Java applet running under Netscape 4.04 and a native Java 1.2 application running on the Java Runtime Environment (JRE).[4]

[3] The point-to-point protocol (PPP) overhead factor is derived from a discussion of serial line throughput factors described in W. Richard Stevens's book on Transmission Control Protocol/Internet Protocol (TCP/IP) [Stevens 1994].

[4] The JRE is the minimum standard Java platform for running Java programs containing the Java virtual machine, Java core classes, and supporting files. The JRE does not contain any of the development tools (such as appletviewer or javac) or classes that pertain only to a development environment.

Start-up times for the Java applet were recorded using three different scenarios: cold start, hard reload, and soft reload. Cold start was measured from the initial server connection to the time the applet was displayed in the browser. Hard reload is measured from the time Shift-Reload is pressed until the applet redisplays (Shift-Reload causes Navigator to retrieve a fresh version from the network server regardless of whether the page has been changed, effectively bypassing the cache). Soft reload is measured from the time the Reload button is pressed until the applet re-displays. Reload displays a fresh copy of the current Navigator page. Navigator checks the network server to see if the page has changed. If there is no change, the fresh copy is retrieved from the cache. If there is a change, the fresh copy is transmitted from the network server.

In addition to measuring the performance of the Java applet for cold start, hard reload, and soft reload, we also varied the installation of the Visigenics jar file "vbj30.jar" containing the client-side CORBA classes.[5] In the first case, we pre-installed the vbj30.jar file in the CLASSPATH for Netscape Navigator on the client's desktop. In the second case, classes contained in the jar file were expanded. Finally, we conducted the experiment with no pre-installation – requiring that the jar file be installed on demand. By definition, the Java application and associated libraries are pre-installed; therefore we only considered this case for the application.

Figure 5 shows the average number of Ethernet packets transmitted between the client and server for both the Java applet and Java application. This includes both HTTP and Internal Inter-ORB Protocol (IIOP) traffic. Measurements for the Java applet were taken for the three cases discussed earlier. Figure 6 shows the average number of bytes transmitted over the Ethernet for both the Java applet and Java

[5] According to Visigenics, the vbj30.jar file has been divided into three separate files: vbjapp.jar, vbjorb.jar, and vbjtools.jar, to provide a lighter weight client footprint.

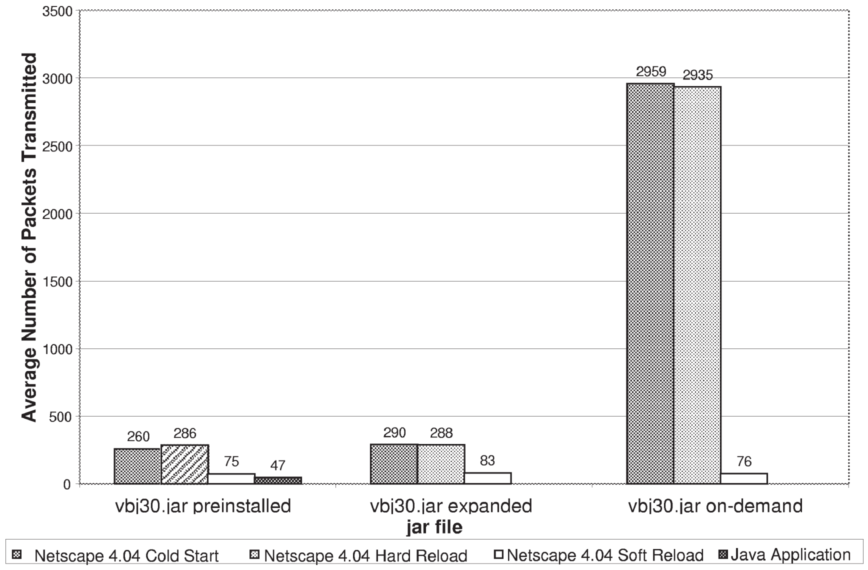

Figure 5. Average number of packets transmitted (Java applet vs. Java application).

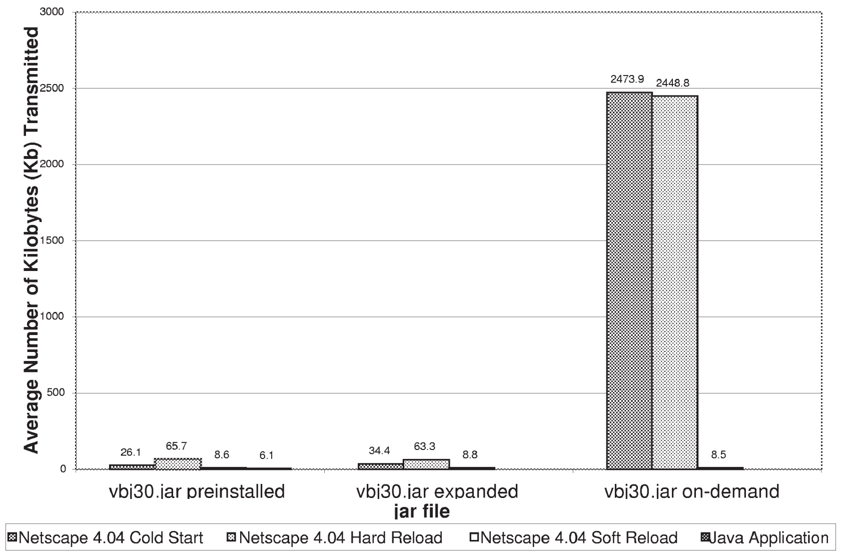

Figure 6. Average number of bytes transmitted (Java applet vs. Java application).

application. This number includes Ethernet, IP, TCP, UDP (User Datagram Protocol), IIOP, and HTTP headers as well as data.

Performance, particularly start-up time, is a major problem with a browser-based approach. Most human factors guidelines recommend start-up times of under one minute, but simply downloading one of the libraries required by the example application in this section exceeds 6 minutes over a 56 Kbps modem. While performance is better in a typical LAN configuration, measured start-up times for an applet were 18 times slower in some cases than a functionally equivalent Java application.

Slow start-up times require that systems be designed to minimize the amount of logic downloaded to the client. This is an artificial design constraint that inhibits the proper separation of functionality between the client and server processes in a distributed system [Seacord 1990].

3.3. Functionality

An obvious advantage of using browser technology for your client application is that the technology provides support for processing hypertext documents specified in HTML. HTML provides a simple means of adding text and graphics to an application and for specifying links between information. The browser provides mechanisms for viewing and navigating through a hypertext document.

The extent to which hypertext is actually used in a graphical user interface (GUI) varies considerably between clients. Hypertext provides a relatively quick mechanism for providing static content, such as text and graphics, around interactive content. Static content can also be added using static, GUI “label” widgets (in X/Motif parlance). Widgets can display fixed graphics or text (in a variety of

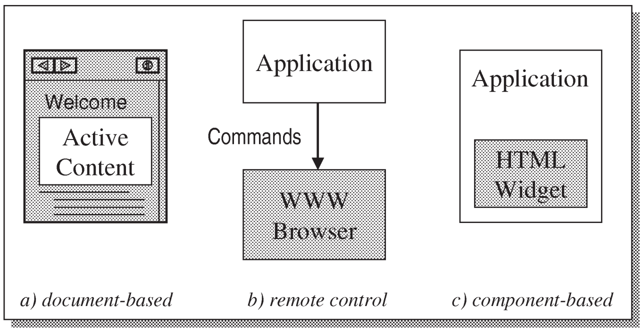

Figure 7. Approaches for integrating hypertext content.

fonts and formats). Adding static content programmatically using GUI widgets is more development intensive than providing HTML markup.

In addition to providing static content, hypertext is useful in more traditional roles as well, such as providing online, interactive help. Netscape, for example, provides a software development kit called "NetHelp" for developing and viewing HTML-based online help. Microsoft has also made the decision to replace WinHelp with an entirely new help-authoring system based on HTML and other Web standards, called HTML Help [Swenson].

Assuming that a valid requirement to integrate hypertext exists, the question becomes: What is the best mechanism for providing this content? Three common approaches used for integrating hypertext content – document-based, remote control and component-based – are shown in figure 7.

The document-based approach assumes the use of a browser. The HTML document becomes the principal container that can hold both hypertext and active content. The component-based approach incorporates an HTML component or control within a client. The HotJava HTML component, for example, is a JavaBean that parses and renders HTML that can be incorporated into an application. In the remote-control approach, the application runs as a separate process from the browser and sends commands to the browser in response to help requests by the end user. The largest problem with the remote-control approach is coordinating activity between the browser and the application, as both systems have independent user controls.

The decision to select a browser-based approach should not be based solely on the fact that HTML content is used in the implementation. The correct approach for integrating static content depends largely on the application, but we would suggest the following. If the application is largely a hypertext document, with some interactive content to enhance the presentation or usability, then it may make sense to take a document-based approach. If the system is principally an interactive application, it may make more sense to take a component-based approach to integrating HTML content. The remote-control approach can be used when a component-based approach makes the most sense, but a suitable component is unavailable.

3.4. Security

Browsers must be careful to restrict access to local resources such as a computer's registry of installed software, file system, or debugging software from downloaded, active content. Destructive programs, such as a Trojan horses, can tamper with or destroy local data or extract sensitive information once they have been downloaded or installed onto a system and allowed to execute. One approach for protecting local resources is to limit access of the active content to within a *sandbox*. The sandbox is a security mechanism that provides a restricted environment in which the active content may safely execute. With Java enabled (e.g., in the Netscape advanced preferences dialog), browsers trust the Java sandbox for any applet.

While this approach may provide an adequate environment for small applets, it is typically too restrictive for the implementation of a large, complex application. To circumvent this problem, browser vendors have invented a variety of mechanisms to extend the sandbox. Signed applets under Netscape can request specific privileges using the *Netscape Capabilities Classes*. These classes allow the applet to request, and the user to grant, permission for the applet to perform specific operations outside of the sandbox. Under Microsoft's Internet Explorer, access to native resources is provided through trust-based security for Java [Microsoft 1997] or simply circumvented through ActiveX controls. The trust-based security model provides fine-grained administration of the privileges granted to Java applets and libraries based on *zones*. Zones allow related sites to be administered as a group. There are five default security zones defined in the trust-based model for IE 4.0: Local Machine, Intranet, Trusted Web, Internet, and Untrusted Web Sites. The idea is to set nonrestrictive security options for trusted areas and, at the same time, have very safe (restrictive) security options elsewhere.

Browsers have a history of alternately disallowing then allowing applications to access local machine resources. Browser vendors cannot seem to decide if they want to allow downloaded applications to access local machine resources or not. The root cause of this conflict stems from the browser's dual roles as hypertext viewer and application development framework. As a hypertext viewer, it is important that the browser not allow downloaded application programs to access local machine resources because of the security risks involved. As an application framework, it is critical that downloaded applications can access local resources to implement advanced application functionality. Because of these conflicting roles, developers are forced to deal with an assortment of specialized security mechanisms and APIs to access functionality outside of the sandbox. This problem is further exasperated by the lack of a standard approach for providing this capability.

In general, security mechanisms for extending the sandbox rely on *object signing* using *digital certificates*. Signing an object (e.g., applet) using a developer's digital certificate identifies the signer and provides tamper-resistant packaging. However, object signing and digital certificate technologies from JavaSoft, Netscape, and Microsoft are largely incompatible. For code signing, Netscape provides signtool for use in Netscape Communicator 4.0. Microsoft offers Authenticode for use in Microsoft's Internet Explorer 3.0 & 4.0. JavaSoft's javakey and jar are used in the JDK appletviewer & the HotJava browser. Netscape's digital certificates can sign Java applets, JavaScripts, plug-ins, or any other kind of code object packaged within a .jar or .zip file. Netscape certificates cannot be used for code signing using JavaSoft's javakey or Microsoft's Authenticode. Microsoft Authenticode certificates can sign 32-bit .exe, .cab, .ocx, and .class files but cannot be used for code signing by JavaSoft's javakey or Netscape's signing tool. Microsoft's Authenticode or Netscape's signing tool cannot use certificates generated by JavaSoft's javakey.

As indicated by the preceding discussion, object-signing technology is still immature and digital certificate technology is in need of further standardization. If the distributed system you are developing has strict security requirements, implementing a browser-based solution requires that you wade into this quagmire and hope you have sufficient stature to keep your head above water.

Digital certificates can also be used for identification and authentication. Identification allows a user to present their credentials to a system. Authentication allows the system to verify that presented credentials are authentic. Once a user's identity is authenticated, the system can determine the user's authorization.

In a browser-based design, identification and authentication are managed between the browser and the HTTP server. The HTTP server interrogates the browser for a client certificate, validates the certificate, and (optionally) looks up the user in a directory server. Authorization for access is granted if the user credentials are found in the directory server. This capability is administered through the HTTP server and does not require the development of custom code.

In a non-browser design, code for certificate manipulation and authentication must be developed. The development of an identification and authorization capability is supported by means of vendor APIs and libraries. Java 1.2 supplies a Certificate API for certificate management; the Netscape LDAP Java SDK can be used for searching the LDAP directory, and encryption and decryption can be performed using patented algorithms from RSA (JSAFE).

An advantage of the non-browser approach is that the authentication policy can be specified, with the corresponding disadvantage that it must be coded. Specifying an authentication policy is necessary if the system has unique authentication requirements. A custom authentication policy has the advantage of being less subject to the attention of hackers, who are likely to target a broadly used authentication policy. This advantage can be quickly lost if the authentication policy is not well considered.

A further disadvantage of a browser-based approach is that to support system auditing, user credentials must be exported from the HTTP server to the system. Under the Netscape Enterprise Server, this is done using Server-Side JavaScript or CGI environment variables. These mechanisms introduce substantial design constraints and add overhead to solve a relatively simple problem.

3.5. Human factors

One of the greatest challenges in designing graphical user interfaces is making the best use of limited display "real estate." When developing a browser-based application, the user interface is limited in size by the real estate required by the browser's window and controls. Even when the majority of these controls are hidden, the real estate required by the browser detracts from the real estate left to the application, particularly on small displays such as those common on laptops and hand-held devices. A typical browser user interface includes a menubar and toolbar that provides commands for filing, editing, viewing, and navigating through a series of Web pages. These commands control functionality in the browser that is independent of any application logic that may be present on any given page. The active content within a page may also have GUI controls, including their own menubar and toolbar. These controls can be used to access commands that interact directly with the application logic, are aware of the application's state, and have access to application data.

From a programming perspective, the above design is flawless. Browser controls are used to interact with the browser, and application controls are used to interact with the active content within the browser. The problem, from a human factors perspective, is that this arrangement is not always apparent to the end user. Consequently, the end user is likely to press browser buttons such as Back, Forward, or Stop in the browser toolbar to interact with the application content. Depending on the application, this can have disastrous effects.

Anecdotal evidence of this problem was gathered from a logistics system in which end users would press the Stop button on the browser in an attempt to end queries that appeared to be hung. Pressing the Stop button in this case caused the client to go away but had no effect on the server, which continued to process the query with no client to receive the results. Dealing with numerous customer complaints and reworking the system to handle this special case resulted in three staff months of diagnosis and repair. In general, unexpected interactions between browser controls and application logic can confuse end users and result in critical errors.

A solution to both these human-factor problems is to launch the active content in a completely separate window. The ability to do this should be a requirement of the mechanism used to extend the browser-based system. However, the requirement to run active content in a separate window and completely independent of browser controls thoroughly refutes the argument for using browsers because they provide a familiar user interface.

3.6. Distribution and installation

Prior to the explosion of the World Wide Web (WWW) and browsers, most companies distributed software on either floppy or CD-ROM, depending on the size of the distribution. Some companies provided customers with an account name and password from which they could download products from an FTP (file transfer protocol) site.

With the advent of the World Wide Web a new model has evolved. Rather than purchase products, users can subscribe to a site or service. Active content in the form of Java applets or ActiveX controls is downloaded at each user access. This model is referred to as on-demand *installation*.

On-demand installation has a number of advantages, chief among them being that downloaded content is current. Ignoring problems with caching, browsers guarantee that both active and static content is current by going back to the source for each user request. Another advantage of on-demand installation is that the disk space used by the client can be automatically reclaimed. The principal disadvantage of on-demand installation includes longer start-up times and the problem that a browser can download only certain types of active content. For example, it is possible to write a Java applet that can be downloaded and executed by both Microsoft IE and Netscape Navigator, although it is necessary to accommodate differences in the Java Virtual Machine (JVM) [Zukowski 1997].[6] IE can also download and execute ActiveX controls. These controls can be written in a variety of languages, including Visual Basic and C++, but must conform to the ActiveX model. Since ActiveX controls are compiled objects, they can run only on the processor for which they were compiled, and only within IE. IE and Navigator provide browser-side scripting languages – instructions to the browser embedded in HTML using the SCRIPT element. Although both browsers support client-side scripting languages, there are differences between Microsoft's VBScript and Netscape's client-side JavaScript.

To address problems of performance and functionality in distributed system development, it is often necessary to install software on the client platform in addition to software installed on demand. We have previously discussed performance problems in downloading large Java class files such as the VisiBroker jar file. Pre-installing the classes on the client machine, for numerous or large files, can help reduce long start-up times, although doing so has further implications that we examine later in this report.

[6] Microsoft, deciding that the core Java class libraries were insufficient for its needs, added about 50 methods and 50 fields into classes within the java.awt, java.lang, and java.io packages. If a developer relies on these changes, or inadvertently uses them, the program will work only within Microsoft's Java system. In addition, a program developed outside of Microsoft's development environment will expect a certain core API. Since the core API is different from the one within Microsoft's environment, the program may not work under the Microsoft JRE. The Netscape Navigator Java 1.1 patch Preview Release 2 also fails to fully implement the Java 1.1 specification.

It may be necessary to pre-install libraries on the desktop to support functionality that is otherwise unavailable. For example, to provide a secure connection between the VisiBroker client and the CORBA server, we would like to use the Secure Socket Layer (SSL). Since the SSL library supplied by Visigenics is implemented in C, it must be preinstalled on the client machine. Plug-ins, another common mechanism for extending the functionality of the browser, also requires an installation step.

An alternative installation approach can be used in a nonbrowser design. Web pages can be provided from which an end user can download the client application. Typically, the distribution file is saved as a compressed archive to reduce the time required to download the file. To install the program, the administrator clicks on the link to the file on the Web page. The browser then prompts the administrator for a location to store the downloaded file. On Windows desktops, the file may be stored as a self-extracting, executable archive. Double clicking this archive causes the program to extract itself and run Install Shield. Under UNIX, the file is normally distributed as a compressed tar file. Compression is usually performed with either the gunzip or compress utility. The corresponding utility is used to decompress the file, and the file is then untarred. The system administrator is normally required to complete the installation by running an installation script.

On-demand installation has advantages in ease of use, since no installation steps are required by the end user except specifying the correct URL; however, on-demand installation has the disadvantage of long start-up times. More complex systems that require the use of client-side libraries (e.g., for security) or browser plug-ins have no choice but to go through an installation process.

3.7. Upgrading and component-based development

On-demand installation simplifies the problem of providing software updates, at least from the point of view of the vendor. New versions of the product installed on the vendor's Web site are automatically downloaded and accessed by customers the next time they use the application. Software updates downloaded to the client machine are able to communicate directly with updated versions of servers at the vendor's site.

The principal problem with this approach is that the customer no longer has a say in the decision to upgrade a system. Under this model, it is normal for upgrades to occur overnight and without warning. This can cause immediate problems: end users are caught unaware of major changes in functionality and interface and are unable to adequately prepare or train prior to the upgrade. Documented processes and procedures can be rendered instantly obsolete.

Within the commercial world, developing component-based systems can be viewed as a "just-in-time" programming model where components move along an assembly line from the developer through the integrator to the end user, and functionality is added at the latest possible point along this line. Component developers build large-scale components that provide services that appeal to a large market; integrators extend and combine the components to build systems; and end users and their support staff tailor the system for local needs [Vigder *et al.* 1996].

Automatic upgrades limit a developer's ability to integrate a system's functionality as a component of a larger system. Tools such as the Web Interface Definition Language (WIDL) allow the resources of the World Wide Web to be described as functional interfaces that can be accessed by remote systems using standard Web protocols [Allen 1997]. Once functionality has been incorporated into a component-based system, either directly or through the use of WIDL or a similar tool, changes in the functionality would disrupt the operation of the overall system. Consequently, it is difficult, if not impossible, to use a system that is automatically upgraded as a component of larger systems, thus creating a further inhibitor to the development of component-based systems.

The alternative to on-demand installation and automatic upgrades requires that the users upgrade their systems. This approach eliminates the above-described disadvantages experienced by the users while shifting the problem to the vendors. The vendor now has no control over when users will upgrade their systems. This puts vendors in the awkward position of supporting obsolete versions of client software beyond a reasonable period. Upgrades to the server can be inhibited or prevented by a need to maintain compatibility with the installed customer base.

A compromise solution that can be implemented in the design of non-browser distributed systems provides for negotiation between the client and server concerning discrepancies in version numbers. For example, a tool provided by the National Software Data and Information Repository (NSDIR) [Card and Hissam 1996] used a four-digit version number scheme to characterize upgrades. Version numbers downloaded from the server are compared to version numbers on the client. If there is a discrepancy the client notifies the user if the change represents a maintenance release, inaccessible capabilities, reduced functionality, or a change in the application protocol stream. Maintenance releases and inaccessible capabilities can generally be ignored without affecting a system for which this may be a component. Reduced functionality may affect the overall system, while a change in protocol will require an upgrade. The user (or integrator) has the ability to upgrade based on their own drivers, while the vendor has a better lever to move customers onto newer versions.

3.8. Runtime configuration management

In the section on distribution and installation, we argued that there is often a requirement to pre-install software on the client platform when building browser-based distributed systems. Requiring a pre-installation step for performance, functionality, or other reason means that there are now software components installed on the client machine. These versions evolve over time, but may be upgraded at will by the customer. At the same time, downloadable active content is updated by the vendor based on availability. This executable content must determine which versions of the pre-installed components are available and if it can work with them or if newer versions must first be installed. Implementing this solution requires that the executable content operate outside of the sandbox, requiring permissions that are considered high risk. This mixed model of pre-installation and on-demand installation creates combinatorial problems in runtime configuration management that outweigh any advantage in the browser model.

Another configuration management problem is the tendency of end users to extend, customize, and upgrade their browsers. Browsers are general-purpose tools used to access data and information services over the Internet. As such, the end user is given endless opportunities to install plug-ins, patches, and upgrades. Each of these changes has the potential to affect other applications that run within the browser. For example, during 1997 Microsoft Corporation updated the Authenticode security module installed with Microsoft's Internet Explorer from version 1.0 to 2.0. Notices regarding this update were embedded in Web pages at Microsoft's Web Site as an embedded JavaScript routine. Upon reaching one of those pages, the client's browser would execute the JavaScript routine, and if Authenticode 1.0 were detected, the script would display a dialog box suggesting the end user upgrade by pressing the Yes button. The irritating behavior of this "nag-ware" would continue until the end user gave in and upgraded the client browser's security component to the current version. In one case the result of this upgrade completely disabled a functioning Java/CORBA orblet – with no way of reversing the upgrade without completely removing the browser from the system and re-installing a previous release of the browser (before Authenticode version 2.0 was introduced).

3.9. Licensing

Vendors have been (relatively) quick to recognize the necessity of supporting browser-based designs by offering server-based licensing. Server-based licensing allows unlimited access by a potentially unknown collection of end users. However, this same consideration has not been generally extended to client/server or distributed object systems.

The architecture of a system is easily influenced by deployment costs driven by the licensing scheme of COTS vendors. This can best be shown by an example. Figure 8 shows a browser-based scenario (A) for deploying a CORBA-based applet and a separate scenario (B) in which a Java application communicates directly with the ORB on the server.

In both scenarios the classes contained in the vbj30.jar file are installed on the desktop and run locally. In scenario A, the classes are installed on demand by the server, stored in the browser's cache, and eventually removed by

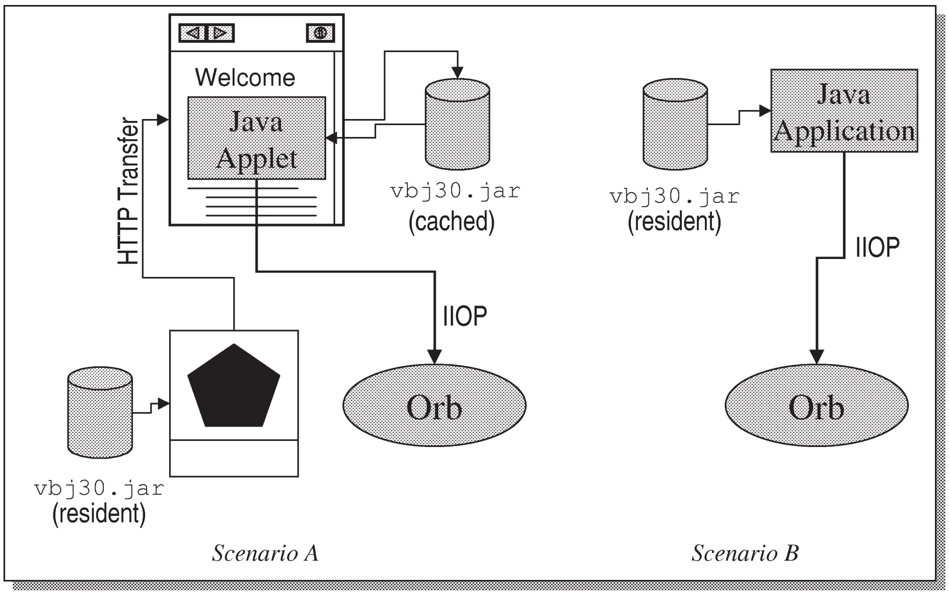

Figure 8. Licensing scenarios.

the browser. In scenario B, the classes are pre-installed on the desktop.

Because it is possible to distinguish between these scenarios, it is likely that some vendor's software licenses do make this distinction. If the cost of deploying a browser-based design is considerably lower than a similar non-browser-based design, it is unlikely that the development organization would incur the added costs to deploy a non-browser-based solution.

3.10. Versions

Beta versions of Java 1.1 were available in December of 1996, and it was generally available in the first quarter of 1997. However, as of March 1998, full 1.1 support is still not available in the standard release of Netscape Communicator. In some sense, this is the traditional COTS problem of incompatible versions, but in another sense it is worse. When developing a C++ application, for example, language functionality is available to the developer as soon as the compiler version is released. If necessary, language libraries can either be statically linked with the executables or installed on the client machine. Likewise, applications developed in Java can be installed, if necessary, with the corresponding JRE version [Sun]. However, when running Java applets within a browser, the browser dictates the Java Development Kit (JDK) version and, therefore, which software is compatible.

In response to this problem, Netscape plans to re-architect its future client software products to support compatible implementations of a Java Virtual Machine from industry leaders such as Sun, IBM, and other operating system vendors. Netscape plans to provide an OpenJava API designed to make it easier for vendors to integrate their native Java VM into Navigator or Communicator.

Microsoft has no public plans to change their current strategy of maintaining their own version of the Java programming language as a proprietary Windows development tool.

4. Conclusions

In a browser-based system, the combination of the browser and HTTP server forms the backbone of the system. While this backbone provides some flexibility and extensibility, it still provides a rigid framework that can only be bent so far without breaking. Functionality can be added only if the designers of the browsers anticipated the needs of your application directly or indirectly by providing appropriate hooks.

A browser-based design may be appropriate if the requirements of the system conform naturally to a browser infrastructure. For hypertext systems, Internet browsers are an ideal solution. Based on the current state of the practice, browser-based designs are not appropriate under the following conditions:

- The desktop client has a large, complex user interface.
- The desktop client requires access to local machine resources.
- Fast application start-up time is an important requirement.
- The system must communicate securely across multiple protocols, browsers, and servers.

Distributed object systems offer an alternative to a browser-based design. A distributed object system can offer faster start-up times and support larger, more complex user interface designs. The application can be implemented in Java for maximum portability, using either Java Remote Method Invocation (RMI) or CORBA for communication with back-end servers. Software development kits are available to provide safe, secure communications between distributed objects.

The use of a browser-based infrastructure is a major decision that is going to influence and limit the overall architecture and design of your distributed system. As such, it is critical that this is an informed, considered decision.

Acknowledgements

The U.S. Department of Defense (DoD) office sponsored this work. Special thanks to Kurt Wallnau and John Foreman for supporting this work. Thanks also to external reviewers Steve Oesterle (Visigenics), Daniel Dardailler (W3C), and Kevin Samborn, and to SEI reviewers Dan Plakosh, Tricia Oberndorf, Jeromy Carriere, Lisa Brownsword, Fred Long (SEI visiting scientist from the University of Wales), Fred Hansen (SEI visiting scientist from the Andrew Consortium), Eileen Forrester, Suzanne Couturiaux, and Edwin Morris.

References

Allen, C.A. (1997), “Automating the Web with WIDL,” *World Wide Web Journal 2*, 4.

Bray, T., J. Paoli, and C.M. Sperberg-McQueen (1998), *Extensible Markup Language (XML) 1.0 Specification, W3C Recommendation* [online]. `http://www.w3.org/TR/REC-xml.`

BrowserWatch Stats Station [online]. `http://browserwatch.internet.com/stats/stats.html.`

Card, D.N., S.A. Hissam, and R.T. Rosemeier (1996),“National Software Data and Information Repository,” *CrossTalk* 9, 2. Software Technology Support Center. `http://www.stsc.hill.af.mil/CrossTalk/1996/feb/national.html.`

ECMA-262 (1997), *ECMAScript: A General Purpose, Cross-Platform Programming Language* [online]. `http://www.ecma.ch/stand/ecma-262.htm.`

Lie, H.W. and B. Bos (1997), *The Cascading Style Sheets – Designing for the Web,* Addison-Wesley/Longman, Essex.

Lie, H.W. and B. Bos (1998), *Cascading Style Sheets, Level 1, W3C Recommendation* [online]. `http://www.w3.org/TR/.`

Lie, H.W., B. Bos, C. Lilley, and I. Jacobs (1998), *Cascading Style Sheets, Level 2, CSS2 Specification, W3C Recommendation* [online]. `http://www.w3.org/TR/REC-CSS2.`

Microsoft (1997), *Trust-Based Security for Java*, Microsoft white paper [online]. `http://www.microsoft.com/java/security.`

Raggett, D., A. Le Hors, and I. Jacobs (1998), *HTML 4.0 Specification, W3C Recommendation* [online]. `http://www.w3.org/TR/REC-html40.`

Seacord, R.C. (1990), “User Interface Management Systems and Application Portability.” *IEEE Computer 23*, 10, 73–75.

Stevens, W.R. (1994), *TCP/IP Illustrated*, Vol. 1: *Protocols*, Addison-Wesley, Reading, MA.

Sun Microsystems, *The Java Runtime Environment Notes for Developers* [online]. `http://java.sun.com/products/jdk/1.1/runtime.html.`

Swenson, J. “Making the Big Move to HTML Help,” *MSDN Online*. `http://www.microsoft.com/msdn/news/htmlhelp.htm.`

Vigder, M.R., W.M. Gentleman, and J.C. Dean (1996), *COTS Software Integration: State of the Art* [online]. Software Engineering Group (NRC No. 39198). `http://wwwsel.iit.nrc.ca/seldocs/cotsdocs/NRC39198.pdf.`

Wallnau, K., E. Morris, P. Feiler, A. Earl, and E. Litvak (1997), “Engineering Component-Based Systems with Distributed Object Technology,” In *Proceedings of Worldwide Computing and Its Applications,* Springer-Verlag, Heidelberg, Germany, pp. 58–74.

Wallnau, K., N. Weiderman, and L. Northrop (1997), “Distributed Object Technology with CORBA and Java: Key Concepts and Implications,” Technical report CMU/SEI-97-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. `http://www.sei.cmu.edu/publications/documents/97.reports/97tr004/97tr04abstract.html.`

Wood, L. et al. (1998), *Document Object Model (DOM), Level 1, Specification Version 1.0, W3C Recommendation* [online]. `http://www.w3.org/TR/REC-DOM-Level-1.`

Zukowski, J. (1997), “How to Avoid Potential Pitfalls of Microsoft’s Non-Standard SDK for Java,” *Java World* [online]. `http://www.javaworld.com/javaworld/jw-11-1997/jw-11-pitfalls.html.`