ISR Institute for Software Research
University of California, Irvine

Architectural Styles of Extensible REST-based Applications

Justin R. Erenkrantz
University of California, Irvine
jerenkra@ics.uci.edu

August 2006

ISR Technical Report # UCI-ISR-06-12

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

Architectural Styles of Extensible REST-based Applications

Justin R. Erenkrantz
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
jerenkra@ics.uci.edu

ISR Technical Report # UCI-ISR-06-12

August 2006

Abstract:

At the beginning of the World Wide Web (WWW or Web), there was no clear set of principles to guide the decisions being made by developers and architects. In these early days, a cacophony emerged without a clear direction to guide the evolution of the Web. If there was any direction during the inception of the Web, it was a weak focus on how communication might occur between machines on the Web and the content that was to be transferred. Within a matter of a few years, scalability and other design concerns threatened the future of the early Web - this led to the introduction of REpresentation State Transfer architectural style (REST). The REST style imposed constraints on the exchange of communication over the Web and provided guidance for further modifications to the underlying protocols. The introduction of REST, through the HTTP/1.1 protocol, restored order to the Web by articulating the necessary constraints required for participation.

In this survey, we will characterize any environment that is governed by REST constraints to be in a RESTful world. Obviously, the largest example of the RESTful world is that of the Web with almost 75 million websites existing today and many more daily users. Yet, to this day, people are still struggling with how to write applications and architectures that adhere to the constraints of the REST architectural style. Consequently, it is all too common to see programs falling into a trap of ignoring and compromising the REST principles. These traps can jeopardize the beneficial induced properties dictated by the REST style - which could ultimately reintroduce the problems that REST was specifically imposed to address.

The existing Web infrastructure, and especially important components of that infrastructure like Apache, Mozilla, and others, can inform us about how to implement other RESTful components; indeed, examining the architectures of these tools and the infrastructure as a whole is key. With the rich history of the Web, we now have over ten years of real-world architectural evolution from which to base our examinations. Our aim in this survey is to classify the evolution, supported by real software architectures and frameworks, and to indicate insights and techniques useful for developing applications as a whole - that is, complete configurations of RESTful nodes that together form RESTful software applications without compromising the beneficial properties of REST.

ISR Technical Report #UCI-ISR-06-12
August 2006

Architectural Styles of Extensible REST-based Applications

Justin R. Erenkrantz

This paper is a survey of past and current architectural styles used in applications that take part in the RESTful world.

At the beginning of the World Wide Web (WWW or Web), there was no clear set of principles to guide the decisions being made by developers and architects. In these early days, a cacophony emerged without a clear direction to guide the evolution of the Web. If there was any direction during the inception of the Web, it was a weak focus on how communication might occur between machines on the Web and the content that was to be transferred. Within a matter of a few years, scalability and other design concerns threatened the future of the early Web - this led to the introduction of Representational State Transfer architectural style (REST) [Fielding 2000, #36]. The REST style imposed constraints on the exchange of communication over the Web and provided guidance for further modifications to the underlying protocols. The introduction of REST, through the HTTP/1.1 protocol, restored order to the Web by articulating the necessary constraints required for participation.

In this survey, we will characterize any environment that is governed by REST constraints to be in a "RESTful world." Obviously, the largest example of the RESTful world is that of the Web with almost 75 million websites existing today and many more daily users[Netcraft 2005, #107]. Yet, to this day, people are still struggling with how to write applications and architectures that adhere to the constraints of the REST architectural style. Consequently, it is all too common to see programs falling into a trap of ignoring and compromising the REST principles. These traps can jeopardize the beneficial induced properties dictated by the REST style - which could ultimately reintroduce the problems that REST was specifically imposed to address.

Looking to the REST architectural style for answers on how to construct RESTful applications leads to an ultimately unfulfilling experience. The REST architectural style purposely provides little-to-no guidance as to how to build such nodes in a principled manner. Other architectural styles, like C2[Taylor, Medvidovic 1996, #135] and PACE[Suryanarayana, Erenkrantz 2004, #134], and practical Web frameworks, like

Axis[The Apache Software Foundation 2005, #146] and Ruby on Rails[Hibbs 2005, #53], specifically constrain how an application is built. However, these particular styles and frameworks do not provide much guidance for interactions with other architectures. Viewed from this perspective, we can separate these architectural styles in two categories: *internal* and *external* architectural styles. An external architectural style, like REST, will govern the interaction between two independent sub-architectures, while an internal architectural style, like PACE, governs how an architecture will respond to the constraints imposed by an external architecture.

The larger research question is what is the relationship between an external architecture and an internal architecture? That is, how do the constraints placed on the network and interaction between nodes affect the constraints placed on individual nodes and vice versa? Are there particular internal architectural styles that are a 'good' fit for an external architecture? Correspondingly, are there 'poor' matches? What are the tradeoffs in selecting, say, a RESTful network architecture and combining it with an internal pipe-and-filter architecture? Are induced properties sacrificed in trying to make this combination work?

The existing Web infrastructure, and especially important components of that infrastructure like Apache, Mozilla, and others, can inform us about how to implement other RESTful components; indeed, examining the architectures of these tools and the infrastructure as a whole is key. With the rich history of the Web, we now have over ten years of real-world architectural evolution from which to base our examinations. Our aim in this survey is to classify the evolution, supported by real software architectures and frameworks, and to indicate insights and techniques useful for developing applications as a whole—that is, complete configurations of RESTful nodes that together form RESTful software applications without compromising the beneficial properties of REST.

Software Architecture and Frameworks

An architectural style is a set of design guidelines, principles, and constraints that dictate how components can be composed, behave, and communicate [Shaw and Garlan 1996, #125]. Architectural styles help to induce desirable qualities over software systems that conform to those styles. Many of the most well-known architectural styles, such as pipe-and-filter, client-server, and blackboard styles provide relatively few principles and constraints; as one might expect, they also induce relatively few good software qualities. However, there are other architectural styles, such as PACE, that are much more significant. These include comprehensive constraints and guidelines, provide knowledge about when and where these styles are applicable, how to apply the style, and supply technological frameworks and tools to facilitate constructing applications in the style. As we will discuss later, the REST style provides such constraints and guidelines for external architectures.

An architecture framework is software that helps to bridge the gap between a specific architectural style (or family of styles) and an implementation platform (e.g., programming language, core set of libraries, or operating system). This makes it easier for appli-

cation developers to correctly (and compatibly) implement applications in a particular architectural style. For example, it could be said that the `stdio` package is an architecture framework for the pipe-and-filter style in the C programming language, since it provides the language with distinguished stream constructs (`in`, `out`, and `err`), as well as methods for interacting with those streams that are consistent with the rules of the pipe-and-filter style.

Architecture frameworks (even for the same style/implementation platform) can vary widely in the amount of support they provide to developers. This is a natural tradeoff: frameworks may provide little support but be very lightweight, or be heavyweight and complex but provide many services.

Software Architecture in the World Wide Web

It is essential to understand the intimate relationship between the architectural style, architecture instances, and actual system implementations. In the context of the modern Web, some of the key participants are:

- REST - the principal architectural style
- HTTP/1.1 - an architectural instance of REST
- Apache HTTP Server - a system implementation of an HTTP/1.1 server
- Mozilla - a system implementation of an HTTP/1.1 user agent
- libWWW - an architectural framework providing useful services for implementing RESTful clients

Next, we will examine the REST architectural style and the constraints that it imposes on the RESTful HTTP/1.1 protocol. After introducing REST and HTTP/1.1, we will begin selecting systems to survey.

Representational State Transfer

The Representational State Transfer (REST) architectural style minimizes latency and network communication while maximizing the independence and scalability of component implementations. Instead of focusing on the semantics of components, REST places constraints on the communication between components. REST enables the caching and reuse of previous interactions, dynamic substitutability of components, and processing of actions by intermediaries - thereby meeting the needs of an Internet-scale distributed hypermedia system. A summary of the domain properties, REST constraints, and REST-induced behavior is presented in Table 1 on page 4.

The first edition of REST was developed between October 1994 and August 1995, primarily as a means for communicating Web concepts while developing the HTTP/1.0 specification and the initial HTTP/1.1 proposal. It was iteratively improved over the next five years and applied to various revisions and extensions of the Web protocol stan-

TABLE 1. Summary of REST constraints in terms of domain and induced properties

Domain Property	REST-imposed Constraint	REST-induced Benefit/Property
A user is interested in some hypermedia document stored externally	User Agent represents User Origin Server has hypermedia docs	User Agent initiates pull-based request from an Origin Server Requests from User Agent have a clearly associated response from an Origin Server
Hypermedia documents can have many formats	**Metadata** describing **representation** presented with document	User Agent can render documents appropriately based on metadata
Many independent hypermedia origin servers	Define a set of common operations with well-defined semantics (**Extensible methods**)	User Agent can talk to any Origin Server
A document may have multiple valid depictions with differing metadata	Distinction between abstract **resource** & transferred **representation** Metadata can be sent by user agent that indicates preferences (**Internal transformation**)	User Agent can request resource and receive an appropriate representation based on presented metadata One-to-many relationship between a resource and representation
Hypermedia documents are usually organized hierarchically + uniquely identified servers	Resources explicitly requested by name	User Agent can 'bookmark' a location and return to it later
	Origin Server controls own namespace	Origin Server can replace backend and persist identical namespace
Origin Server may not be able to receive inbound connections from the world User Agent may not be able to make outbound connections to the world	Gateway node (Origin Servers) **Proxy** node (User Agents)	Even if direct paths are not available between two nodes, indirect paths may be available through REST intermediaries
	No assumption of persistent connection or routing; Hop-by-hop only Any **state** must be explicitly transferred in each message	Gateway and Proxy nodes treat routing of each message independently (packet-switched) Duplicate copies of Origin Servers may be deployed
Common hypermedia operations do not change the content + Documents may change over time	Idempotent methods	Ability to reuse a representation
	Cacheability components introduced	Each node can independently have a local cache of documents; cache can re-serve representations
REST nodes may need to handle large amounts of traffic or otherwise optimize network bandwidth	Expiration control data can be presented with a representation	Mechanism to locally expire cached content
	Control data presented in requests to indicate current cached version	Mechanism to cheaply re-validate 'stale' content in the cache

dards. The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application may behave: a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.

In writing RESTful applications, it is essential to understand that all REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it. This restriction accomplishes four functions: 1) it removes any need for the connectors

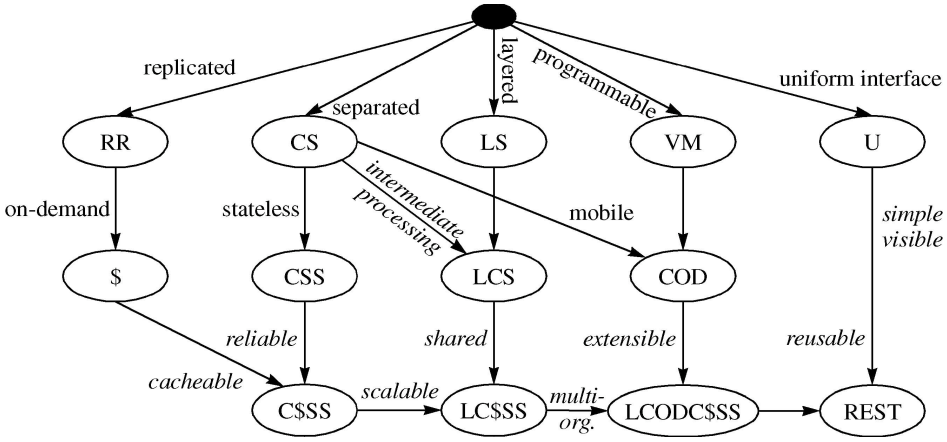

FIGURE 1. REST style derivation diagram (From [Fielding 2000, #36])

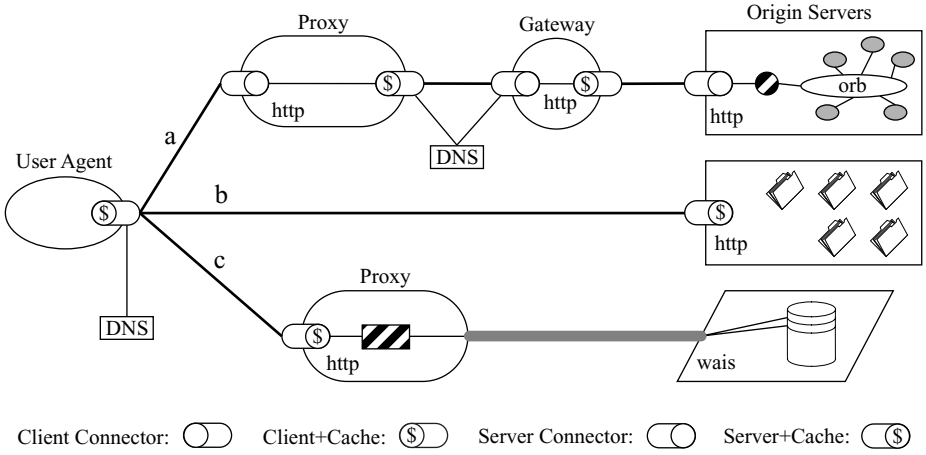

FIGURE 2. Process view of a REST-based architecture at one instance of time (From [Fielding and Taylor 2002, #37])

to retain application state between requests, thus reducing consumption of physical resources and improving scalability; 2) it allows interactions to be processed in parallel without requiring that the processing mechanism understand the interaction semantics; 3) it allows an intermediary to view and understand a request in isolation, which may be necessary when services are dynamically rearranged; and, 4) it forces all of the information that might factor into the reusability of a cached response to be present in each request.

Another important contribution of REST is the layer of indirection between abstract resources and concrete representations. The key abstraction of information in REST is a *resource*. Any information that can be named can be a resource: a document or image, a temporal service (e.g. “today’s weather in Los Angeles”), a collection of other

resources, a moniker for a non-virtual object (e.g. a person), and so on. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time. In turn, REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A *representation* is a sequence of bytes, plus *representation metadata* to describe those bytes.

As characterized in Figure 1, REST is derived from a number of specific constraints. The relevant base styles from which REST were derived include Replicated Repository (RR), Cache ($), Client-Server (CS), Layered System (LS), Stateless (S), Virtual Machine (VM), Code on Demand (COD), and Uniform Interface (U). Furthermore, REST defines a series of connector types that identify each node in the overall architecture: client, server, cache, resolver, and tunnel. As depicted in Figure 2, in a typical REST interaction on the modern Web, a user agent (web browser) requests a representation of a resource from an origin server, which may pass through caching proxies before ultimately being delivered.

HTTP/1.1 and the Modern Web

The Hypertext Transfer Protocol (HTTP) has a special role in the Web architecture as both the primary application-level protocol for communication between Web components and the only protocol designed specifically for the transfer of resource representations. REST was used to identify problems with the existing HTTP implementations, specify an interoperable subset of that protocol as HTTP/1.0 [Berners-Lee, Fielding 1996, #12], analyze proposed extensions for HTTP/1.1 [Fielding, Gettys 1999, #31], and provide motivating rationale for deploying HTTP/1.1.

The key problem areas in HTTP that were identified by REST include planning for the deployment of new protocol versions, separating message parsing from HTTP semantics and the underlying transport layer (TCP), distinguishing between authoritative and non-authoritative responses, fine-grained control of caching, and various aspects of the protocol that failed to be self-descriptive. REST has also been used to model the performance of Web applications based on HTTP and anticipate the impact of such extensions as persistent connections and content negotiation. Finally, REST has been used to limit the scope of standardized HTTP extensions to those that fit within the architectural model, rather than allowing the applications that misuse HTTP to influence the standard equally.

REST Mismatches in HTTP Extensions

HTTP/1.1 as it is used on the Web today (taking into account third-party extensions deployed outside the standards process and concessions made for backward compatibility with HTTP/1.0) does not conform entirely to the REST style. Applications that support HTTP/1.1 consequently must make allowances for these non-RESTful characteristics of the Web to remain compatible and interoperable with the substantial base of legacy applications already deployed. While complete details of these mismatches are out of scope, it is useful to examine a representative example of how the Web is not 100% RESTful (for more details, please see [Fielding 2000, #36, Fielding and Taylor 2002, #37]).

Perhaps the most pervasive example of the non-RESTfulness of the Web is the use of Cookies for client-side state management. In this case, an inappropriate extension has been made to the protocol to support features that contradict the desired properties of the generic REST interface [Kristol and Montulli 1997, #63]. Cookie interaction fails to match REST's model of application state, often resulting in confusion for the typical browser application. The same functionality should have been accomplished via anonymous authentication and true client-side state. Cookies also violate REST because they allow data to be passed without sufficiently identifying its semantics, thus becoming a concern for both security and privacy.

Selecting Appropriate RESTful Applications

Our primary criteria in selecting systems is that such systems must participate in the RESTful world. Due to the ubiquitous deployment of the World Wide Web, there are plenty of RESTful systems that are available to examine. Furthermore, one of our stated goals is to examine how the constraints of a REST-governed external architecture as represented by its protocol affects the internal architecture of a system. In order to achieve this goal, our examined architectures should reside as close to the protocol as possible - this can best be achieved by directly implementing the HTTP/1.1 protocol.

REST also defines specific classifications of nodes that can participate in the RESTful world. We will limit our discussion to origin servers, user agents, and frameworks. As we will discuss, some origin servers can also fulfill the responsibilities of a proxy or gateway node. With respect to the specific selections we make, we will attempt to choose a representative sample of the broad range of RESTful systems that are available.

Our focus on architectures will look at their extensibility characteristics - that is, the constraints it imposes on modifications to its architecture. The reason for selecting architectures that explicitly support extensiblity is predicated on the diverse nature of the RESTful world. The particular location in the ecosystem of the RESTful world we will examine is an important one: complete RESTful applications are built on top of these architectures. These architectures we will survey provide the glue by interfacing with the larger RESTful world through protocol implementations and passing along the constraints of the RESTful world on to its extensions to form complete applications.

A vast range of applications have emerged that use the WWW in innovative ways - ranging from electronic-commerce sites to collaborative news sites. The specific content requirements often differ for each individual application. Instead of constructing an origin server or user agent from scratch each time for every desired modification, these applications can take advantage of pre-existing architectures if they provide suitable extensibility mechanisms. Therefore, those architectures which support extensibility have a definitive advantage over static architectures in the RESTful world.

While our principal focus is on applications directly implement a REST-governed protocol and offer extensibility capabilities, we will also present a brief discussion of:

- Server-side scripting languages (such as CGI, PHP, JSP)

- Client-side scripting languages (such as JavaScript)
- HTML forms
- Protocols on top of HTTP (such as SOAP, XML RPC)
- HTML frameworks (such as Servlets, Struts, Ruby on Rails)

to discuss how they can further encourage conflicts and collisions with REST. However, these applications traditionally build on top of the systems that we will select to survey in detail. With these systems, there is an additional level of indirection with regards to REST as they are necessarily constrained by the architectures of which they are a smaller part.

Framework Constraint Prism

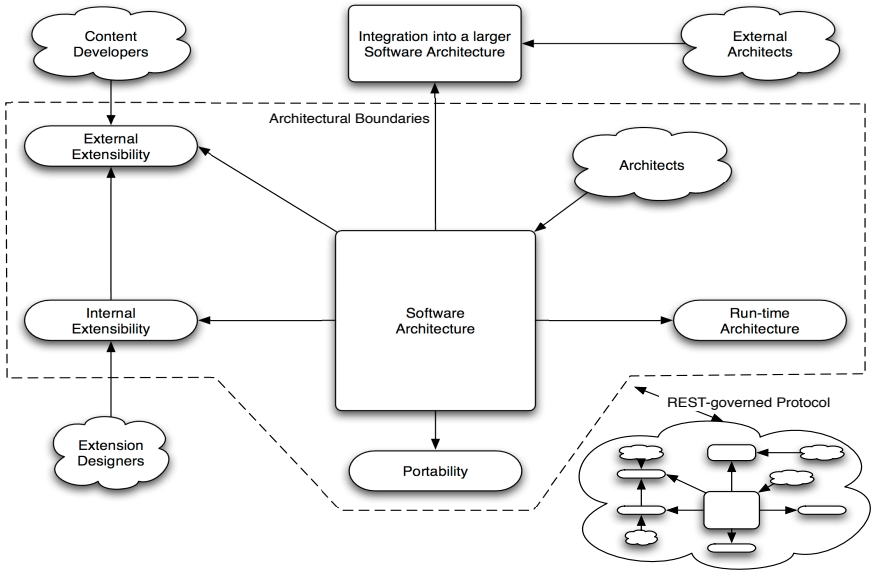

FIGURE 3. RESTful Architectural Constraint Prism

In order to highlight the relevant material, our architectural examination will separate the architecture into the following broad characteristics: portability, run-time architecture, internal extensibility, external extensibility, and the influence of specific REST constraints. A diagram showing the relationships of these characteristics for RESTful architectures is shown in Figure 3 on page 8. These characteristics are derived from the direct decisions made by the system's architects. We will define the border of our architecture by identifying where an architect has direct influence over the architecture and where control of the architecture is ceded to others. As a part of a larger RESTful world, these architectures must operate with other independent architectures through a REST-governed protocol. These architectures can also be integrated by external architects into a larger architecture to incorporate the functionality provided by the system.

Content developers and extension designers can influence the system, but this work is largely limited to following the constraints established by the original architects.

Portability. We will define portability as the indirect limitations and constraints upon the overall system architecture with respect to its environment. These may include the choice of *programming languages* to implement with, *operating systems* that the system will execute on, and *user interface toolkits*. As we will discuss, each of these choices can introduce constraints for robustness, scalability, and security. They may affect the degree to which the system can conform to the environment in which it must operate. However, these choices are typically not directly related to the functionality of the system. These serve as the base platform characteristics.

Run-time architecture. In contrast to portability, run-time architecture will be defined as the specific direct limitations and constraints that the architecture represents with respect to the problem domain. Such constraints can include how the system *parallelizes*, whether it is *asynchronous*, and what *protocol features* and *protocols* it supports. These constraints are generally decided independently of any constraints defined as Portability. By building on top of these run-time architecture constraints, a system will present characteristics that govern what features it will ultimately be able to support.

Internal Extensibility. We will define internal extensibility as the ability to permit modification through explicit introduction of architectural-level components. This characterizes the scope of changes that can be made by third-party developers in specific *programming languages*. The critical characteristic here is what functionality does the architecture provide to developers to *modify* the behavior of the system. For a user agent, new toolbars can be installed locally through specific extensions. These toolbars can change the behavior of the program via the internal extensibility mechanisms. Or, perhaps, *new protocols* can be introduced.

External Extensibility. Similarly to internal extensibility, we will define the external kind as those changes that can be effected without the introduction of architectural-level components. This classifies what behavior can be passed through to the user without altering the architecture. Each of these specific external extensibility mechanisms can be viewed on a cost-benefit scale: how much *access* is provided at what *cost*? For an origin server, a scripting language like PHP can be viewed as an external extensibility component. A PHP developer can create a script that alters the behavior of the system without any knowledge of the architecture inside the system. Typically, RESTful systems in the same area (such as user agents) will share external extensibility mechanisms.

Integration. Integration defines the ability of an architecture to participate as part of a larger architecture. Some architectures that we will examine are intended to run only by themselves. However, other architectures offer the additional capability to be reused as part of a larger whole through a set of *programming languages*. These architectures may provide *control* ranging from simply integrating a user agent inside another application to creating a different type of server entirely.

REST Constraints. The final broad characteristic we will leverage is to examine the degree to which the architecture constrains its extensions to follow REST-derived constraints. A more detailed discussion of these constraints follows.

REST Constraints

We earlier presented a summary of the domain properties, REST constraints, and induced behavior in Table 1 on page 4. In our subsequent analysis, we will specifically examine the behavior derived from these architectures to see how they deal with these REST constraints as listed in Table 2 on page 10.

TABLE 2. REST Architectural Constraints

Constraint	Assessing Degrees of Conformance to Constraint
Representation Metadata	How much control, for both requests and responses, does the architecture permit over representation metadata?
Extensible Methods	How much flexibility is offered to redefine or add methods within the architecture?
Resource/representation separation	How well does the architecture treat the divide between requests for a resource and resulting representations?
Internal Transformation	How conducive is the architecture to permitting representation transformations inside the system?
Proxy	How does the architecture enable the use of proxies and gateways?
Statefulness	How much control does the architecture provide to control statefulness?
Cacheability	In what ways does the architecture support cache components?

Architectural Characteristic Matrix

A matrix summarizing all of the architectural characteristics of these selected systems is presented as an addendum to this paper.

Origin Servers

In the REST world, Fielding defines an origin server as:

> An origin server uses a server connector to govern the namespace for a requested resource. It is the definitive source for representations of its resources and must be the ultimate recipient of any request that intends to modify the value of its resources. Each origin server provides a generic interface to its services as a resource hierarchy. The resource implementation details are hidden behind the interface.[Fielding 2000, #36, 5.2.3]

In common usage on the web, this is characterized by an HTTP server. Figure 4 on page 11 presents a timeline of market share as determined by Netcraft's Web Server Survey ([Netcraft 2005, #107]) for the three origin servers we will now discuss:

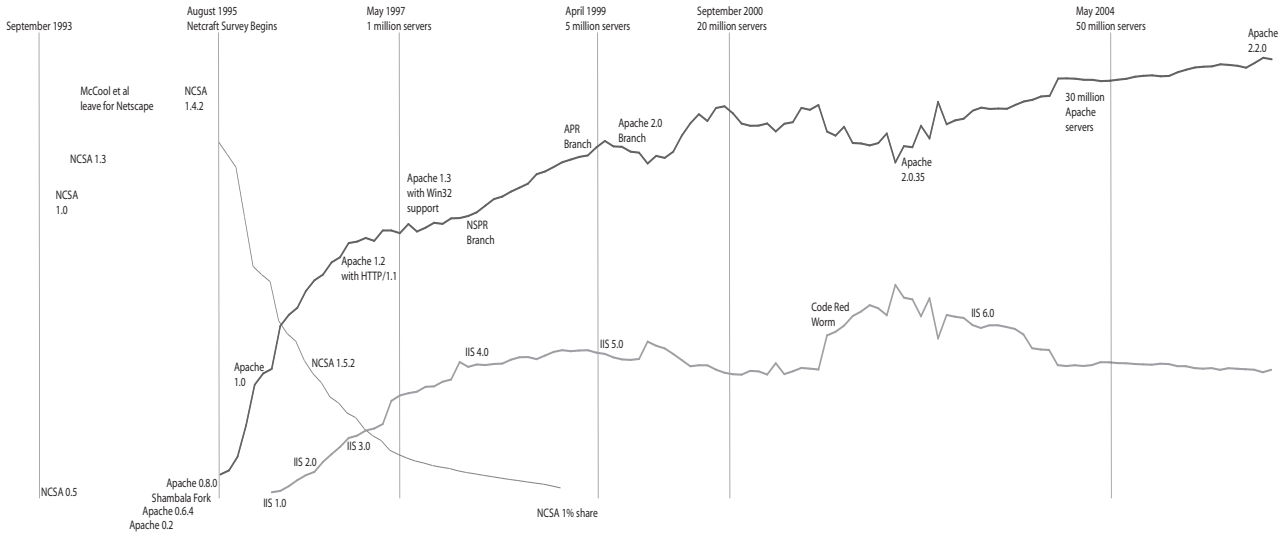

FIGURE 4. Origin Server Timeline with Netcraft Market Share Data from [Netcraft 2005, #107]

- NCSA HTTP Server
- Apache HTTP Server
- Microsoft Internet Information Services

NCSA HTTP SERVER

One of the early origin servers for the Web was produced at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign[Kwan, McGrath 1995, #64]. The name for this Web server was NCSA HTTPd (httpd) and it was released to the public domain for all to use at no cost. httpd was initially designed and developed by Rob McCool and others. After McCool left to join Netscape in 1994, NCSA development largely ceased with a few later ultimately unsuccessful efforts by NCSA to restart development around httpd.

NCSA ARCHITECTURE

Run-time architecture. The d in HTTPd refers to the Unix concept of a daemon. The word daemon has a long tradition in the Unix operating environment to mean a long-running process that assists the user. Therefore, HTTPd stands for "HTTP daemon" - meaning that the server responds to incoming HTTP traffic by generating the proper responses to the users without any direct intervention by the server administrator.

Upon initial execution, the httpd process would start listening for incoming HTTP traffic. As new HTTP requests arrived, this listening process would spawn two identical copies - in Unix parlance, the parent forked a child. One process (the parent process) would resume listening for more HTTP requests. The other instance (the child) would process the just-received incoming connection and generate the response. After that one response was served back to the client, this child process would close the socket and terminate.

This particular interaction model is particularly suited for a web server due to the repetitive nature of HTTP requests. Each resource on an HTTP server can be requested by

independent clients a large number of times and possibly in parallel. If the resource has not changed (retrieving a page is a read-only operation), then the representations served for each one of these requests should be identical as any state will be explicit in the request exchange. (HTTP labels methods having this behavior as *idempotent*.) This attribute allows one server process to independently handle any incoming requests without having to coordinate with other server instances.

However, due to the uneven nature of Web traffic, it does not make sense to dedicate one server instance to a particular 'part' of the server's namespace[Katz, Butler 1994, #58]. If the namespace were divided as such and a large burst of activity were to come in on one portion of the namespace, this could present significant bottlenecks - as that one process would be tied up serving all of the requests for that dedicated namespace. Therefore, httpd's run-time architecture allows all instances to respond to any part of the namespace independently. In addition to parallelizing on a single machine, this architecture also allows for replicated instances of httpd to work across multiple machines by the use of round-robin DNS entries and networked file systems[Katz, Butler 1994, #58, Kwan, McGrath 1995, #64].

Portability. httpd was written in the C language and the implementation was solely targeted towards Unix-derived platforms. Therefore, it had no intrinsic concept of portability outside of C and Unix systems. Yet, even Unix-derived platforms differ from each other greatly and httpd utilized language preprocessor macros for each flavor of the operating system that was explicitly supported. Additionally, the administrator had to hand-modify the build system in order to indicate which operating system httpd was being built on. Therefore, in comparison to modern-day servers, the portability of the original NCSA httpd server was quite restricted.

Internal Extensibility. While the code base behind httpd was relatively small, there was no clear mechanisms for extending the internal operations of the server. For example, most of the code relied upon global variables without any dedicated structures or objects. Therefore, if you wanted to support extensions to the protocol, there was no level of abstraction through which to effect these changes.

External Extensibility. Even without an internal extensibility layer, httpd did provide an effective external extensibility mechanism - the Common Gateway Interface (CGI)[Coar and Robinson 1999, #20]. The only other mechanism to produce a representation for a resource with httpd was to deliver static files off the file system. CGI was placed as an alternative to static files by allowing external dynamic execution of programs to produce output that a specific client will then receive. We will explore CGI more completely in "Common Gateway Interface (CGI)" on page 50.

With CGI, we begin to see a constraint of the external architecture peeking through: HTTP mandates synchronous responses. While the CGI program was processing the request to generate a response, the requestor would be 'on hold' until the script completes. During the exection of the script, NCSA warned that "the user will just be staring at their browser waiting for something to happen."[National Center for Supercomputing Applications 1995, #103] Therefore, CGI script authors were advised to hold the execu-

tion time of their scripts at a minimum so that it did not cause the user on the other end to wait too long for a response.

TABLE 3. REST Architectural Constraints: NCSA httpd

Constraint	Imposed Behavior
Representation Metadata	Explicit global values for each header value
Extensible Methods	Only through CGI's REQUEST_METHOD environment
Resource/Representation	No structure for requests or responses
Internal Transformation	None
Proxy	No, could not serve as a proxy
Statefulness	No explicit session management
Cacheability	No

Lessons Learned. Run-time architecture featuring parallel identical processes is well-suited for HTTP servers; extensibility focused on 'end user' extensibility instead of 'developer' extensibility; some characteristics of HTTP introduce very specific and otherwise awkward features to CGI.

APACHE HTTP SERVER

The Apache Project formed in February 1995 to resume active development of NCSA's popular but abandoned httpd. The goal of this new project was to incorporate bug fixes and new features. Besides important social innovations in distributed and open-source software development [Fielding 1999, #35, Mockus, Fielding 2000, #95], one of the keys to Apache's long-term success can be attributed to the sustained proliferation of third-party modules (now totalling over 300) around the core product. (This author is a contributor to the Apache HTTP Server.)

As shown in Figure 4 on page 11, The Apache HTTP Server is currently the most popular HTTP server used today. The various versions and derivatives of Apache collectively account for around 70% of the servers in use today [Netcraft 2005, #107], and has been the market leader for over nine years [The Apache Software Foundation 2004, #143]. The long-term mission of the Apache HTTP Server Project is to "provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards." [The Apache Software Foundation 2004, #142]

Due to its lineage from NCSA httpd codebase, there are a lot of surface similarities between the two codebases. In stark contrast to NCSA httpd however, the internals of the Apache HTTP Server are characterized by an extremely modularized design with almost all aspects of functionality available to be altered without modifying the core code. We will consider two snapshots of Apache's architecture: the 'initial' Apache architecture comprising all releases through the 1.3 series and the current release series (2.x and beyond).

INITIAL APACHE ARCHITECTURE

With the split of Apache from NCSA, there was a concerted change to make the internals of the new server much more extensible. Instead of relying upon custom ad hoc modifications to the codebase, the intention was to allow third-parties to add modules at build-time and run-time that modified Apache's behavior. These changes were balanced

by a strong effort to be end-user backwards compatible with NCSA httpd to ease the effort in migrating to Apache.

The principal mechanisms behind this rearchitecture were introduced in the “Shambala” fork by Robert S. Thau. These changes were merged into the mainline Apache codebase to become Apache 0.8.0 release in July 1995. These modifications formed the architectural basis of all future Apache releases. An exposition of the rationales for these decisions were put forth in a paper by Thau[Thau 1996, #137]. We will now summarize these rationales and their impact on the Apache architecture. A summary depiction of Apache’s architecture, as produced by The Apache Modeling Project, is presented in Figure 5 on page 14[Gröne, Knöpfel 2002, #46, Gröne, Knöpfel 2004, #47]. Hassan and Holt present another description of Apache’s architecture in Figure 6 on page 15[Hassan and Holt 2000, #52].

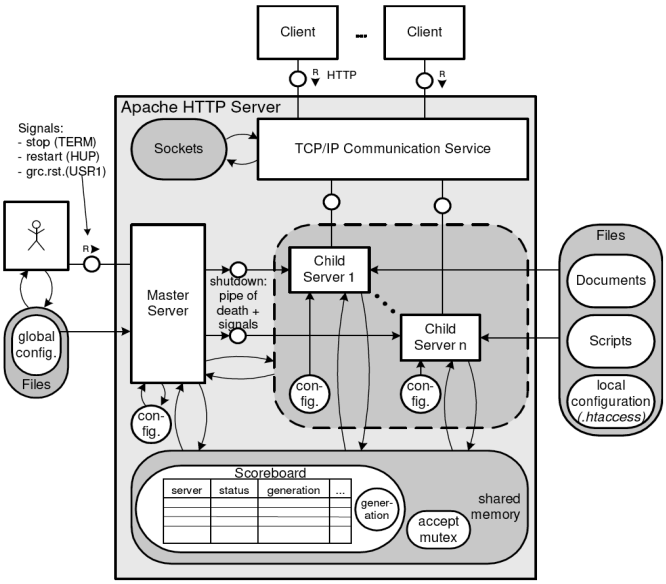

FIGURE 5. Apache HTTP Server Internal Architecture (From [Gröne, Knöpfel 2004, #47] Figure 4.6)

Run-time Architecture. As with NCSA httpd, early versions of Apache rely upon forking to handle incoming requests. However, Apache introduced the ability to reuse children via a “prefork” mechanism and to run these children at a low-privilege level. On Unix platforms, the cost of starting up a new process is relatively high. With NCSA httpd, every incoming connection would spawn a fresh process which caused a delay as the operating environment launched this new process. Instead, Apache starts (“preforks”) a configured number of children ahead of time and each process would take turns handling incoming requests. By having the servers initialized ahead of time, this allows a better response time and for spare capacity to be held in reserve. Any spare servers would be idle waiting for incoming requests and the process initialization costs can be amortized.

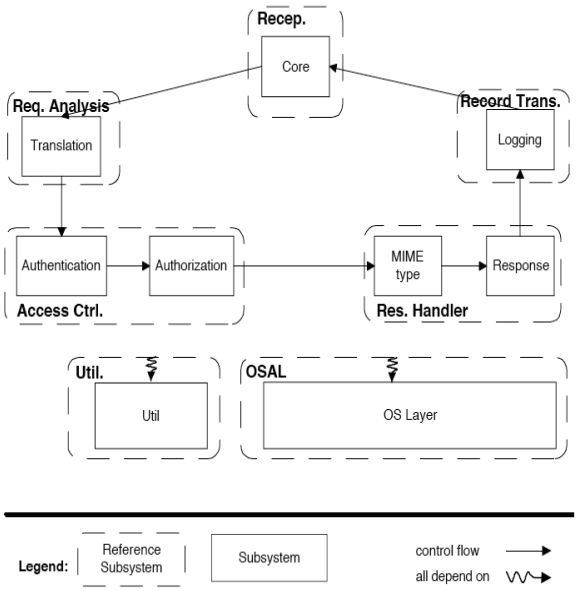

FIGURE 6. Apache HTTP Server Internal Architecture (From [Grosskurth and Godfrey 2005, #50])

In this preforking architecture, there is a parent process that keeps an eye on the children that are running. This parent is responsible for spawning or reaping children processes as needed. If all of the children are active and there is still space for new children, it will create a new child. On the other hand, if too many children are idle, it will remove some children from operation. While the parent process usually executes as a privileged user, it does not directly service any incoming requests from the users. Instead, the children that interact with clients are executed as an unprivileged user. This means that the attack surfaces for security attacks is minimized - however, there have been security exploits on certain operating systems that will elevate a unprivileged user to a privileged user.

Portability. The core implementation language of Apache is unchanged from httpd, so it is still written in C. While Unix is still the main target platform for Apache, later releases of Apache 1.x added support for Windows, OS/2, and Netware. While reaming in C, Shambala took advantage of some constraints enforced by the programming language and turned it into a substantial performance advantage.

One of the defining characteristics of C is that it requires explicit allocation (*malloc*) and deallocation of memory (*free*). These memory operations are rather expensive, so a pool-based allocation system was introduced in Shambala[Thau 1995, #136, The Apache Software Foundation 2003, #141]. This opportunity for efficiency is only available due to the well-defined lifecycle of HTTP traffic. In Apache, needed memory chunks are allocated from the operating system as part of normal operation during a request through *malloc* invocations. Normally, Apache would have to return all of the allocated memory back to the operating system through explicit invocations of *free*. If the each allocation was not explicitly freed, then memory leaks could occur. Over the

lifetime of a server process with constant traffic and memory leaks, this could eventually overload the memory capabilities of the system.

With the new pool system introduced in Shambala, when a response is completed, the allocated memory is instead added to a internal free-list maintained by Apache. On subsequent requests to the same process, the memory on the free-list can be reused instead of allocating more memory from the operating system. In practice, after a few requests are served, no more memory allocation is required from the operating system - previously allocated memory can suffice for subsequent runs. This pool model also has a large benefit for both internal and external developers. Since Apache tracks the memory itself, there is far less opportunity for memory leaks which impair the memory footprint of Apache. This can permit developers to not have to worry about detecting memory leaks in their modules as the pool system automatically tracks all allocations. In comparison to languages that offer intrinsic garbage collection (such as Java), there is no substantial performance penalty incurred for maintaining this list. Actually, in performance tests, this pool reuse system is a significant structural advantage of Apache that allows it to fare well against other HTTP servers[Gaudet 1997, #41].

Internal Extensibility. As discussed earlier, NCSA's architecture relied heavily upon CGI programs to produce content or alter the server's behavior. The CGI system suffers a severe drawback in that it is largely decoupled from the web server. This independence from the server comes at a steep cost as there is no clean mechanism to share configuration information between the web server and CGI application. This can create challenges for the content developer as their application becomes more complex by enforcing such a strict separation. Additionally, there are also performance implications with using CGI programs in that their process lifetime is only that of a particular request. Techniques like FastCGI can avert this performance issue by attempting to reuse a script interpreter across multiple connections[Brown 1996, #16]. However, this can introduce compatibility problems when global variables are used in CGI programs that are not correctly reset after each request.

Therefore, Apache specifically allowed for extensibility internally by exposing fixed points at which a third-party can interface in-process with the web server. Apache's initial extensibility phases, called *hooks*, included:

- URI to filename translation
- several phases involved with access control
- determining the MIME type of the resulting representation
- actually sending data back to the client
- logging the request

Each of these play a critical part in the functionality of the web server, but they can be logically independent. For example, the MIME type of a representation (which is content-specific) would not typically indicate a relationship as to how the server should log the request (which is usually server-specific). However, if there is a relationship, then a module can still hook into all needed phases and coordinate execution. Besides allowing dynamic behavioral modification through hooks, Apache has an internally extensible configuration syntax which allows dynamic registration of new commands with module-specific directives.

The drawbacks of this internal extensibility mechanism is that all of the modules run at the same privilege level and share the same address space. Consequently, there are no barriers preventing a malicious module from compromising the integrity of the system. A poorly written Apache module could expose a security vulnerability that could cause the server to crash. However, Apache's run-time architecture limits the effects of a bad module to only the specific process handling the request. Other children that are servicing a request are not affected if any other child dies through a software fault.

External Extensibility. Scripting languages such as PHP and JSPs are accomodated as *handlers* within Apache. These are specific modules that register for the handler hook and can deliver content for a specific resource. These handlers can be associated through content types or file extensions among other mechanisms. Therefore, in the case of PHP, its handler is responsible for converting the PHP script into usable HTML representations. The main advantage of having these scripts use a handler over a CGI mechanism is that there is no inter-process communication overhead required. Additionally, the scripting languages can take advantage of more Apache-specific features than what are available only through CGI.

REST Constraints. Apache represents an improvement over the NCSA httpd in constraining the extensions to follow the REST style. There is a clear separation between data and metadata with dedicated metadata structures. There is also less usage of global variables through dedicated request structures. However, Apache does not enforce a clear separation between the resource and representation as they share the same data structure (*request_rec*). A proxy component was added in later versions of Apache 1.3. However, these modules had significant implementation and design problems that resulted in its removal from later releases - limiting Apache's effectiveness as a proxy/gateway. As we will discuss in the following section, improving these modules was a factor behind some subsequent architectural changes.

TABLE 4. REST Architectural Constraints: Early Apache HTTP Server

Constraint	Imposed Behavior
Representation Metadata	Headers are in a hash-table structure; can be merged
Extensible Methods	Yes, through a dedicated request field
Resource/Representation	Response and request are coupled in the same structure
Internal Transformation	None
Proxy	Present in early versions of 1.3, but removed due to problems
Statefulness	No explicit session management
Cacheability	None

Lessons Learned. Optimizations created based on language-choice and domain-specific constraints; Run-time architecture modified to better suit the underlying platform; Modularity and internal extensibility heavily stressed through hooks and discrete separated dynamic modules; External extensibility through scripting languages; improvements in maintaining REST constraints.

APACHE 2.X ARCHITECTURE

The Apache HTTP Server 2.0 has redesigned the popular Apache HTTP Server by incorporating feedback from the development and user community. While remaining

faithful in spirit to the initial design of the 1.3 series server, the 2.0 series does break compatibility with the previous version in several areas.

Resolved design issues. Thau identified a number of design shortcomings of Apache in [Thau 1996, #137] - all of these issues have been resolved in Apache 2.x. The first issue raised is that Apache did not have a protocol API. The protocol code was refactored in 2.x and now has modules that implement FTP, SMTP, and NNTP in a clear and principled approach. Secondly, Thau indicated that it was hard to customize existing modules. This has been addressed by the introduction of the provider API, first introduced in mod_dav. Several other modules (such as authorization and caching modules) have since been broken down to use this provider API to easily alter their operation. Thirdly, Thau identified that the order dependencies of hooks were problematic. There is now a different hook registration system that allows explicit ordering of hooks (including predecessors and successors). Finally, Thau identified the lack of hooks that conform to system startup and teardown. These have now been added.

Portability and Run-time Architecture. 2.0 introduces a new portability layer called the Apache Portable Runtime that provides a "predictable and consistent interface to underlying platform-specific implementations."[The Apache Portable Runtime Project 2004, #140] The path to APR was, however, not a straight line. After the introduction of support for Netware, Windows, and even more Unix variants in Apache 1.3, a consensus emerged that a comprehensive portability strategy had to evolve to support more platforms in a cleaner way. There were initially two concurrent strategies: porting Apache to Mozilla's new runtime layer (NSPR) and the introduction of the Multi-Processing Modules (MPM).

These two approaches were noticably different: one (NSPR) would replace all of the platform specific code out of Apache and move all of it into a portability layer. A move to a new portability layer, such as NSPR, would necessitate a rewrite of the entire code base to use the primatives supported by the portability layer. In return, all of the concerns about supporting a new operating system would be off-loaded to the portability layer.

The other approach would isolate all of the "complicated" platform-specific code into a new policy layer within Apache - called the MPM. The rest of the code would rely upon standard ANSI C semantics. The MPM would specify the policy for handling the incoming connections: the default policy would be the *prefork* strategy initially introduced with Shambala and discussed earlier. Other policies would include a *worker* strategy that leveraged a hybrid process-thread approach, a *mpm_winnt* strategy that worked only on Windows platforms, and a *mpm_netware* module for Netware systems. The goal of the MPM design was that the process or thread management code these threads would be restricted to these policy modules. This containment was based on the belief that the difficult portability aspects could be constrained to the MPM modules alone.

These branches evolved in parallel until the group forced a decision over the adoption of the NSPR modules. The key argument against NSPR was not a technical one - but, rather, a social one - the developers did not agree with the licensing terms presented by NSPR. An attempt to resolve these concerns were inconclusive - therefore, the developers started their own portability layer based on the code that was already present in

Apache 1.3. This code formed the basis for the Apache Portable Runtime (APR) layer first present in 2.0. However, the MPM components were also ultimately integrated into the new APR branch. Therefore, even though the strategies seemed at odds initially, both strategies were eventually merged. The code was rewritten on top of a new portability layer and a policy layer was introduced to abstract the process management code. Through the MPM system, a number of strategies have been experimented with - including a policy that supports asynchronous writes introduced in the recent 2.2 release[The Apache HTTP Server Project 2005, #138].

Internal Extensibility. A recurring issue that was raised by developers throughout the 1.3 series was that it was hard to layer and combine functionality between modules. If a developer wanted to extend how REST representations are generated in the Apache handlers, code had to be duplicated between modules. Therefore, a major advance in the 2.0 release was the addition of a layering system for data to allow principled composition of features and resource representation transformations (e.g., on-the-fly compression and dynamic page generation). Compatibly integrating this system while maintaining as much backward compatibility as possible was a key development challenge.

Another issue with the 2.0 series was the evolution of the mod_proxy code, which allows a standard Apache httpd server instance to act as a proxy. Since Apache's original design intended it to act as an HTTP server, the prevailing design assumptions throughout the code is that the system is an HTTP server not an HTTP client. However, when a proxy requests a page from an upstream origin server, it acts as a client in the REST architecture. The concept of input and output from the architecture perspective became switched with a proxy. This presented a number of mismatches between mod_proxy and the rest of the httpd architecture that required design compromises to compensate.

Integration. A set of extensions to the Apache HTTP Server allow the core server functionality to be integrated into a larger and different architecture. Through modules such as *mod_perl* and *mod_python*, new applications around the core Apache HTTP Server architecture can be constructed. For example, the Perl-based qpsmtpd SMTP mail server can leverage the features of Apache through *mod_perl*[Sergeant 2005, #123]. This arrangement offloads all of the connection management and network socket code from Perl to the httpd's C core, but any extensions to qpsmtpd can be maintained in Perl.

TABLE 5. REST Architectural Constraints: Apache HTTP Server

Constraint	Imposed Behavior
Representation Metadata	Headers are in a hash-table structure; can be merged
Extensible Methods	Yes, through dedicated request field
Resource/Representation	Response and request are coupled in the same structure
Internal Transformation	2.0 adds filter support; 2.2 permits more complicated chains
Proxy	Can serve as a proxy in 2.0; load-balancing support in 2.2
Statefulness	No explicit session management
Cacheability	Production-quality in 2.2 release

Lessons learned. Portability concerns led to a new portability layer and new run-time architecture policy layer; the absence of an internally extensible RESTful representation

required the shoe-horning of filters; early design assumptions of where a node fits in the overall REST architecture challenged as the system evolves; use in different server-based applications.

MICROSOFT INTERNET INFORMATION SERVER (IIS)

Microsoft first released their HTTP server named Internet Information Server (IIS) in February, 1996. This initial version of IIS was only available on Windows NT 3.51. Over time, it was updated to work on newer releases of the Windows platform. The early releases of IIS featured basic HTTP and FTP serving support. Over time, more features and extensibility models were added. At the same time, however, many security vulnerabilities were exposed in IIS servers. This led to a number of prevalent worms, such as Code Red, on the Internet that spread through the vulnerabilities in IIS[Cook 2005, #21, Moore, Shannon 2002, #96].

IIS 6.0, first included with Windows 2003 Server, was the beginning of a security-centric architectural rewrite for Microsoft's server products. At this point, Microsoft also renamed IIS to stand for "Internet Information Services." After numerous security vulnerabilities had to be fixed, Microsoft engineered a number of modifications to the IIS architecture with an eye towards security. Besides being no longer installed by default, IIS 6.0 offers a number of features focused on forcing administrator to make security-conscious decisions about their server.

Portability. From the outset, IIS was only intended to operate on Microsoft's Windows platforms. Therefore, it can take extreme advantage of Windows-specific functionality that are only available on that platform. However, this means that portability to other operating systems is not feasible with the IIS architecture. One distinction that is challenged with IIS 6.0 is the separation between the kernel mode and user mode in the operating system.

A new kernel-mode driver called *http.sys*, running at the highest privileges inside the Windows kernel, was introduced that takes over a portion of the HTTP functionality from the traditional user-mode applications[Microsoft Corporation #71, Wang 2005, #152]. The goal of this new driver was to "to increase Web server throughput and scalability of multiprocessor computers, thereby significantly increasing the following: the number of sites a single IIS 6.0 server can host; the number of concurrently-active worker processes."[Microsoft Corporation #72]

Run-time architecture. IIS presents the administrator with two run-time architectural models to chose from. Depicted in Figure 7 on page 21 is the *IIS 5.0 isolation mode* architectural model. This legacy model is targeted towards "applications developed for older versions of IIS that are determined to be incompatible with worker process isolation mode." [Microsoft Corporation #73] The downfall of this architectural model is that all instances share the same process - one fault could jeopardize the reliability of the server. This architectural fault led to numerous reliability problems[Peiris 2003, #116, Web Host Industry Review 2001, #153].

To increase reliability, IIS 6.0 introduces a new run-time architectural option called *Worker Process Isolation Mode*, depicted in Figure 8 on page 21. This model defines a collection of *application pools* that are assigned to a specific web site - a fault in one website will only jeopardize the application pool it resides in.[Microsoft Corporation

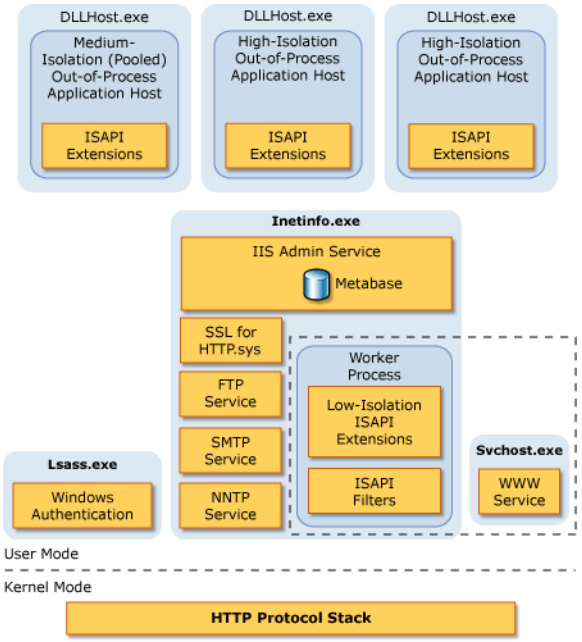

FIGURE 7. Internet Information Services (IIS5 Compatibility Mode) (From [Microsoft Corporation #74])

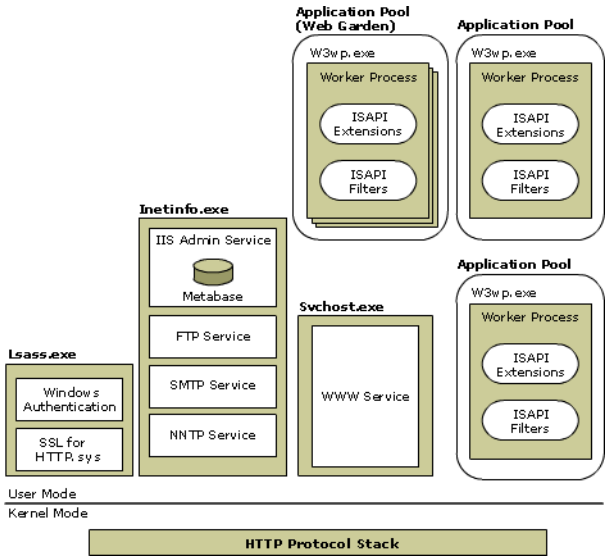

FIGURE 8. Internet Information Services 6.0 Architecture (From [Microsoft Corporation #74])

#75] These pools register with the kernel-mode HTTP driver for a particular namespace and incoming requests for that namespace is then forwarded to the appropriate user-space process to generate a response.[Smith 2004, #127]

Internal Extensibility. ISAPI is the code name given to Microsoft's Internet Server API specification, which debuted with the initial release of IIS. Microsoft claims that they initially positioned ISAPI to compete with CGI - however it differs substantially from CGI. ISAP modules would be executed inside the server process not outside the server process like CGI[Schmidt 1996, #122]. ISAPI modules are required to be compiled as Windows DLLs and explicitly inserted into the server configuration. Therefore, even with Microsoft's initial characterization as ISAPI as a competitor to CGI, we will characterize ISAPI as an internal extensibility mechanism instead of an external extensibility mechanism.

ISAPI offers two dimensions of access: extensions and filters. ISAPI extensions must be explicitly registered for a configured URI namespace[Microsoft Corporation #76]. For convenience, specific file types can also be associated with an ISAPI extension - files bearing an .asp extension can be mapped to the ASP.dll extension. In this manner, extensions are like CGI applications as they create a virtual namespace under its own control; however, extensions offers far more control while introducing more security risks than CGI applications. As we will discuss in the following section, ISAPI extensions presented significant source disclosure risks.

ISAPI filters, instead of being explicitly requested, are explicitly configured for a specific site. A filter is set up for a specific virtual host and is then executed on every request for that virtual host. The filter can then transform the incoming and outbound data before it is processed by other filters or extensions. In addition, filters can perform a number of other tasks, including:[Microsoft Corporation #77]

- Control which physical file gets mapped to the URL
- Control the user name and password used with anonymous or basic authentication
- Run processing when a connection with the client is closed
- Perform special logging or traffic analysis
- Perform custom authentication

While the range of functionality offered through filters is similar to that offered by Apache HTTP Server's hooks, Microsoft recommends that "the work performed by ISAPI filters [be] minimized"[Microsoft Corporation #78]. This is because every filter is executed on each request which can introduce substantial invocation overhead if it is not needed on every request.

Also, as part of the new *Worker Process Isolation mode* in IIS 6.0, ISAPI Extensions and Filters are now relegated to the individual process space of the specific application pool. Since errors in the kernel-mode driver can cause stability problems, it is protected from any external modifications. Yet, this directly constrains what operations can be performed by the ISAPI filters. Previously, raw data filters had the ability to access the underlying connection stream to introduce modifications into the data stream. With the new kernel-mode code handling the brunt of the protocol interactions, all of this is required to be handled by the http.sys driver directly. Therefore, any applications that

require raw data access must use the lower-security *IIS 5.0 isolation mode* and bypass the HTTP kernel driver.

External Extensibility. IIS 3.0 introduced Active Server Pages (ASP) and is classified as a server-side scripting language. As discussed before, the implementation of ASPs in IIS are handled by an ISAPI extension. Numerous security issues discovered with IIS over the years permitted the source code of these ASPs to be disclosed through bypassing these extensions. This presented a number of security risks as sensitive information (such as database usernames and passwords) were often stored inside the ASP files under the assumption that the client would never see the source behind these ASP files. Eventually, most ASP content developers began to understand that various vulnerabilities would occur which would disclose the source of their files and consequently limited the amount of sensitive information in the ASP files themselves.

While IIS 6.0 retains support for ASPs, CGIs, WebDAV, and other server-side technologies, they must now be explicitly enabled by the site administrator. Only static content will be served by default. This action is now required "to help minimize the attack surface of the server."[Microsoft Corporation #79] Any requests to these inactive services, even if they are otherwise installed, will result in an HTTP error being returned to the user.

TABLE 6. REST Architectural Constraints: IIS

Constraint	Imposed Behavior
Representation Metadata	Request: Fetch request header with ':' Response: Add headers with manual delimiting[Microsoft Corporation 2004, #90]
Extensible Methods	HTTP processing by the kernel prevents this with IIS 6.0
Resource/Representation	Extensions Response object not clearly defined
Internal Transformation	Filters are defined for an entire site IIS 6.0 further reduces filter flexibility for security
Proxy	None
Statefulness	ASP session information hides state from ISAPI modules
Cacheability	Added in IIS 6.0

Lessons Learned. Lack of separation in run-time architecture presented serious security risks; Installing and activating unnecessary components by default can be dangerous; Security constraints can restrict range of functionality that can be provided.

User Agents

Fielding defines a user agent in the REST world as:

> A user agent uses a client connector to initiate a request and becomes the ultimate recipient of the response. The most common example is a Web browser, which provides access to information services and renders service responses according to the application needs.[Fielding 2000, #36, 5.2.3]

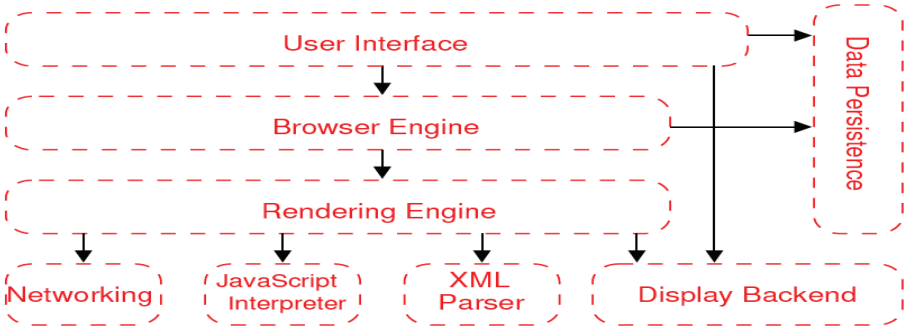

FIGURE 9. Web Browser Reference Architecture (From [Grosskurth and Godfrey 2005, #50])

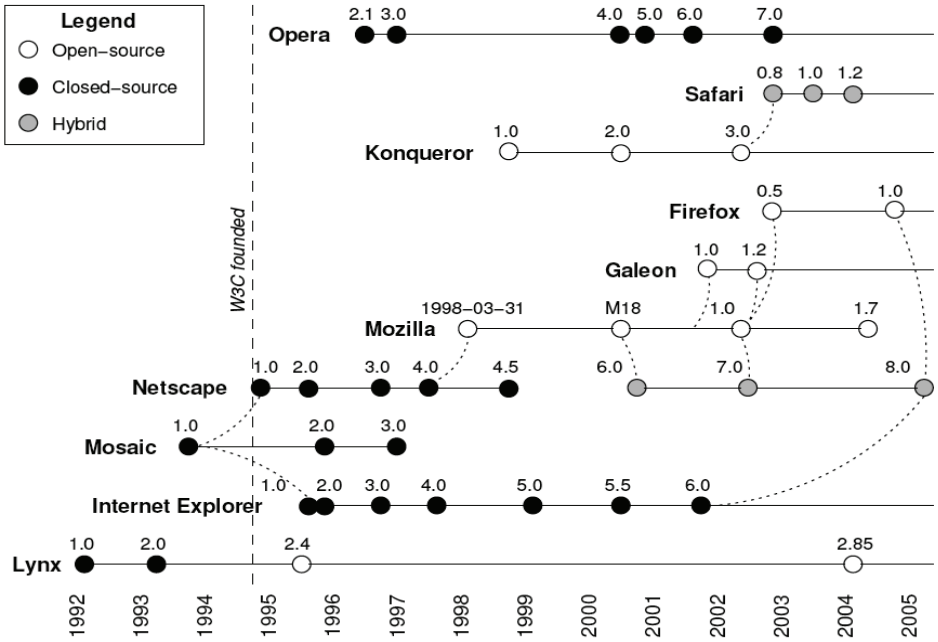

FIGURE 10. Web Browser Timeline (From [Grosskurth and Godfrey 2005, #50])

Grosskurth and Godfrey used automated architectural recovery processes to define a reference architecture for web browsers depicted as Figure 9 on page 24[Grosskurth and Godfrey 2005, #50]. Besides producing a reference architecture, they also presented a timeline that covers the early history of web browsers as depicted in Figure 10 on page 24. In an earlier work, Grosskurth and Echihabi extracted software architectures for several other web browsers[Grosskurth and Echihabi 2004, #49]. For the web browsers that they extracted an architecture for and are discussed here, we will present their architecture diagrams as well. However, it should be noted that these architectural diagrams are relatively high-level and tell us little about the behavior of the system along the prism dimensions we introduced earlier.

Mosaic And Descendants

The migration from text-based browsers to graphical browsers allowed the content on the World Wide Web to evolve from *hypertext* to *hypermedia*. One of the earliest successful graphical web browsers was NCSA Mosaic which started development in 1993[National Center for Supercomputing Applications 2002, #106]. Like httpd, Mosaic was developed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. The ability to render images, and hence, richer documents, over the World Wide Web was critical in facilitating an explosive growth in web traffic[Kwan, McGrath 1995, #64].

Mosaic Architecture

Compared to web browsers today, Mosaic was highly restricted as to what content it could render internally. Mosaic was initially restricted to supporting only the HyperText Markup Language (HTML), the Graphics Interchange Format (GIF) format, and the XPixMap image format (XPM).

Portability. There were several versions of Mosaic that were distributed by NCSA: ones for Windows, Macintosh and the various Unix platforms. While they shared the same brand name, they did not share a common architecture. Each one was written independently and had separate release cycles. By far the most popular version during this time was Mosaic for X Windows (X Windows being the standard windowing system on Unix platforms). Therefore, we will restrict the following architecture discussions to Mosaic for X Windows.

Run-time architecture. Since Mosaic was an X Windows program, its architecture followed the constraints of X: a single process generated and responded to user events. New windows could be created which would request a URL and render the response within the window. Mosaic did support other protocols, including FTP and Gopher, that could be accessed through the URL syntax.

External Extensibility. In each HTTP response, there is usually an associated metadata field called *Content-type*. The presence of this metadata allowed programs such as Mosaic to determine how to best render the received representation. If Mosaic could render the content-type natively (such as for HTML, GIF, or XPM), it would display the content inside of the browser window. However, if it was not one of the types that it supported, Mosaic relied upon *helper applications* to render the document.

However, this setup had a significant drawback: these helper applications were truly external to Mosaic[Schatz and Hardin 1994, #121]. Mosaic would download the representation locally to disk and pass the local file information to the specified application. The viewer would then execute independently and render the content separately in its own window space. Thus, a critical aspect of hypermedia was lost: all navigation capabilities stopped once the helper application was launched.

Internal Extensibility and Integration. To help address the loss of navigation through unknown media types, an experimental Common Client Interface (CCI) was first released with Mosaic 2.5 for X Windows in late 1995[National Center for Supercomputing Applications 1995, #102, Schatz and Hardin 1994, #121]. By this time, most of the original Mosaic development team had left to start Netscape, which among other competitors, started their own internal extensibility efforts. There were also serious inherent security problems with CCI - any incoming connection to the CCI TCP port would

allow the user to control the browser without any authentication. Therefore, there was little practical adoption of CCI.

CCI allowed for external applications to send instructions to Mosaic: such as navigating to a particular page. One such CCI-enabled application was X Web Teach which allowed a teacher to browse websites with student Mosaic instances automatically navigating to the same sites.[Braverman 1994, #15] Other work with CCI would allow control of the user interface within Mosaic[Newmarch 1995, #110]. Notably, CCI did not allow for drawing of unknown media types within the Mosaic windows.

TABLE 7. REST Architectural Constraints: Mosaic

Constraint	Imposed Behavior
Representation Metadata	None
Extensible Methods	No
Resource/Representation	Content went straight from parser to user's window
Internal Transformation	External viewers only
Proxy	Can pass requests to a proxy
Statefulness	No state management issues
Cacheability	Partially: some features didn't work 'right' with the cache

Lessons Learned. Internal rendering of types facilitate hypermedia, but the lack of tight integration for unknown media types present a difficulty in persisting the hypermedia experience.

EARLY NETSCAPE NAVIGATOR ARCHITECTURE

Even though little code was shared between NCSA Mosaic and the new Netscape Navigator, the key designers behind the two browsers were constant. Similar to what transpired with NCSA httpd and Apache HTTP Server, a large percentage of the user base quickly migrated from the depleted NCSA project to a viable comptetitor - in this case, Netscape Navigator. There was, however, one notable difference between the server administrator's transition to Apache: Netscape, unlike NCSA Mosaic, was only available for a fee. However, until its competitors became viable alternatives and undercut its prices by giving their browsers away, Netscape had acquired over an 80% market share by the summer of 1995[Wilson 2003, #155] It should also be pointed out that the internal codename for Netscape Navigator was Mozilla - which would ultimately resurface later.

Given a chance to re-examine past architectural decisions based on their Mosaic design experiences, the team decided to address several issues that were unresolved with NCSA Mosaic. Among the key architectural changes were the introduction of more current HTML support, Java applets, JavaScript, Cookies, and the introduction of a client-side plug-in system to internally incorporate the concept of helper applications.

Portability. Like NCSA Mosaic, Navigator was written in C with versions of Navigator available for Unix, Macintosh, Windows, and other platforms. During this time, Java emerged on the scene with its "write once, run anywhere" promises. Navigator was one of the first non-Sun browsers to incorporate applet support - which allowed Java applications to run inside of the browser[Gosling and Yellin 1996, #44].

As an object-oriented language with integrated memory management and the promises of pure portability, the Netscape developers initially found Java an attractive language.[Zawinski 2000, #158] Therefore, Netscape started the process of rewriting Navigator in Java under the codename of "Xena" (the press labeled this effort "Javigator"). Jamie Zawinski, one of the lead Navigator developers, has commented, "I think C is a pretty crummy language. I would like to write the same kinds of programs in a better language."[Zawinski 2000, #158] Unsurprisingly, however, the promises and reality of Java were far apart: the portability, efficiency, and confusing mix of concepts caused serious problems. Zawinski eventually concluded, "I'm back to hacking in C, since it's the still only way to ship portable programs."[Zawinski 2000, #158] Given these technical problems, Netscape management later cancelled the Java porting effort and only released incomplete portions of the Mail client under the code name Grendel[Zawinski 1998, #157].

Internal Extensibility. One of the significant advances introduced with Navigator was the addition of a plug-in architecture. Developers could now write dynamically-loaded plugins to handle specific content-types and render them inside of the browser - instead of requiring an external application. For example, a user who wanted to view a QuickTime movie inside of their browser only needed to install a QuickTime plug-in for Netscape. Additionally, if the content type being viewed supported links (such as a movie trailer pointing to a website for more information), the plug-ins could further the hypermedia context by directing the browser to fetch that URL. True two-way interaction between the browser and its plugins was achieved.

Plug-ins inside of Netscape can perform the following tasks[Oeschger 2002, #113]:

- draw into a part of a browser window
- receive keyboard and mouse events
- obtain data from the network using URLs
- add hyperlinks or hotspots that link to new URLs
- draw into sections on an HTML page
- communicate with JavaScript/DOM from native code

These plug-ins would be compiled into native code by the developer and distributed to the users for installation. To maintain backwards compatibility and promote their own adoption rates, most current web browsers still support loading these original Netscape plug-ins.

External Extensibility. Netscape also introduced a number of ways for content designers to influence the behavior of the browser above what could be achieved with simple HTML. The first of these was the introduction of JavaScript[Netscape 1996, #108]. JavaScript is a client-side scripting language that allows content developers, through special HTML tags, to control the behavior of the browser when viewing that specific HTML page. We will discuss JavaScript more completely in "JavaScript" on page 52.

The other key feature that Netscape Navigator introduced was cookies.[Kristol and Montulli 1997, #63, Netscape 1999, #109] As discussed previously in "REST Mismatches in HTTP Extensions" on page 6, cookies are one of the most pervasive examples of non-RESTfulness on the Web. Cookies allow a server to provide an opaque

token to the client as a meta-data field. The client can then save this cookie and then present that same opaque token to the same server in any subsequent requests. Since the server issued the "cookie" in the first place, it can then determine the client that is making the request. Numerous security implications have been discovered through the improper use of cookies, but their usage still remains prevalent today.

TABLE 8. REST Architectural Constraints: Early Netscape

Constraint	Imposed Behavior
Representation Metadata	Requests: Meta-data represented as content[Oeschger 2002, #113] Responses: Content-type is the vector for determining viewer
Extensible Methods	Only POST and GET methods were supported
Resource/Representation	Plug-ins could transform based on representation type
Internal Transformation	Content could dynamically change through plug-ins
Proxy	Can pass requests to a proxy
Statefulness	Introduction of Cookies conflicts with REST
Cacheability	Yes

Lessons Learned. Attempts at porting the web browser to Java failed; Internal extensibility greatly enhanced through client-side plugins; external extensibility enhanced with introduction of JavaScript; Statefulness REST constraints violated with the introduction of cookies.

NETSCAPE 6.0 / MOZILLA ARCHITECTURE

After the success of Netscape 4 and the failure of their Java rewrite, the developers behind Netscape decided to rewrite the codebase from scratch. This caused the release of Netscape 6.0 to be delayed until April 2000. One commentator criticized this decision[Spolsky 2000, #128]:

> Netscape 6.0 is finally going into its first public beta. There never was a version 5.0. The last major release, version 4.0, was released almost three years ago. Three years is an awfully long time in the Internet world. During this time, Netscape sat by, helplessly, as their market share plummeted.
> It's a bit smarmy of me to criticize them for waiting so long between releases. They didn't do it on purpose, now, did they?
> They did it by making the single worst strategic mistake that any software company can make:
> **They decided to rewrite the code from scratch.**

This period was one of large social turmoil for the project as Netscape was purchased by America Online and the Mozilla Foundation was started[Markham 2005, #68]. The codebase that eventually formed the basis of Netscape 6.0 was first open-sourced under the Mozilla moniker in 1998. Since the opening of the Mozilla codebase, all subsequent Netscape releases were derived to some degree from the Mozilla codebase.

Architecture Recovery Process. As part of the TAXFORM project, Godfrey and Lee reconstructed the software architecture behind Mozilla Milestone 9 through an automated architectural recovery process[Godfrey and Lee 2000, #42]. Milestone 9 was first released to the public in August 26, 1999 and represented a web browser, mail cli-

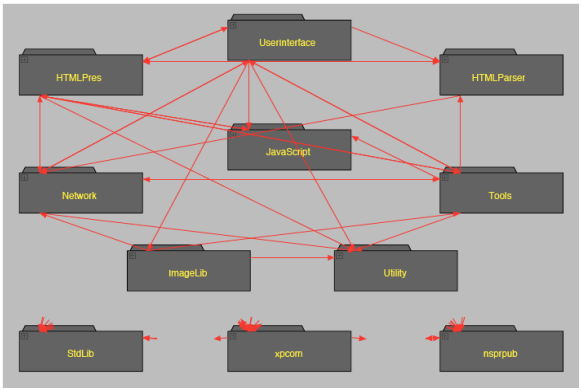

FIGURE 11. Mozilla Milestone 9 Architecture (From [Godfrey and Lee 2000, #42])

TABLE 9. Mozilla Architecture Breakdown (From [Godfrey and Lee 2000, #42])

Name	Description	Associated Subsystems	Associated Files
HTMLPres	HTML layout engine	47	1401
HTMLParser	HTML Parser	8	93
ImageLib	Image processing library	5	48
JavaScript	JavaScript engine	4	134
Network	Networking code	13	142
StdLib	System include files (i.e., ".h" files)	12	250
Tools	Major subtools (e.g., mail and news readers)	47	791
UserInterface	User interface code (widgets, etc.)	32	378
Utility	Programming utilities (e.g., string libraries)	4	60
nsprpub	Platform independent layer	5	123
xpcom	Cross platform COM-like interface	23	224

ent, news reader, and chat engine in one integrated application[Mozilla Foundation 2002, #100]. Mozilla's aim with Milestone 9 was to introduce a new networking layer called Necko[Mozilla Foundation 2001, #99]. The Mozilla developers justified Necko because "Mozilla's current networking library is in a sorry state" and the old layer "was designed for a radically different non-threaded world."[Harris and Potts 1999, #51]

Godfrey and Lee's recovered architecture diagram for Mozilla Milestone 9 is presented in Figure 11 on page 29. A breakdown of Mozilla's architectural systems is presented Table 9 on page 29. While Godfrey and Lee also provided the number of lines of code for each architectural division, we exclude that number here.

Portability. At this point in time, the complete Mozilla codebase consisted of over 7,400 source files and over two million lines of code in a combination of C and C++[Godfrey and Lee 2000, #42]. To place the cost of portability in perspective, Godfrey determined that only 20% of the C files and 60% of the C++ files were actually required to operate on the Linux operating system. To ease the difficulties associated

with the recovery process, Godfrey therefore eliminated the code that was not required on Linux. Therefore, their analysis did not consider how 80% of the C code or 40% of the C++ code fit into rest of the overall architecture. We believe that removing these codes understated the true impact of portability for Mozilla.

Analysis of Mozilla's Architecture. Godfrey summarized their architectural observations about Mozilla that "either its architecture has decayed significantly in its relatively short lifetime, or it was not carefully architected in the first place"[Godfrey and Lee 2000, #42]. As can be seen in the figure, the recovered architecture indicates a "near-complete graph in terms of the dependencies between the different [Mozilla] subsystems." Their recovered architecture indicated a dependency upon the network layer by the image processing layer. They concluded that "the architectural coherence of Mozilla to be significantly worse than that of other large open source systems whose software architectures we have examined in detail."

It is compelling to compare this rather harsh architectural assessment with that of Brendan Eich, one of the Netscape developers and co-founders of the Mozilla project, who remarked in November, 2005[Eich 2005, #24]:

> Some paths were long, for instance the reset in late 1998 around Gecko, XPCOM, and ... XPFE. This was a mistake in commercial software terms, but it was inevitable given Netscape management politics of the time, and more to the point, it was necessary for Mozilla's long-term success. By resetting around Gecko, we opened up vast new territory in the codebase and the project for newcomers to homestead.

Godfrey and Lee used an underlying codebase for the architectural recovery which included these precise modifications that Eich credits for Mozilla's "long-term success." Therefore, we must question either the validity of the developer's informal assessment or the faithfulness of reconstructed architecture. This leads to an interesting line of questioning: Were these changes really in the codebase and detectable by the fact extractors? If they were present, did they have any measurable architectural impact at this point in time? If the architectural coherence is "significantly worse" than comparable systems, what does this indicate for the future? Given the conflicting nature of the architectural assessments, it is imperative to continue to trace the evolution of the Mozilla codebase with an eye towards its architecture.

Lessons Learned. Need for complete architectural rewrite due to decay allowed competitors to overtake it in the market; challenges in understanding recovered software architecture in context of evolving systems.

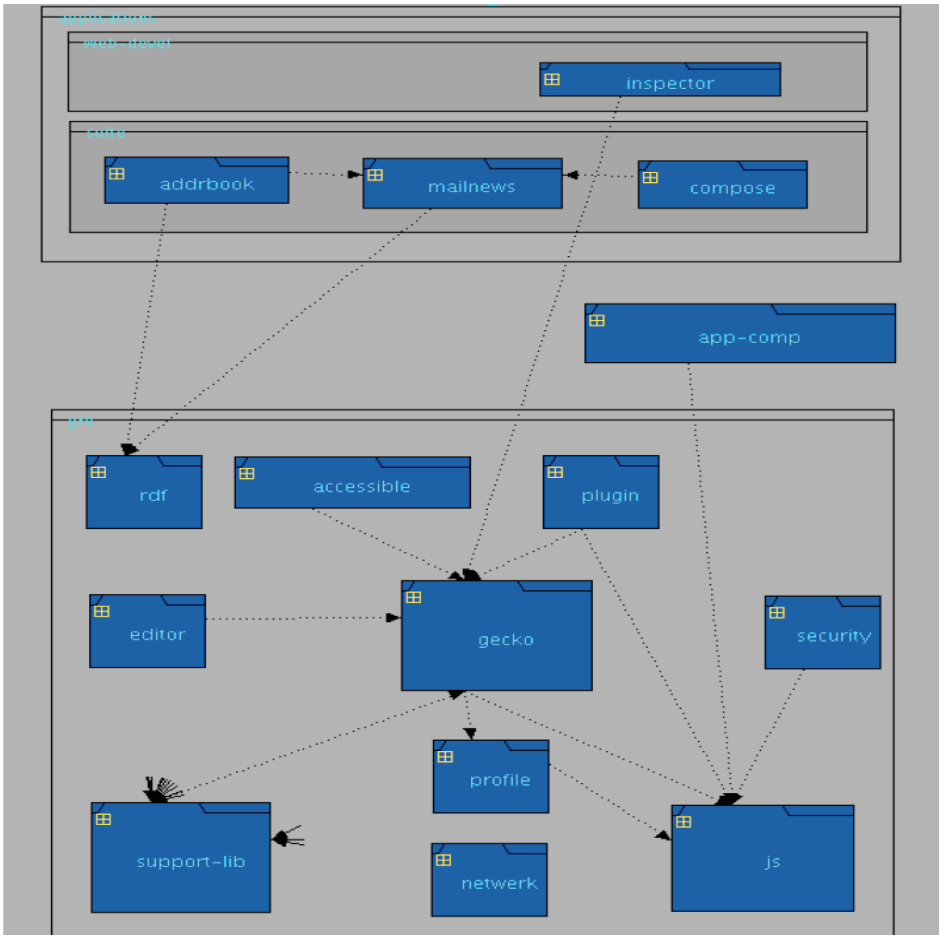

FIGURE 12. Mozilla Concrete Architecture - circa 2004 (From [Grosskurth and Echihabi 2004, #48])

Current Mozilla Architecture

In 2004, Grosskurth and Echihabi returned to Mozilla to assess the architecture's progress[Grosskurth and Echihabi 2004, #48]. By this time, Mozilla had launched a spinoff product called Firefox. Firefox differed from Mozilla in that only delivered web browsing functionality without any mail-reading functionality. Most of the resulting discussion of the current Mozilla architecture applies to Firefox as well.

Grosskurth and Echihabi's resulting concrete architecture is presented in Figure 12 on page 31. They also fit this recovered Mozilla architecture into a reference architecture for web browsers as presented in Figure 13 on page 32. Finally, we present an architecture diagram created by a manual process and was presented as part of a book on developing applications with Mozilla in Figure 14 on page 32[McFarlane 2003, #69].

Grosskurth's architectural recovery techniques were similar in nature to the analysis conducted by Godfrey and Lee in 2000. Therefore, the two sets of resulting architectures can be compared with relative ease. Even though the code size did not increase

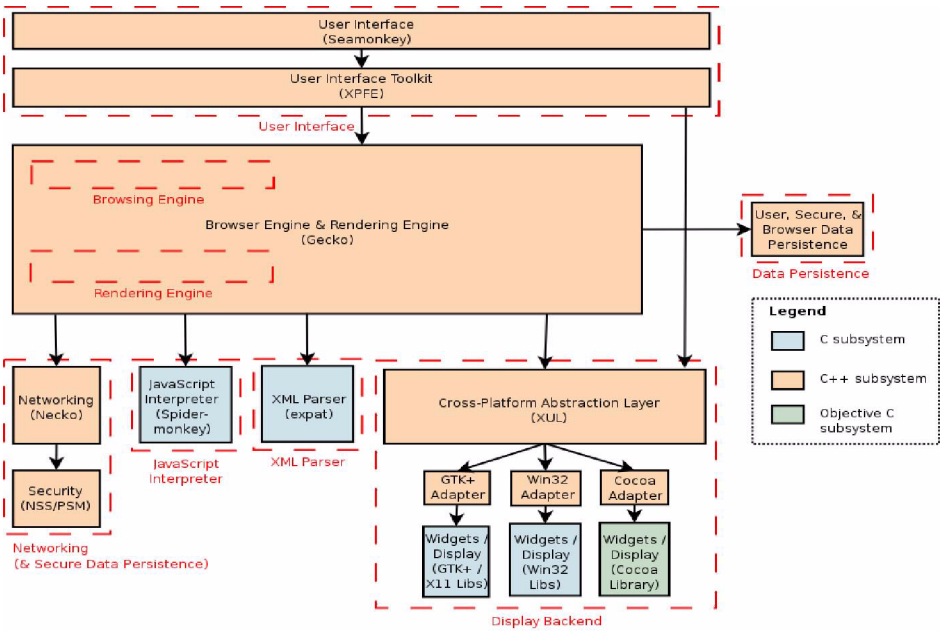

FIGURE 13. Mozilla Architecture (From [Grosskurth and Echihabi 2004, #49] Figure 8)

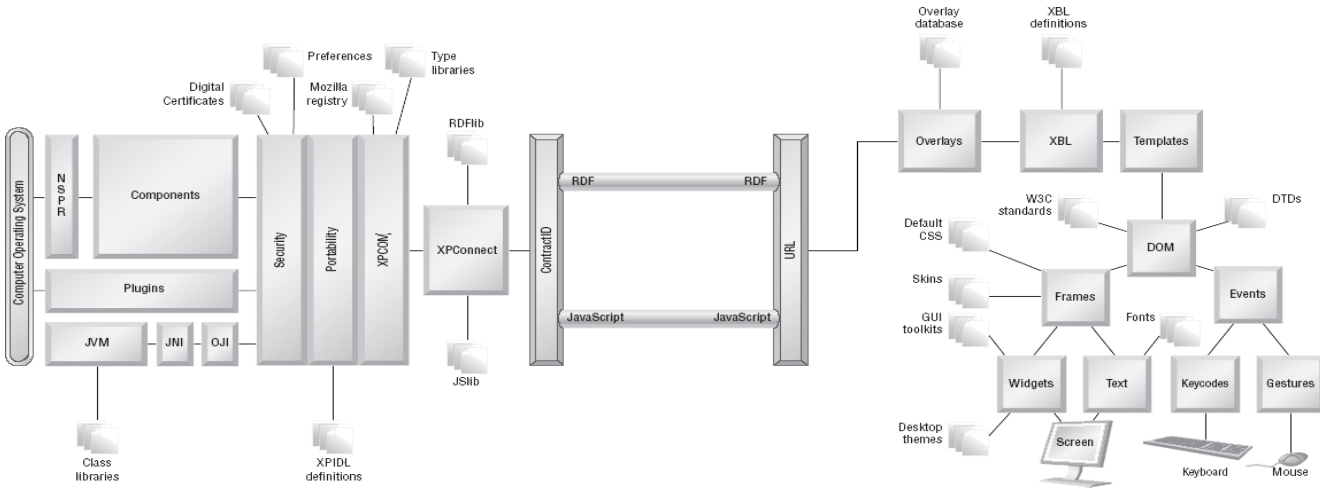

FIGURE 14. Mozilla Architecture (From [McFarlane 2003, #69])

substantially, the striking difference between the two architectural snapshots is that the complex graph of intra-module dependencies has now been eliminated. The cyclical dependencies that caused Godfrey and Lee to label Mozilla as an exemplar of architectural decay is no longer.

Grosskurth and Echihabi also termed the Mozilla architecture as a modified pipe-and-filter system. However, we believe that this is an over-simplistic classification of Mozilla's architecture. Features of Mozilla's architecture, specifically the networking layer, do indeed exhibit the characteristics of a pipe-and-filter system[Saksena 2001, #120]. However, the higher-level portions of Mozilla, which include the renderer and interfaces, have characteristics closer to an event-driven architecture than a pipe-and-filter architecture.[Larsson 1999, #65]

Internal Extensibility. One of the defining characteristics of this new architectural model is the breakdown of components via Cross Platform Component Object Model (XPCOM)[Turner and Oeschger 2003, #148]. While XPCOM's design is inspired by Microsoft's COM system, XPCOM only operates within the Mozilla architecture rather than across an operating system[Parrish 2001, #115]. Building upon XPCOM with user interface extensions like XPFE[Trudelle 1999, #147], Mozilla now offers third-parties the ability to customize all facets of the system dynamically. This has created a wealth of third-party extensions that modify Mozilla's behavior in a variety of mechanisms. New protocols can also be added through XPCOM[Rosenberg 2004, #119].

Even with this new model, Mozilla still supports Netscape plug-ins. Yet, there is also a hidden cost for backwards-compatibility inside the Mozilla architecture for this support. The previous Unix-based plug-ins were based on the Motif Xt library, while new plug-ins are built on top of the GTK+ library[Grosskurth and Echihabi 2004, #48]. Therefore, run-time emulation is performed with these legacy modules by dynamically translating Motif calls to GTK+.

Portability. Like all of its architectural predecessors, Mozilla is still written in C and C++. However, JavaScript has been introduced as a critical part of Mozilla: almost all Mozilla extensions can now be written in JavaScript via XPCOM[Bandhauer 1999, #10]. Therefore, extension developers no longer need to write their extensions in C, but instead can access the full flexibility of Mozilla's interfaces through XPCOM and JavaScript.

Additionally, Mozilla has built up the Netscape Portability Runtime (NSPR)[Mozilla Foundation 2000, #98]. This layer abstracts all of the non-user-interface differences between the different platforms that Mozilla supports. It should be noted that when developing Apache HTTP Server 2.0, the Apache developers approached the Mozilla developers about using NSPR for their base portability layer as well. However, licensing differences between these groups caused the construction of the Apache Portable Runtime components which is now used by Apache HTTP Server and several other projects.

Integration. Through the Gecko engine, other applications can import the functionality of Mozilla into their own applications[Evans 2002, #28]. Gecko is described as:

> the browser engine, networking, parsers, content models, chrome and the other technologies that Mozilla and other applications are built from. In other words, all the stuff which is not application specific.[Mozilla Foundation 2004, #101]

While applications embedding Gecko have a fine-grained behavior over browsing, it is rather inflexible in its approach in that it forces the application to fit the mold of a web

browser[Lock 2002, #67]. Therefore, Gecko-derived applications tend to be variants on Mozilla but are not functionally much different than Mozilla.

TABLE 10. REST Architectural Constraints: Mozilla and Firefox

Constraint	Imposed Behavior
Representation Metadata	Visitor pattern on nsIHttpChannel object allows examination of metadata for requests and responses
Extensible Methods	Yes
Resource/Representation	Extensions can now operate on more than just Content-Type
Internal Transformation	Changes can occur even without embed tags
Proxy	Can pass requests to a proxy
Statefulness	Cache can now handle multiple representations over time which alleviates the negative stateful impact of Cookies
Cacheability	Yes

Lessons Learned. Significant effort to clean up architecture; Internal extensibility provided via JavaScript and C++; distinct rendering engine permits integration by third parties but isn't sufficiently powerful to permit different kinds of applications

MICROSOFT INTERNET EXPLORER

Microsoft's Internet Explorer can trace its lineage back to the NCSA Mosaic codebase. To bootstrap their delivery of a web browser, Microsoft initially licensed the code for a web browser from a company called Spyglass. Spyglass, in turn, was the commercial variant of NCSA Mosaic for Windows-based platforms. However, in the years since the first release of Internet Explorer, the corresponding code base and architecture greatly evolved to the point where it now has little architectural similarity with the original Mosaic architecture.

After Internet Explorer 6 was released, Microsoft disbanded the IE development team. Around this same time, a slew of security vulnerabilities were discovered that presented extreme risks to their users. Besides limiting the responsiveness to security reports, this absence in the market also led to an opening for other competitors to innovate. Given these criticisms and advances made by competitors, Microsoft has recently reformed the Internet Explorer team to focus on a new release of Internet Explorer 7 set to coincide with the next major Windows release of Windows Vista currently slated for late 2006. One of the stated goals of this new version is to revamp Internet Explorer's architectural approach to better support security. Therefore, we will examine the current state of the Internet Explorer architecture and look towards the architecture that has been disclosed for the upcoming Internet Explorer 7.

INTERNET EXPLORER ARCHITECTURE

Due to Internet Explorer's closed-source nature, the lack of access to source code presents a difficulty to recover a detailed and accurate software architecture representation. However, we can rely upon publicly available architectural information made available by Microsoft. One such source, presented in Figure 15 on page 35, is a public architectural description of Microsoft Internet Explorer for Windows CE available on the Microsoft Developer Network (MSDN) website.[Microsoft Corporation 2005, #92] Another source for architectural information about Internet Explorer is contained within recent presentations given by Microsoft's Internet Explorer development team discuss-

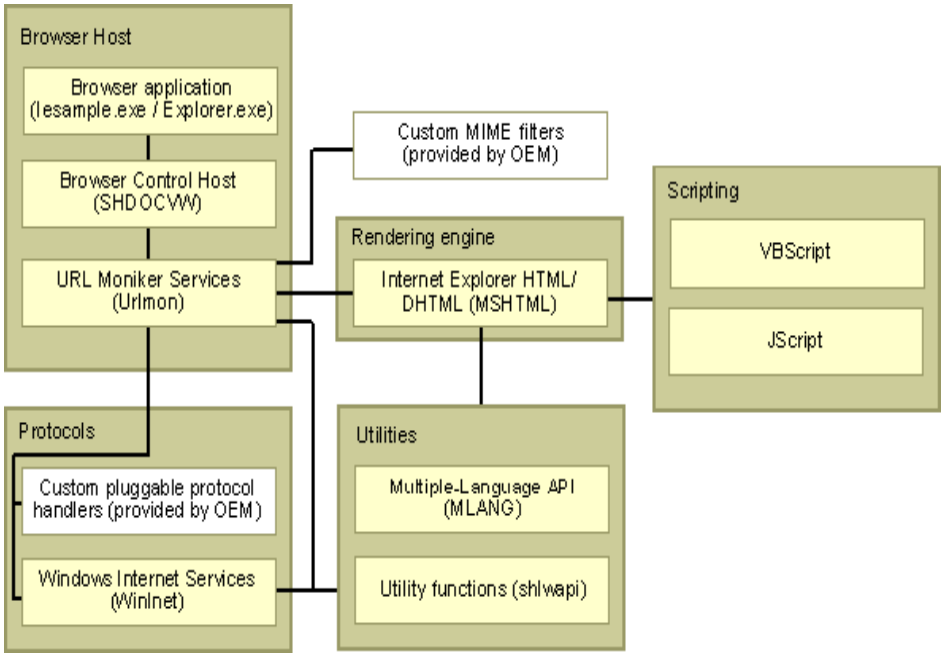

FIGURE 15. Microsoft Internet Explorer for Windows CE (From [Microsoft Corporation 2005, #92])

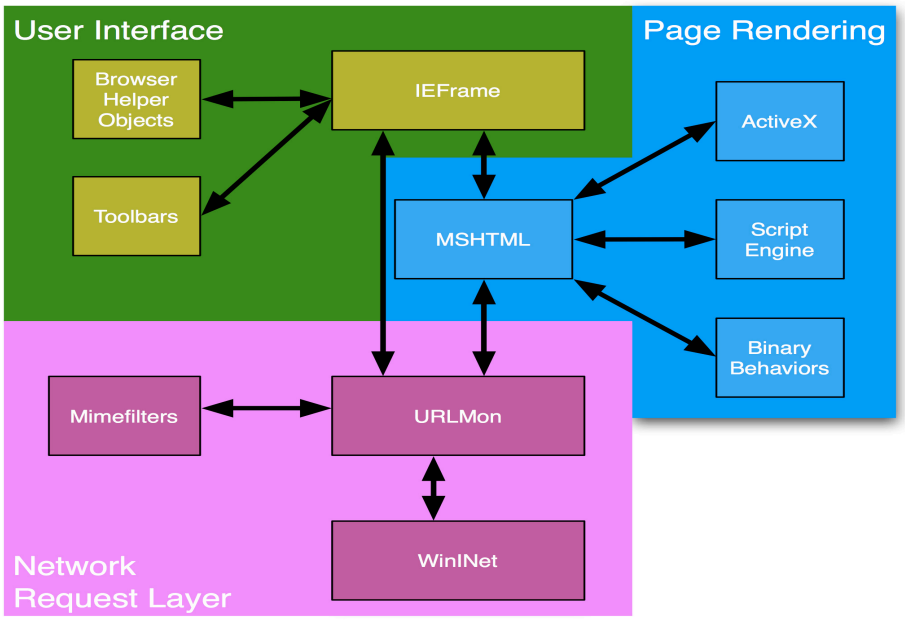

FIGURE 16. Microsoft Internet Explorer Architecture (Adapted from [Chor 2005, #18])

ing the impact of security on IE's architecture.[Chor 2005, #18] This particular architectural representation of Internet Explorer is reproduced in Figure 16 on page 35.

Portability. Besides the ubiquitous Windows versions, there have been versions of Internet Explorer released on a variety of non-Microsoft operating systems, including Mac OS, Solaris, HP-UX, and a variety of handheld devices. However, as alternatives emerged, these non-Windows platforms have quietly had their support dropped in recent years[Microsoft Corporation 2005, #93]. Additionally, the code base behind these versions of Internet Explorer was often independent of the code base for the main Windows-based version. Therefore, while the Internet Explorer brand name was shared across implementations, behind the scenes, there was usually little in common. For our purposes, we will only consider the Windows-based architectures of Internet Explorer.

Internal Extensibility. With Internet Explorer 4.0, Microsoft introduced a set of PowerToys that allowed developers to produce extensions to Internet Explorer. These extensions were contained in an HTML file which Internet Explorer would execute to alter its behavior. Example modifications that were supported was controlling the zoom factor of an image, listing all of the links on a page, and displaying information about the current page.[Microsoft Corporation #80]

In Internet Explorer 5.0, this functionality was broadened to allow modification by COM objects and events[BowmanSoft 2001, #14]. At the same time, the feature set was renamed to Web Accessories. One facet of modification was through "bands" which dedicate a region of the Internet Explorer window to a third-party extension[Microsoft Corporation #81]. These bands can display any desired information in this region through any programming language that supports COM objects and events. New download managers, toolbar buttons, and menu items can be added through Web Accessories[Microsoft Corporation #82, Microsoft Corporation #83, Microsoft Corporation #84]

However, Web Accessories only have access to relatively high-level and coarse-grained user-centric events. A plug-in that wishes to examine the complete HTTP headers for a response must install a custom proxy. This proxy must then interface with the WinInet layer to capture the HTTP stream and relay it externally to the Web Accessory plug-in. An example of such a system, Fiddler, is provided by Microsoft[Lawrence 2005, #66].

Through the URLmon component, developers can also register an *asynchronous pluggable protocol* to register a new protocol or MIME filter[Microsoft Corporation #85]. When a resource is requested, Internet Explorer will use the provided URL scheme (such as http or ftp) to look up which module defines the protocol interactions. The request is then handed off and the module initates the proper protocol. Besides raw protocols, filters can be registered that will be invoked whenever a representation's mime-type matches, which allows for custom internal transformations of the representation before the user will see the result.

Integration. Through a COM object called WebBrowser, any Windows application can import the functionality of Internet Explorer[Microsoft Corporation #86]. All of the browsing and parsing functionality is then handled internally by Internet Explorer. Additional customizations can be introduced through Browser Helper Objects, which allow a developer to customize the look and feel of the browser[Esposito 1999, #27].

External Extensibility and Run-time Architecture. The run-time architecture of Internet Explorer is presented in Figure 16 on page 35. The external extensibility items that are supported are in the "page rendering" layer via the MSHTML components. In addition to the other mechanisms (such as JavaScript through the Script Engine component) that IE supports, the key addition with IE is support for ActiveX controls.[Microsoft Corporation #87] Internet Explorer can act as a container for these COM objects and content developers can request their inclusion through special HTML tags. When requested and if it is installed on the client machine, an ActiveX control will appear as part of the returned page. It should be noted that until Internet Explorer 6.0, ActiveX controls would automatically be downloaded and installed without asking permission from the user[Microsoft Corporation 2004, #91]. This presented serious security risks.

These security concerns with ActiveX arise from the fact that these controls can perform any action on the computer that the current user can perform.[Microsoft Corporation #88] One defensive mechanism that was introduced is that a control developer can mark a control as "safe for scripting." If an ActiveX control isn't marked as safe for scripting, it can not be executed by Internet Explorer. Unfortunately, most developers do not have enough knowledge about when to mark a control as safe for scripting or not. Microsoft themselves allowed Internet Explorer to be scripted by external sites until IE 6.0[Microsoft Corporation #89]. Even with these opt-in measures available, as we will discuss next with Internet Explorer 7, the lack of privilege separation in Internet Explorer 6 and earlier still present significant opportunities for malicious attacks that can compromise the system.

Internet Explorer 7 Architecture

Faced with the deluge of security vulnerabilities, Microsoft has embarked on a rewrite of Internet Explorer focused on introducing a security-centric architecture to the next release of Internet Explorer to be shipped with the upcoming Windows Vista release currently expected in the second half of 2006.

Internet Explorer's current Group Program Manager, Tony Chor, admits that "compatibility and features trumped security" in previous versions of Internet Explorer[Chor 2005, #18]. The main problems identified were that various architectural flaws and deficiencies combined to lead to the poor security measures of Internet Explorer. Among those identified was that Internet Explorer led users to be confused about the impact of certain choices, architectural vulnerabilities were exposed allowing malicious code to be installed, and attacks on the extensibility features present in Internet Explorer. To rectify this situation going forward, this new version introduces a revised architectural model aimed at improving the security characteristics of IE.

The core of Internet Explorer 7's redesigned architecture will rely upon a new feature in Windows Vista called user account protection (UAP). This new operating system feature segregates the normal day-to-day operation of the user with the administrative functions. This divide prevents a process from being able to perform malicious activities without explicit authorization. Microsoft claims that "the goal of UAP is to enable the use of the Windows security architecture as originally designed in Microsoft Windows NT 3.1 to protect the system so that the these [threat] scenarios are blocked."[Microsoft Corporation 2005, #94]

IE7 will now run at this lower “privilege mode.” This implies that the IE process will be prevented from writing outside a set of specified directories or communicating with other higher-privilege processes[Silbey 2005, #126] If a requested operation (such as saving a file) would violate the privilege, the new Windows Vista system will provide the user with the ability to block the operation from completing or explicitly allow the operation. Certain high-risk sequences, such as installing an ActiveX control, will require administrator rights.

In order to maintain compatibility with previous extensions, the exposed ActiveX interfaces will remain the same. However, any commands that require additional privileges will be stopped and explicit authorization will be requested along with a description of the command that is being attempted.

TABLE 11. REST Architectural Constraints: Internet Explorer

Constraint	Imposed Behavior
Representation Metadata	Event triggered before navigation to view outbound headers If want to view headers, use a separate proxy server
Extensible Methods	No
Resource/Representation	Extensions can only operate on content-type
Internal Transformation	MIME filters can be registered as protocol handlers
Proxy	Security zones allows proxy requests on a zone basis
Statefulness	Limited control over cache for state considerations
Cacheability	Yes

Lessons Learned. Again, a lack of separation in run-time architecture presented serious security risks; portability strategies mandating independent implementations for each platform may lead to maintenance concerns that force eventual product withdrawal; appropriately combining with operating system security capabilities can improve overall security.

KONQUEROR

Konqueror from the K Desktop Environment project (KDE) is one of the few user agents that can not trace its heritage to the NCSA Mosaic codebase. Instead, Konqueror evolved from the file manager for the KDE environment. The Konqueror name itself is a subtle reference to the other browsers. The KDE developers explain it thusly:

> After the Navigator and the Explorer comes the Conqueror; it's spelled with a K to show that it's part of KDE. The name change also moves away from “kfm” (the KDE file manager, Konqueror's predecessor) which represented only file management.[KDE e.V. 2005, #60]

Konqueror’s architecture, as extracted by Grosskurth and Echihabi, is presented in Figure 17 on page 39.

Portability. All KDE components are written in C++ with window management duties delegated to the QT library. Befitting Konqueror’s heritage as the file manager on the KDE desktop, it is an intrinsic part of the KDE environment and relies heavily upon the services provided by other KDE components. Therefore, Konqueror can not truly be viewed as a stand-alone application, but rather as a fundamental part of a desktop envi-

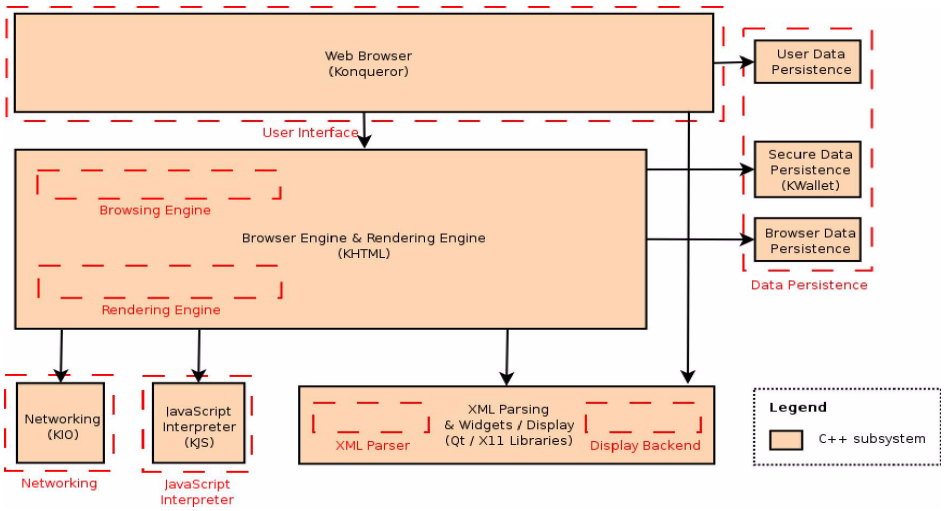

FIGURE 17. Konqueror Architecture (From [Grosskurth and Echihabi 2004, #49] Figure 12)

ronment. This limits the portability of Konqueror in that it will operate on any system that supports KDE, but due to the large dependency chain, the Konqueror application as a whole can not easily be considered separately from KDE.

Run-time architecture and External Extensibility. The application called Konqueror is just a relatively thin layer on top of other KDE components. One of Konqueror's main dependencies is upon the khtml engine which handles the rendering of any returned representation (such as for HTML, JavaScript, and CSS). Through a Konqeuror plugin and operating system emulation, Konqueror can also support ActiveX controls.[KDE e.V. 2001, #59] khtml is also responsible for directly interfacing with the networking layer (kio) - therefore, the Konqueror application never directly interfaces with the networking layer. As we will discuss later with Safari, khtml represents the most important functional component of Konqueror.

Internal Extensibility. Konqueror's main extensibility mechanism is through KDE's KParts component framework[Faure 2000, #29]. Through KParts, a developer can render media elements inside the Konqueror window[Granroth 2000, #45]. However, KParts only really supports embedding of an application inside of a Konqueror window. KParts does not specifically permit a developer to alter the look and feel of the Konqueror application.

If extensions to the underlying protocol are desired, new protocols can be added through KDE's networking layer. All protocols that are supported by Konqueror are handled through ioslaves. The KDE input/output layer understands the concepts of URLs and can delegate protocol handling to registered modules. However, there is no mechanism

to extend a specific URL handler - therefore, any extensions to a protocol would have to be handled through a completely separate kioslave mechanism.

TABLE 12. REST Architectural Constraints: Konqueror

Constraint	Imposed Behavior
Representation Metadata	Konqueror itself does not have access to any metadata KHTML and kio do to varying extents
Extensible Methods	New URL scheme would be required
Resource/Representation	KHTML modifications based upon content-type
Internal Transformation	kio layer has concept of filter transformations
Proxy	Can pass requests through the kio layer
Statefulness	Explicit state support
Cacheability	Yes, handled by the KHTML and kio layer

Lessons Learned. Evolution from a different application; tight integration with other layers of platform offers compartmentalization, but it blurs the distinction between an application and the platform it is on top of.

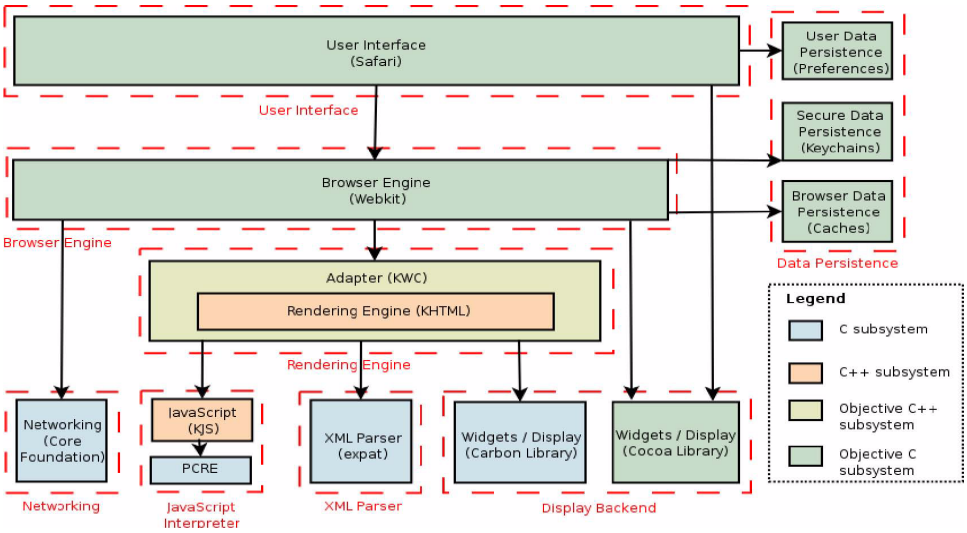

FIGURE 18. Safari Architecture (From [Grosskurth and Echihabi 2004, #49] Figure 14)

SAFARI

In 2003, Apple announced the Safari Web Browser for their Macintosh OS X platform. Until this time, the prevailing browser on Mac OS X was Microsoft's Internet Explorer for Macintosh. In the words of one expert on CSS, with Internet Explorer for Macintosh, "the port (of Internet Explorer) to OS X has gone horribly wrong, and I've written 5.2 off."[Koch 2004, #62]. Therefore, Apple decided to produce their own browser; but instead of writing the browser from scratch, they decided to examine other alternatives.

Run-time architecture. As seen in Safari's architecture, as extracted by Grosskurth and Echihabi and presented in Figure 18 on page 40, Safari is based upon Konqueror's KHTML rendering engine and KJS JavaScript engine. Apple's development manager explained their choice to the KDE community as follows:

> The number one goal for developing Safari was to create the fastest web browser on Mac OS X. When we were evaluating technologies over a year ago, KHTML and KJS stood out. Not only were they the basis of an excellent modern and standards compliant web browser, they were also less than 140,000 lines of code. The size of your code and ease of development within that code made it a better choice for us than other open source projects. Your clean design was also a plus. And the small size of your code is a significant reason for our winning startup performance.[Melton 2003, #70]

Since a number of developers on Apple's Safari team had previously worked for Netscape on Mozilla, the implied questions that quickly arose focused on why Apple did not choose Mozilla for their engine instead. Many viewed this as an attack on Mozilla[Festa 2003, #30]. A Mozilla contributor, Christopher Blizzard, dismissed those claims as follows:

> First of all, I don't think that we should be having the Safari vs. Mozilla/Chimera discussion at all. It takes our eyes off of the real prize (Internet Explorer) and that which we all should be worried about. I mean, if you control the browser, you control the Internet. It sounds kooky, but it's true. When we squabble amongst ourselves it doesn't do us any good.
> That being said, I do have a few things to say about the fact that Apple is going this alone. First, it's great that they decided to choose an open source solution, even if it isn't Mozilla. If they manage to engage the KHTML community and get well integrated with them then they have the chance to enjoy the fruits of that relationship, like as we have with the Mozilla project. ...
> Now, is our layout engine huge and ungainly and hard to understand? Yes. Yes it is. And, at least to some degree it's important to understand that Mozilla's layout engine has warts because the web has warts. It's an imperfect place and that leads to imperfect code. Remember that while KHTML is a good bit smaller than our layout engine, it also doesn't render a lot of sites anywhere near as well as Mozilla does. Over time, they are going to have to add many of the same warts to KHTML as we have to our layout engine. They might be able to do so in a more clean way, but they will still be there.[Blizzard 2003, #13]

Portability. Safari extracted the KHTML and KJS code from KDE and used that as the base rendering engines for their web browser. The main consequence of this is that the rendering characteristics of Safari and Konqueror are largely the same because they use the same rendering components. However, this extraction did not change the implementation language of the KDE components (which was originally in C++), therefore Safari relies upon a wide mix of programming languages to achieve its tasks: ranging from C, C++, Objective-C, and Objective-C++.

All of the code in KHTML that depends on the KDE component foundation had to be changed to work on Mac OS X's foundation instead. For example, all of the windowing primatives based on QT in KHTML had to be adapted to Mac OS X's Cocoa interfaces. According to one Mozilla developer, Safari "also took one of the most complex and effort-intensive parts of Gecko (Mozilla's rendering engine), the view manager, to add to KHTML, because Gecko's worked so well."[Baker 2003, #9, Shaver 2003, #124]

Internal Extensibility. Safari supports two kinds of extensibility mechanisms: Netscape-compatible plug-ins and WebKit[Apple Computer Inc. 2005, #3]. To ease the transition for prospective users and developers, plug-ins designed against the Netscape plug-in interface will work with Safari without any source modification. The end-user only needs to receive a Mac OS X-compiled version of the Netscape plug-in from the developer. Recent versions of Safari also offer custom extensions to the Netscape plug-in architecture to support scripting functions[Apple Computer Inc. 2005, #4]

On top of support for Netscape plug-ins, Safari also offers a set of extensions in Objective-C called WebKit[Apple Computer Inc. 2005, #5, Apple Computer Inc. 2005, #6]. Extensions written for WebKit allows use of Apple's bundled development tools for easy construction.[Apple Computer Inc. 2005, #7] Since most of Apple's Mac OS X extensions are already written in Objective C, the learning curve for the WebKit framework is not high for developers who are already familiar with Apple's extension frameworks. Therefore, WebKit's target audience is squarely those who are already familiar with Apple's frameworks not those who are interested in just the web browsing functionality.

Integration. By leveraging the WebKit interface, applications on Mac OS X can reuse the services provided by Safari. Dashboard, a new widget system introduced in Mac OS X 10.4, uses the WebKit engine for retrieving dynamic content and rendering the items on the user's screen.[Apple Computer Inc. 2005, #8] Contrary to statements which have stated otherwise, the iTunes player on Mac OS X does not use WebKit.[Hyatt 2004, #56] Since WebKit would not likely be available on other platforms due to its extreme dependency on Mac OS X services, this would preclude iTunes from using WebKit because a major platform target for iTunes is Windows.

TABLE 13. REST Architectural Constraints: Safari

Constraint	Imposed Behavior
Representation Metadata	Dictionary of all header fields for HTTP objects
Extensible Methods	Like KDE, new URL schemes required to extend methods
Resource/Representation	Response and Request have defined objects
Internal Transformation	Modifications of returned HTML content through DOM
Proxy	Can pass requests to a proxy
Statefulness	Each application can define a policy for accepting cookies
Cacheability	Yes, on a per-application basis

Lessons Learned. Possible to take generic, portable code and make it optimized for only one platform; OS's native framework system allows any application to integrate a web browser; applications that use WebKit only work on that OS though.

Libraries and Frameworks

The origin servers and user agents we have examined so far provide a complete usable system aimed at either content providers and interested end-users. However, not all

RESTful applications fit the mold of an HTTP server or web browser. Some applications which take part of the RESTful part follow a completely different interaction paradigm. To serve these needs, a collection of RESTful frameworks have emerged to provide the structural necessitities for these applications. Again, we will limit ourselves to the criteria presented in “Selecting Appropriate RESTful Applications” on page 7.

One notable characteristic of this classification of systems is that most of the systems described here do not provide support for external extensibility mechanisms. Interpretation of HTML, JavaScript, and CSS are typically associated with the role of web browsers. Therefore, developers looking to integrate web browsing into their application will tend to integrate one of the user agent architectures instead of using one of these frameworks.

LIBWWW

Having been around in some form since 1992, one of the oldest frameworks for designing and constructing HTTP applications is libwww[1] [Aas 2004, #1, Fielding 1998, #32, Kahan 2003, #57]. libwww has been used to design and develop a variety of HTTP client applications, such as the Amaya web browser [Vatton 2004, #151].

Portability and Run-time Architecture. libwww is written in C and has been explicitly ported to Unix, Windows, and Macintosh platforms. However, there is no portability layer - so all developers using libwww must explicitly handle platform differences themselves. While not directly supporting threads, libwww is built upon an event loop model[Nielsen 1999, #111]. An application can register its own event loop that will be called whenever an event is triggered. Through this event loop and non-blocking networking performance, libwww can handle multiple connections simultaneously.

Internal Extensibility. There have been conflicting descriptions about the underlying architecture of libwww. One popular description of the architecture of W3C’s libwww can be found in Chapter 7 of Bass [Bass, Clements 1998, #11]. Here, the architecture of libwww is divided into five layers: *application*, *access*, *stream*, *core*, and *generic utilities*. They also claim that libwww can be utilized to construct both server and client-side HTTP applications. Finally, they present the following lessons that can be learned from libwww: 1) Formalized APIs are required; 2) A layered architectural style must be adopted; 3) An open-ended set of features must be supported; 4) Applications should be thread-safe.

However, the original designers of libwww have presented their architecture as a “Coke Machine” architecture [Frystyk Nielsen 1999, #39]. This view of the architecture provides designers with a wide range of functionality, in no particular ordering, that can be used to construct a RESTful application. Furthermore, while libwww could theoretically be used to write server applications, the stated intent is for W3C’s libwww to be a “highly modular, general-purpose client side Web API written in C.” [Kahan 2003, #57] Hence, the express focus of libwww is therefore on helping to develop HTTP clients not servers. The initial positioning as an HTTP client framework introduces fundamental

1. This name is shared by at least two unrelated libraries; we refer to the W3C’s C-based libwww.

assumptions throughout the framework that raise serious challenges when designing applications with libwww for other REST connector types.

TABLE 14. REST Architectural Constraints: libwww

Constraint	Imposed Behavior
Representation Metadata	Restricted set of headers that can be set or fetched
Extensible Methods	Yes
Resource/Representation	Separate request and response structures
Internal Transformation	Filter mechanisms to morph content with chaining
Proxy	Can pass requests to a proxy
Statefulness	Cookies can be handled through extension mechanisms
Cacheability	Yes

Lessons Learned. Providing a framework that is not inherently coupled to a web browser is feasible; however, interface is too limited to use it for any other REST node.

cURL

cURL ("client for URLs") is an open-source project focused on facilitating the retrieval or transmission of content with a wide range of protocols through URLs[Stenberg 2006, #132]. Two sub-projects are distributed as part of Project cURL: libcurl, a C library, and curl, a command-line program built on top of libcurl. Since the curl command-line program is a thin wrapper on top of libcurl, we will focus principally on the attributes of libcurl. libcurl provides support for a number of protocols, including FTP, HTTP, TELNET, and LDAP and is available on most currently available operating systems.

As of this writing, the main author behind cURL is currently embarking on a 'high performance' version of cURL, called hiper, that will add HTTP Pipelining and a greater degree of parallelism [Stenberg 2005, #131].

Portability. libcurl is written in C and has been ported to almost all modern operating systems today. Additionally, libcurl also has a number of bindings to over 30 different languages available (such as Java, Python, Perl, Lisp, and Visual Basic). Therefore, a developer can leverage libcurl in their preferred programing language. This process is helped by the fact that almost all programming languages provide some mechanism for interacting with C libraries. However, these bindings are not uniform in the functionality provided. Each language binding provides a range of libcurl's functionality. Some of these bindings export only the minimal functionality of libcurl (such as the easy interfaces), while other bindings provide the complete functionality of libcurl to that particular language.

Run-time Architecture. libcurl offers two interfaces for developers: an *easy* interface and the *multi* interface. With the *easy* interface, a developer can simply provide a URL and the response will be emitted to the end-user's screen by default. With the *multi* interface, a number of requests can be handled simultaneously by libcurl. However, the libcurl design specifically requires that any application using the multi interface manage any threads and network connections independently. Therefore, if a developer wishes to multiplex across different connections in a threading environment, they must manage the asynchronous communication without libcurl's assistance. This greatly increases

the burden on developers attempting to use libcurl; therefore, most libcurl extensions tend to shy away from the multi interface.

Internal Extensibility. A developer can extend the functionality of libcurl through the use of options. These options are in the form of key-value pairs that are set by the application before the communication process with the server begins. At specific well-defined points in time, libcurl will examine its options to determine if and how its behavior should be altered. For example, a callback function can be provided that will be invoked whenever libcurl wants to write the response to a request. By default, libcurl will write to the user's screen; by replacing that option with a callback to a developer-defined function, the application can process the response in memory or other tasks as desired.

Importability. The main application that uses libcurl is the curl command-line client itself. curl provides users with the ability to tranfer files through URLs and supports all of the underlying protocols that libcurl supports. A selection of applications that use libcurl include[Stenberg 2006, #133]:

- clamav - a GPL anti-virus toolkit for UNIX
- git - Linux source code repository tool
- gnupg - Complete and free replacement for PGP
- libmsn - C++ library for Microsoft's MSN Messenger service
- OpenOffice - a multiplatform and multilingual office suite

None of these applications would be viewed as traditional RESTful applications like a web server or browser, but each of them incorporates RESTful functionality through libcurl.

TABLE 15. REST Architectural Constraints: libcurl

Constraint	Imposed Behavior
Representation Metadata	Requests: Private linked list; can add headers Responses: Metadata combined with data stream
Extensible Methods	Yes
Resource/Representation	Lack of separation between the resource being requested and the returned representation
Internal Transformation	Option mechanism allows only one level of chaining
Proxy	Can pass requests to a proxy
Statefulness	Explicit support for setting, preserving, or ignoring cookies
Cacheability	No

Lessons Learned. Truly different applications from a web browser can be created on top of a RESTful framework; providing support for a vast range of languages can increase penetration; multiple interfaces allow for a gentle learning curve.

HTTPClient / HTTP Components

The Apache Software Foundation's Jakarta Commons HTTPClient library is a Java-based HTTP client framework. HTTPClient focuses on "providing an efficient, up-to-

date, and feature-rich package implementing the client side of the most recent HTTP standards and recommendations."[The Apache Software Foundation 2005, #144] Note that, as of this writing, the project is preparing to be renamed to "Jakarta HTTP Components."

Portability. HTTPClient is written in the Java programming language, therefore it requires a Java Virtual Machine (JVM) to operate. While Java does provide a simple HTTP client interface in its standard class libraries, it is not easily extensible and does not support a wide-range of features. Therefore, HTTPClient focuses on offering a more complete range of features compared to the built-in Java interfaces. An overview of replacement Java HTTP client frameworks are available at [Oakland Software Incorporated 2005, #112].

Run-time Architecture. HTTPClient will attempt to reuse connections via HTTP Keep-Alive's wherever possible via connection pooling strategies. Therefore, HTTPClient requires that developers explicitly release a connection after it is done to return it to the connection pool. If the connection is still viable and has been released while another request is conveyed to the same server, it will reuse the open connection. HTTPClient can also support multiple concurrent connections through its *MultiThreadedHttpConnectionManager* class. Each connection is allocated to a specific thread with the manager class being responsible for multi-plexing the active connections efficiently across threads.

Internal Extensibility. Since HTTPClient is written in Java, it is also written in an object-oriented manner. Therefore, any core HTTPClient class can be extended and replaced to alter its functionality. For instance, HTTP methods are introduced by extending the primitive method classes. HTTPClient also supports a wide-range of authentication mechanisms through this same object-oriented extensibility mechanisms. HTTPClient also supports altering its protocol compliance through the use of a preference model.

Importability. Due to the choice of Java, most usage of HTTPClient is restricted to Java applications. Still, a broad range of applications have emerged using HTTPClient. The following is a selection of applications which have been written on top of HTTPClient[The Apache Software Foundation 2005, #145]:

- Jakarta Slide - a content repository and content management framework
- Jakarta Cactus - a simple test framework for unit testing server-side Java code
- LimeWire - a peer-to-peer Gnutella client
- Dolphin - a Java-based Web browser
- Mercury SiteScope - a monitoring program for URLs and lots more

TABLE 16. REST Architectural Constraints: HTTPClient

Constraint	Imposed Behavior
Representation Metadata	Metadata fields part of request and response objects
Extensible Methods	Yes
Resource/Representation	Separates request and response streams as discrete objects

TABLE 16. REST Architectural Constraints: HTTPClient

Constraint	Imposed Behavior
Internal Transformation	Extensible object model allows for one level of chaining
Proxy	Can pass requests to a proxy
Statefulness	Explicit support for setting, preserving, or ignoring cookies
Cacheability	No

Lessons Learned. Possible to construct RESTful frameworks in Java; however, applications using HTTPClient are realistically limited to only those applications written in Java.

NEON

neon differs from the other client frameworks mentioned so far in that it is focused on supporting a specific extension to HTTP: WebDAV[Orton 2005, #114]. Web Distributed Authoring and Versioning (WebDAV) is an official extension of HTTP which facilitates multiple authors collaboratively editing REST resources[Clemm, Amsden 2002, #19, Goland, Whitehead 1999, #43, Whitehead and Wiggins 1998, #154]. Therefore, in addition to basic HTTP client functionality, neon offers a number of features that are of specific interest to WebDAV clients.

Portability. The neon library is written in the C programming language which does not have explicit memory management support. Therefore, neon does offer some memory management capabilities on top of the standard C libraries. neon can be configured in a special memory-leak detection mode which tracks all allocations to the source files where the allocation was initially made. Still, all memory allocations must be explicitly deallocated or leaks will occur.

Since neon is not built on top of an explicit portability layer, it must therefore handle all of the differences between platforms itself. neon offers support for Windows explicitly. Unix-based platforms are supported through GNU autoconf, which facilitates auto-discovery of most features of platform.[Free Software Foundation 2005, #38] Additionally, bindings to the Perl language are available for neon.

Internal Extensibility. Like libcurl, neon offers two levels of interfaces: a *simple* interface and a low-level interface. Most developers can leverage the simple interfaces to perform basic HTTP client tasks. These simple interfaces wrap a more intrinsic interfaces which help shields the user from unnecessary complexities. If more complicated client operations are required, the lower-level interfaces are available for use.

Extensibility with neon occurs through passing callbacks pointers that are then invoked at certain points in time during the response lifecycle. With WebDAV methods, many of the responses are often XML-based. To provide assistance to applications interacting with WebDAV,. neon offers the ability to give callback functions that are invoked during the XML parsing stage. This allows the application not to have to deal with the parsing themselves while retaining the ability to see the parsed data.

Run-time architecture. neon presents a synchronous network-blocking run-time architecture. When a user requests a URL from neon, control will not be returned until the response has been completely handled by the registered handlers and readers. In addi-

tion to these readers, neon offers the ability to receive notifications at certain connection-level events (such as when a connection is established). At this time, neon does not support handling multiple connections at the same time.

Importability. Due to neon's focus on incorporating WebDAV-friendly features, applications that take advantage of WebDAV are the target audience. Since WebDAV is an extension to HTTP, neon can also perform HTTP tasks as well[Stenberg 2003, #130]. Applications that use neon include:

- Litmus - a WebDAV server test suite
- Subversion - a version control system that is a compelling replacement for CVS which uses WebDAV
- davfs2 - WebDAV Linux File System

TABLE 17. REST Architectural Constraints: neon

Constraint	Imposed Behavior
Representation Metadata	Request: Add metadata fields to request structure
	Response: Register callbacks for specific metadata names
Extensible Methods	Yes
Resource/Representation	Separate request and response structures
Internal Transformation	Explicit function to support a representation transformation
Proxy	Can pass requests to a proxy
Statefulness	Cookie support either enabled or disabled by developer
Cacheability	No

Lessons Learned. A RESTful framework that focuses on providing support for an HTTP extension (in this case, WebDAV) is possible and desired for those applications that use these extensions.

SERF

Serf is an HTTP client framework that is inspired by the Apache HTTP Server's design[Stein and Erenkrantz 2004, #129]. Serf was designed by some of the principal architects of Apache HTTP Server. (This author is one of those architects behind serf.) One of serf's principal goals was to explore the question of whether a REST-centric framework written for an origin server can also apply to a client. Due to these goals, serf shares a lot of conceptual ideas with the Apache HTTP Server. Besides transporting these ideas to a client, Serf also takes the opportunity to rethink some of the design decisions made by the Apache HTTP Server.

Portability. Serf is written in C on top of the Apache Portable Runtime (APR) portability layer. This is the same portability layer currently used by Apache HTTP Server. Therefore, the cost of portability are shared with a much larger project that already has an established portability layer. Additionally, serf uses the same pool-based memory management model used by Apache HTTP Server. Therefore, serf's memory model is similar to that of Apache HTTP Server's.

Internal Extensibility. The key extensibility concept in serf is that of buckets. These buckets represent data streams which can have transformations applied to them dynami-

cally and in a specific order. This name can trace its origins through Apache HTTP Server[Woolley 2002, #156] and, from there, back to the libwww-ada95 library[Fielding 1998, #33]. In turn, this layering concept is related to Unix STREAMS[Ritchie 1984, #118].

Serf's usage of buckets has more in common with the Onions system of libwww-ada95 than with Apache HTTP Server's bucket brigade model. The description of Onions described its model as:

> A good network interface should be constructed using a layered paradigm in order to maximize portability and extensibility (changing of underlying layers without affecting the interface), but at the same time must avoid the performance cost of multiple data copies between buffers, uncached DNS lookups, poor connection management, etc. Onions are layered, but none of the layers are wasted in preparing a meal.[Fielding 1998, #34]

However, Onions was only implemented as an abstract layer without any actual client implementations completed. Apache HTTP Server 2.x implemented a complete system around their bucket brigade system and serf based its initial bucket types on the choices represented in Apache HTTP Server. Therefore, serf represents a fusion of the concepts behind Onions and the concrete contributions of Apache HTTP Server.

Run-time architecture. Serf is designed to perform non-blocking network connections - this means that, at no time, do serf buckets wait to write or read data on the network. Therefore, the buckets can only process the immediately available data. This decision was made to allow serf to handle more connections in parallel than other synchronous (network-blocking) frameworks. If no data is available to be written or read on any active connection, serf will leverage platform-specific optimizations to wait until such data is available (such as polling). Therefore, serf can gracefully scale up to handling large numbers of network connections in parallel as it will only be active when data is immediately available.

This decision to support asynchronous network behavior comes at a cost of extra complexity in writing buckets for serf. This complexity is related to the fact the bucket can not wait for the next chunk of data - only the connection management code can perform these wait operations. In order to address this concern, serf buckets must be written following the behavior of a finite-state machine. If enough data is not available to proceed to the next stage, then the bucket must indicate that it can not proceed further. After all connections reach this exhausted state, the connection manager will then enter the waiting state until more data is received.

Importability. At this point, no specific applications exist which use serf. A simple program which fetches resources using serf is available. There is also a proof-of-concept threading spidering program that uses serf's parallelization and pipelining capabilities. Plans are currently in place to integrate Subversion with serf. The rationale behind this integration is that Subversion has introduced a number of custom WebDAV methods for performance reasons because neon does not support HTTP pipelining[Erenkrantz 2005, #26]. We believe that that serf can resolve these performance problems and

remove the need for custom methods solely for performance reasons[Roy's post about REPORT considered harmful].

TABLE 18. REST Architectural Constraints: serf

Constraint	Imposed Behavior
Representation Metadata	Requests: Add metadata fields to request bucket
	Responses: Retrieve metadata bucket chain from response
Extensible Methods	Yes
Resource/Representation	Explicit response and request buckets
Internal Transformation	Multiple transformations can be applied independently
Proxy	Can pass requests to a proxy
Statefulness	No cookie support
Cacheability	No

Lessons Learned. Possible to reuse portability layers from a RESTful server (Apache HTTP Server) in a RESTful framework; asynchronous behavior places additional constraints on developers; transformations through STREAM-like interfaces increases flexibility in transformations

Constructing RESTful Application Architectures

Even in the presence of these systems that have been described so far, these architectures do not fully describe all RESTful applications. Fully-functional REST applications, like electronic-commerce web sites, leverage these architectures already described to build a larger application. However, the fact that particular internal architectural constraints foster the benefits provided by REST does not imply that an application building upon that style could never violate the REST principles. We will now examine a few technologies that are commonly used to build RESTful applications and how they interact with the REST constraints.

COMMON GATEWAY INTERFACE (CGI)

NCSA described a prototypical Common Gateway Interface (CGI) application as:

> For example, let's say that you wanted to "hook up" your Unix database to the World Wide Web, to allow people from all over the world to query it. Basically, you need to create a CGI program that the Web daemon will execute to transmit information to the database engine, and receive the results back again and display them to the client. This is an example of a gateway, and this is where CGI, currently version 1.1, got its origins.[National Center for Supercomputing Applications 1995, #103]

A CGI program can be written in any compiled programming language (e.g. C, Java, etc.) or can be interpreted through a scripting language (e.g. Unix shell, Perl, Python, TCL, etc.). The only requirement is that the CGI must be executable on the underlying platform. When a CGI program is invoked by httpd, the CGI program can rely on four ways to transfer information from the CGI program and the webserver and vice versa:

- Environment variables: determine the metadata sent via the HTTP request

- Command line: determine if there are any server-specific arguments
- Standard input: receive request bodies from the client (such as through POSTs)
- Standard output: Set the metadata and data that would be returned to the client

Example CGI Applications. Deployment of CGI was common by 1994 and documentation relating to CGI was included in the NCSA HTTPd documentation.[National Center for Supercomputing Applications 1995, #104] The NCSA HTTPd 1.3 release included a number of example CGI scripts. One example included in NCSA HTTPd 1.3 was an order form for Jimmy John's submarine shop located in Champaign, Illinois (cgi-src/jj.c). Upon initial entry to the submarine shop site, an order form was dynamically presented to the user listing all of the ordering options: subs, slims, sides, and pop. The user would then submit an HTML form for validation. The jj CGI script would then validate the submitted form to ensure that the name, address, phone, and a valid item order was placed correctly. After validation, orders were then submitted via an email to FAX gateway for further processing.

REST Constraints. We begin to see a constraint of the external architecture peeking through with CGI: HTTP mandates synchronous responses. Therefore, while the CGI program was processing the request to generate a response, the requestor would be 'on hold' until the script completes. During the exection of the script, NCSA warned that "the user will just be staring at their browser waiting for something to happen."[National Center for Supercomputing Applications 1995, #103] Therefore, CGI script authors were advised to make the execution of their scripts short so that it did not cause the user on the other end to wait too long for the response.

CGI introduced clear support for two REST constraints: extensible methods and namespace control. Although, CGI was most commonly used with the GET and POST HTTP methods, other methods could be indicated through the passed *REQUEST_METHOD* environment variable. This allows the CGI script to respond to new methods as they are generated by the client.

Additionally, CGI scripts could define an arbitrary virtual namespace under its own control. This was achieved by the *PATH_INFO* environment variable. NCSA's CGI docs describe *PATH_INFO* as:

> The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as PATH_INFO. This information should be decoded by the server if it comes from a URL before it is passed to the CGI script.

For example, if a CGI program is nominally located at */cgi-bin/my-app*, then a request to */cgi-bin/my-app/this/is/the/path/info*, would execute the my-app CGI program and the *PATH_INFO* environment variable would be set to "*/this/is/the/path/info*". This presents the appearance of a namespace that the CGI script can respond to appropriately.

HTML Forms

A browser supporting HTML forms allows a content developer to allow the end-user to fill out fields on a web page and submit these values back to the server. Without forms, the interaction a user could have with a site was relatively limited as they could not specify any input to be submitted to the server other than the selection of a hyperlink. A

simple example of an HTML form as it would appear to a user as shown in Figure 19 on page 52.

```
<FORM METHOD="POST" ACTION="http://www.example.com/cgi-bin/post-query">
First Name: <INPUT NAME="first"><br/>
Last Name: <INPUT NAME="last"><br/>
To submit the query, press this button: <INPUT TYPE="submit" VALUE="Submit">.
 </FORM>
```

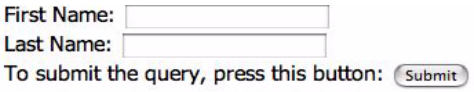

FIGURE 19. Form Browser Example (HTML snippet and screenshot)

As shown above, there are two key HTML tags in an HTML form: *form* and *input*. The *form* tag declares to the browser that a form should be displayed. Within the *form* tag, the *method* attribute indicates whether a GET or POST method should be used when the form is submitted and the *action* attribute specifies what URL the method should be performed on. The *input* tag defines all of the fields in the form. The *type* attribute indicates the format of the data field. A special type attribute is the "SUBMIT" field which indicates that when this button is selected, the entire form is submitted to the server.

Deployment. NCSA Mosaic for X 2.0, released in November 1993, was one of the first browsers to support FORM tags.[Andreessen 1993, #2, National Center for Supercomputing Applications 1999, #105] A specification of forms was first included in HTML+ announced in November 1993 as well[Raggett 1993, #117]. Almost all browsers after that point included HTML forms support and form usage remains a cornerstone of websites to this day.

REST Constraints. Forms have a particular interaction within the REST semantics. For a "GET" *action* form, the data is submitted appended to the specified URL as a query string using the GET HTTP method. In the example above, if a user typed 'John' into the 'first' field and 'Smith' into the 'last' field and chose to submit the form, the corresponding GET request would look like:

```
http://www.example.com/cgi-bin/post-query?first=John&last=Smith
```

However, if the *action* specified a "POST", that same form would be submitted to the *http://www.example.com/cgi-bin/post-query* resource with the following request body:

```
first=John&last=Smith
```

Limitations in early browsers limited the amount of data that could be appended to a GET query string; therefore, usage of forms gravitated towards POST forms instead of GET. Depending upon the meaning of form submissions (specifically whether or not it changed the underlying resource), this could be an incorrect usage of the POST method.

JAVASCRIPT

As discussed in "Early Netscape Navigator Architecture" on page 26, JavaScript was first introduced with Netscape Navigator in 1995. JavaScript is a client-side interpreted scripting language that allowed content developers, through special HTML tags, to con-

trol the behavior of the browser.[Champeon 2001, #17] Therefore, it differs from server-side scripting languages like PHP in that it is executed within the context of the user agent - not that of the origin server. However, the content developer still remains in control of the script. After the success of Navigator, almost all browsers subsequently introduced JavaScript support. Additionally, the JavaScript language is now an ECMA standard.[Eich and Clary 2003, #22] As mentioned with "Current Mozilla Architecture" on page 31, JavaScript as a language provides the core extensibility language for Mozilla Firefox extensions.

Brendan Eich, the initial implementor of JavaScript at Netscape, relates the beginning of JavaScript, "I hacked the JS prototype in ~1 week in May [1995]...And it showed! Mistakes were frozen early"[Eich 2005, #23] This new scripting language was originally called "Mocha", but was later renamed to "JavaScript" due to marketing influences between Netscape and Sun. While JavaScript's syntax was loosely modeled after the Java programming language, the relationship was only superficial. The object model of JavaScript was inspired more by HyperCard than Java and was tailored to the specific minimal needs of a content designer attempting to control the browser. JavaScript would be embedded inside of the HTML representations and a JavaScript-aware browser could then interpret these embedded scripts on the client-side.

More recently, sites are now using asynchronous JavaScript mechanisms and other browser technologies (under the collective moniker AJAX) to create richer web-centric applications.[Garrett 2005, #40] Earlier works such as KnowNow's JavaScript-based micro-servers presaged this later work.[Khare and Taylor 2004, #61, Udell 2001, #149] However, these AJAX applications only take advantage of the services already provided by modern browsers. Therefore, they are relying squarely upon the RESTful extensibility mechanisms provided by the current generation of user agents.

Discussion

Through our examination of these RESTful architectures, a clear pattern emerges that can describe the progress made over the last ten years as viewed through our framework prisms. These evolutionary stages are:

- External Extensibility - Attract end-users
- Internal Extensibility - Attract developers
- Portability - Expand the reach of the underlying architecture
- Run-time Architecture - Improve performance and lessen security vulnerabilities

At each stage, we can clearly see how the constraints set forth by REST interacted with the decisions made by architects to improve their systems.

EXTERNAL EXTENSIBILITY

As we have seen, initially, RESTful applications (although it wasn't termed as such then) featured extensibility only through external modifications that were not part of the internal architecture. There were very few changes that could be made architecturally to NCSA httpd and Mosaic. In a hypermedia domain, as was the initial target of the

WWW's creators, being able to support various types of content is critically important. Instead of just supporting delivery of static files, NCSA httpd introduced CGI to allow different forms of content to be dynamically generated and delivered to the client. This concept has evolved to other scripting languages such as PHP, JSP, and ASP which offer more specialized features meant for constructing Web-enabled content.

Similarly, NCSA Mosaic introduced the concept of helper applications in order to permit the user to view a broad spectrum of media types. This allowed formats that Mosaic did not natively understand to be viewed by an external application. However, by having such a sharp divide between these helper applications and the user agent, the browsing experience suffered a severe blow since the concept of having links between content was lost. Netscape Navigator repaired this problem by introducing internal content plug-ins which could render specific media types inside the browser window and maintain the complete hypermedia experience.

These initial choices represented the priorities of the communities at that time. At the early stages of the WWW, the main goal was to attract end-users - not architects. This goal predictably led to architectures focused on interfacing with external applications. As these capabilities were utilized by a wider community, more people became interested in how to change the behavior of the architecture dynamically. The need for more expressive and powerful architectures became understood.

Internal Extensibility

Once this critical mass of users was reached, businesses started to investigate how they could leverage the WWW for their own purposes. Due to their experiences with the initial basic hypermedia content, they began to understand more about what they could conceptually achieve with the WWW. Eventually, electronic-commerce and other richer Web-enabled application were conceived. This led to a boom of interest around the core infrastructure providing this framework. However, the architectures present at that time were not flexible enough to address their individual needs for this next generation of websites.

These assessments led to either radical rewrites (Apache with its Shambala fork, Mozilla with XPCOM) or new code bases (NCSA Mosaic to Netscape Navigator) that greatly improved the extensibility of the overall system compared to their predecessors. Those architectures, such as NCSA httpd, that did not have these extensibility characteristics faced marginalization over time and have largely disappeared from use.

The defining characteristic of these new architectures is that they focused heavily on extensibility by allowing extension designers to alter the behavior of the system dynamically without altering the original implementation. Instead of providing a monolithic architecture that aimed to achieve every conceivable task, these architectures provided for a minimal core that could be extended through well-defined mechanisms. In the case of Apache, almost all functionality bundled with the server is not built into the core, but rather through its own extensibility mechanisms (*hooks and filter*s). Over time, a strong community of external extension designers emerged that modified Apache to suit their needs. Without this minimal and extensible core, the diverse range of Apache modules would not have been possible.

Portability

For those architectures that did not explicitly target a single platform (as Microsoft's IIS and Internet Explorer did), the next challenge was how to support a broad range of platforms without sacrificing performance or other beneficial characteristics. This work led to the production of two platform abstraction layers: APR with Apache HTTP Server and NSPR with Mozilla. Notably, these abstraction layers are characterized by providing optimizations on platforms where they are available. This is in contrast to the "least common denominator" approach taken by other portability layers and programming languages.

Additionally, while some ultimately aborted efforts were undertaken to rewrite these RESTful architectures in a "better" programming language such as Java, the top choices remain C and C++. By using C directly, as seen with cURL, a number of bindings to other languages can be provided which allow extension developers to enhance the architecture in the language of their choice.

Even though most of our surveyed systems are written in C or C++, most have incorporated special features to help deal with supposed shortcomings of C - specifically with regards to memory management. In some instances, these features take advantage of the RESTful protocol constraints. For example, the Apache HTTP Server takes extreme advantage of the defined RESTful processing flow in its memory management model. Instead of tying itself to a non-deterministic garbage collector (such as offered by Java), Apache's memory model ties allocations to the life span of an HTTP response. This offers a predictable memory model that makes it easier for developers to code modules with Apache, not suffer from memory leaks, and offer significant performance advantages.

Run-Time Architecture

After the previous three dimensions were addressed, we often see a return to the initial run-time architecture decisions. By this time, the systems have usually had a lot of real-world experience to provide substantial feedback as to how the run-time architecture could be improved. RESTful protocols through its mandated explicit stateful interactions imply that the ideal run-time architecture does not need to exhibit complex coordination capabilities - each interaction can be handled independently and in parallel. Even with this beneficial characteristic, the scalability of the architecture can strain the underlying operating environment with certain types of workloads. Therefore, threading or asynchronous network behavior is introduced to the architecture. However, the cost of adding threading or asynchronous behavior after the system has been deployed is extremely prohibitive.

We see the negative effects of this with the jump from Apache HTTP Server 1.3 to 2.0 through the introduction of threading with the MPM policy layer. As part of the transition from 1.3 to 2.0, extension developers had to make their modules thread-safe. Many Apache modules were not written with thread-safety in mind and hence have not been updated to the new versions of Apache due to the additional complexity in making the code thread-safe. Retrofiting in threads was also painful for the Mozilla architecture as early Mozilla builds had to introduce a new networking layer so that the network layer would be multi-threading. Therefore, if high-performance workloads are ultimately desired from a RESTful system, threading and asynchronous network access should be essential anticipated qualities from the beginning of the architectural design.

Scalability is not the only reason to reconsider the run-time architecture of these RESTful systems. As seen with IIS 6.0 and the forthcoming Internet Explorer 7, poor early architectural decisions about the run-time architecture can impact the security of the system by not providing enough barriers against malicious behaviors.

Impact of Security on RESTful Architectures

In the provenance of a RESTful world, extensibility can not remain unchecked. Due to the proliferation and ubiquitous nature of the WWW today, these RESTful architectures are constantly under attack by malicious entities. Worms like Code Red, which specifically attacked Microsoft's IIS, caused two noticeable reactions: a slight drop in market share of the affected product and a new security-centric architecture release. Microsoft reacted to the attacks by redesigning IIS to focus on security at the expense of extensibility. Microsoft is also redesigning Internet Explorer in similar ways for an upcoming versions of Windows to combat its poor security reputation.

Therefore, from an architectural perspective in this domain, we can view security as the imposition of constraints on extensibility. For these RESTful systems, the minimal core architecture is generally trusted to be secure - however, extensions or content are no longer as trusted as they once were. A fence has been erected between the core of the RESTful architecture and its components. The absence of this fence came at an extreme price to those people who had their systems compromised due to faults that a sound architecture could have prevented.

While the link between security and extensibility is real, it is however not quite as strong as Microsoft claims with their Internet Information Services 6 and Internet Explorer 7 rearchitectures. They may be over-emphasizing the importance of security due to their own past poor attitudes towards security. Other competitors, such as Apache HTTP Server and Mozilla, have an arguably better long-term reputation towards security than Microsoft. While these projects haven't been free of security vulnerabilities either, large-scale attacks haven't occurred against their products.

The reason for these lack of attacks can't be attributed to poor market share alone: Apache HTTP Server currently has a 2-to-1 advantage over IIS according to Netcraft[Netcraft 2005, #107]. Mozilla Firefox has made improvements in its market share in the last year by trying to capitalize on the security problems with Internet Explorer in the minds of the consumers. A commentator recently compared Firefox's security with Internet Explorer's and remarked:

> I ask only that the vendor be responsible and fix the security vulnerabilities, especially the critical ones, in a timely fashion. Microsoft isn't one of those vendors. According to Secunia, Internet Explorer 6.x has several unpatched, critical security vulnerabilities dating back to 2003 (the first year Secunia offered its own security alerts). And this month, Microsoft arrogantly decided not to issue any security patches--none.[Vamosi 2005, #150]

Future Directions: REST Constraints

The obvious question that remains is what should be the architectural focus going forward for these RESTful architectures. We believe that the next evolutionary stage to emerge is going to be specifically centered on addressing these REST constraints that we have highlighted. None of the surveyed systems offer a perfect fit with the REST constraints we outlined. To lend further credence to this argument, we are beginning to see hints of progress on precisely this front. The recent release of Apache HTTP Server

2.2 focused heavily on three areas: improving control over filters - an integral part of supporting internal transformations, a more scalable proxy, and a production-quality cache mechanism.[The Apache HTTP Server Project 2005, #139] The recent release of Firefox 1.5 introduces a revised cache system that offers better stateful characteristics.[Mozilla Corporation 2005, #97]

Impact of Market Share on Future Adoption

The architectures that remain ten years hence have evolved to facilitate exposure of the core interfaces to support both rapid internal and third-party modifications. While it is not impossible to introduce new RESTful origin servers or user agents today, there is a definite gravity effect in place that prevents new products from capturing large amounts of market share for HTTP servers and browsers. On the other hand, frameworks for RESTful applications that are not traditional servers and browsers have not yet reached a point where there is a compelling universal choice. Each RESTful framework that we examined serves a slightly different clientele with its own set of architectural tradeoffs. Therefore, we believe there remains an opportunity for introducing a set of RESTful frameworks that are targeted towards these different REST applications.

Lessons For Future RESTful Frameworks

While libraries like libwww, cURL, and others can serve as architecture frameworks for the REST style, we believe that these frameworks focus too heavily on acting as HTTP protocol implementations and provide too few services to be effective in developing complete end-to-end RESTful applications. In contrast, we envision a family of frameworks that focus generically on the construction of RESTful applications from the perspective of all the various participants: servers, proxies, caches, and clients. Furthermore, an ideal framework will focus deeply on the REST constraints, such as state management and protocol extensibility, that have largely been ignored by other frameworks, but are among the most difficult parts of REST implementations to “get right.”

To assist future RESTful framework developers, Table 19 on page 57 summarizes the lessons that we believe are important for future architects to incorporate in future architectural framework decisions based on collective past experiences.

TABLE 19. Lessons for Future RESTful Architects

Prism	Lesson
REST Constraints	Tying the architecture to one REST node type impacts future flexibility Early familiarity with these constraints can prevent later conflicts
External Extensibility	Provide same services at a minimum as other systems; If not designing a typical web server or client, may not be as important
Integration	Provide multiple interfaces to balance the learning curve and power Allow external architects fine-grained control over integration options
Internal Extensibility	Provide a minimal core architecture; the rest should be modular Provide appropriate hooks to allow developers to alter your behavior Support multiple representation transformations through filter chains

TABLE 19. Lessons for Future RESTful Architects

Prism	Lesson
Portability	Optimize by understanding how REST can influence workload High-quality portability layers are now available for reuse Language selection has an impact on integration options
Run-time Architecture	Explicit state of REST lends itself well to independent processes Higher scalability loads helped by threads and network asynchronous behavior, but has a high cost if added after the fact; add it early Separation and isolation can minimize vulnerability attack surfaces

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0438996.

Bibliography

1. Aas, G. *libwww-perl*. <http://lwp.linpro.no/lwp/>, HTML, June 25, 2004.
2. Andreessen, M. *NCSA Mosaic for X 2.0 available*. <http://ksi.cpsc.ucalgary.ca/archives/WWW-TALK/www-talk-1993q4.messages/444.html>, Email, November 10, 1993.
3. Apple Computer Inc. *Introduction to Web Kit Plug-in Programming Topics*. <http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit_PluginProgTopic/index.html>, HTML, August 11, 2005.
4. ---. *About Web Browser Plug-ins*. <http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit_PluginProgTopic/Concepts/AboutPlugins.html>, HTML, August 11, 2005.
5. ---. *Introduction to Web Kit Objective-C Programming Guide*. <http://developer.apple.com/documentation/Cocoa/Conceptual/DisplayWebContent/index.html>, HTML, April 29, 2005.
6. ---. *URL Loading System Overviewww.* <http://developer.apple.com/documentation/Cocoa/Conceptual/URLLoadingSystem/Concepts/URLOverview.html>, HTML, October 8, 2005.
7. ---. *Creating Plug-ins with Cocoa and the Web Kit*. <http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit_PluginProgTopic/Concepts/AboutPlugins.html>, HTML, August 11, 2005.
8. ---. *Mac OS X - Dashboard*. <http://www.apple.com/macosx/features/dashboard/>, HTML, 2005.
9. Baker, M. *Browser Innovation, Gecko and the Mozilla Project*. <http://www.mozilla.org/browser-innovation.html>, HTML, February 25, 2003.
10. Bandhauer, J. *XPJS Components Proposal*. <http://www.mozilla.org/scriptable/xpjs-components.html>, HTML, July 1, 1999.
11. Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison Wesley: Reading, MA, 1998.
12. Berners-Lee, T., Fielding, R., and Frystyk, H. *Hypertext Transfer Protocol -- HTTP/1.0*. Internet Engineering Task Force, Request for Comments Report 1945, May, 1996.
13. Blizzard, C. *Untitled*. <http://www.0xdeadbeef.com/html/2003/01/index.shtml#20030114>, HTML, January 14, 2003.
14. BowmanSoft. Mastering Internet Explorer: The Web Browser Control. *Visual Basic Web Magazine*. 2001. <http://www.vbwm.com/art_2001/IE05/>.
15. Braverman, A. *X Web Teach - a sample CCI application.* <http://archive.ncsa.uiuc.edu/SDG/

Software/XMosaic/CCI/x-web-teach.html>, National Center for Supercomputing Applications,, HTML, September 23, 1994.

16. Brown, M. FastCGI: A High-Performance Gateway Interface. In *Proceedings of the Programming the Web - a search for APIs Workshop at Fifth International World Wide Web Conference*. Paris, France, May 6, 1996. <http://www.cs.vu.nl/~eliens/WWW5/papers/FastCGI.html>.
17. Champeon, S. JavaScript: How Did We Get Here? *O'Reilly Web DevCenter.* April 6, 2001. <http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html>.
18. Chor, T. Internet Explorer Security: Past, Present, and Future. In *Proceedings of the Hack in the Box*. Kuala Lumpur, Malaysia, September 26-29, 2005. <http://www.packetstormsecurity.org/hitb05/Keynote-Tony-Chor-IE-Security-Past-Present-and-Future.ppt>.
19. Clemm, G., Amsden, J., Ellison, T., Kaler, C., and Whitehead, E.J. *RFC 3253: Versioning Extensions to WebDAV.* IETF, Request for Comments Report, March, 2002.
20. Coar, K.A.L. and Robinson, D.R.T. *The WWW Common Gateway Interface Version 1.1*. <http://cgi-spec.golux.com/draft-coar-cgi-v11-03-clean.html>, HTML, June 25, 1999.
21. Cook, M. *Securing I.I.S. 5 and 6 Server.* <http://escarpment.net/training/Securing_Microsoft_IIS_5_and_6(slides).pdf>, PDF, February 14, 2005.
22. Eich, B. and Clary, B. *JavaScript Language Resources*. <http://www.mozilla.org/js/language/>, HTML, January 24, 2003.
23. Eich, B. JavaScript at Ten Years. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*. Tallinn, Estonia, September 26-28, 2005. <http://www.mozilla.org/js/language/ICFP-Keynote.ppt>.
24. ---. *New Roadmaps*. <http://weblogs.mozillazine.org/roadmap/archives/009218.html>, HTML, November 3, 2005.
25. Erenkrantz, J.R. *Web Services: SOAP, UDDI, and Semantic Web.* Institute for Software Research, Report UCI-ISR-04-3, May, 2004. <http://www.isr.uci.edu/tech_reports/UCI-ISR-04-3.pdf>.
26. ---. *Serf and Subversion*. <http://svn.haxx.se/dev/archive-2005-11/1373.shtml>, Email, November 27, 2005.
27. Esposito, D. *Browser Helper Objects: The Browser the Way You Want It.* <http://msdn.microsoft.com/ie/iedev/default.aspx?pull=/library/en-us/dnwebgen/html/bho.asp>, Microsoft Corporation, HTML, January, 1999.
28. Evans, E. *Gecko Embedding Basics*. 2002. <http://www.mozilla.org/projects/embedding/embedoverview/EmbeddingBasics.html>.
29. Faure, D. Chapter 13: Creating and Using Components (KParts). In *KDE 2.0 Development*, Sweet, D. ed. Sams Publishing, 2000.
30. Festa, P. Apple snub stings Mozilla. *CNet News*. January 14, 2003. <http://news.com.com/2100-1023-980492.html>.
31. Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. *Hypertext Transfer Protocol -- HTTP/1.1*. Internet Engineering Task Force, Request for Comments Report 2616, June, 1999.
32. Fielding, R.T. *libwww-perl: WWW Protocol Library for Perl*. <http://ftp.ics.uci.edu/pub/websoft/libwww-perl/>, HTML, June 25, 1998.
33. ---. *libwww-ada95: WWW Protocol Library for Ada95*. <http://ftp.ics.uci.edu/pub/websoft/libwww-ada95/>, HTML, May 13, 1998.
34. ---. *Onions Network Streams Library.* <http://ftp.ics.uci.edu/pub/websoft/libwww-ada95/current/Onions/README>, Text, May 13, 1998.
35. ---. Shared Leadership in the Apache Project. *Communications of the ACM*. 42(4), p. 42-43, 1999.
36. ---. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Thesis. Information and Computer Science, University of California, Irvine, 2000. <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
37. Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*. 2(2), p. 115-150, May, 2002.

38. Free Software Foundation. *Autoconf*. <http://www.gnu.org/software/autoconf/>, HTML, February 2, 2005.
39. Frystyk Nielsen, H. *W3C Libwww Review.* W3C, Report, June, 1999. <http://www.w3.org/Talks/1999/06/libwww/>.
40. Garrett, J.J. *Ajax: A New Approach to Web applications*. <http://www.adaptivepath.com/publications/essays/archives/000385.php>, HTML, February 18, 2005.
41. Gaudet, D. *Apache Performance Notes*. <http://httpd.apache.org/docs/1.3/misc/perf-tuning.html>, HTML, September 30, 1997.
42. Godfrey, M.W. and Lee, E.H.S. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proceedings of the Second Symposium on Constructing Software Engineering Tools (CoSET'00)*. Limerick, Ireland, June, 2000.
43. Goland, Y., Whitehead, E.J., Faizi, A., Carter, S., and Jensen, D. *RFC 2518: HTTP Extensions for Distributed Authoring – WEBDAV.* Internet Engineering Task Force, Request for Comments Report 2518, p. 1-94, February, 1999.
44. Gosling, J. and Yellin, F. *Window Toolkit and Applets*. The Java(TM) Application Programming Interface. 2, 406 pgs., Addison-Wesley Professional, 1996.
45. Granroth, K. *Embedded Components Tutorial*. <http://www.konqueror.org/componentstutorial/>, HTML, March 3, 2000.
46. Gröne, B., Knöpfel, A., and Kugel, R. Architecture recovery of Apache 1.3 -- A case study. In *Proceedings of the 2002 International Conference on Software Engineering Research and Practice*. Las Vegas, 2002. <http://f-m-c.org/publications/download/groene_et_al_2002-architecture_recovery_of_apache.pdf>.
47. Gröne, B., Knöpfel, A., Kugel, R., and Schmidt, O. *The Apache Modeling Project*. Hasso Plattner Institute for Software Systems Engineering, Report, July 5, 2004. <http://f-m-c.org/projects/apache/download/the_apache_modelling_project.pdf>.
48. Grosskurth, A. and Echihabi, A. *Concrete Architecture of Mozilla*. University of Waterloo, Report, October 24, 2004. <http://www.cs.uwaterloo.ca/~agrossku/2004/cs746/mozilla-concrete.pdf>.
49. ---. *A Reference Architecture for Web Browsers*. University of Waterloo, PDF Report, December 7, 2004. <http://www.cs.uwaterloo.ca/~agrossku/2004/cs746/browser-refarch-slides.pdf>.
50. Grosskurth, A. and Godfrey, M.W. A Reference Architecture for Web Browsers. In *Proceedings of the 2005 International Conference on Software Maintenance*. Budapest, Hungary, September 25-30, 2005.
51. Harris, W. and Potts, R. *Necko: A new netlib kernel architecture*. <http://www.mozilla.org/docs/netlib/necko.html>, HTML, April 14, 1999.
52. Hassan, A.E. and Holt, R.C. A Reference Architecture for Web Servers. In *Proceedings of the Seventh Working Conference on Reverse Engineering*. p. 150-159, 2000. <http://doi.ieeecomputersociety.org/10.1109/WCRE.2000.891462>.
53. Hibbs, C. Rolling with Ruby on Rails. *O'Reilly Databases*. January 20, 2005. <http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html>.
54. Hunter, J. and Crawford, W. *Java Servlet Programming*. 2nd ed. 753 pgs., O'Reilly Media, Inc., 2001.
55. Hunter, J. New features added to Servlet 2.5. *JavaWorld*. January 2, 2006. <http://www.javaworld.com/javaworld/jw-01-2006/jw-0102-servlet.html>.
56. Hyatt, D. *iTunes and WWebKit*. <http://weblogs.mozillazine.org/hyatt/archives/2004_06.html#005666>, HTML, June 8, 2004.
57. Kahan, J. *Libwww - the W3C Sample Code Library.* <http://www.w3.org/Library/>, W3C, HTML, September, 2003.
58. Katz, E.D., Butler, M., and McGrath, R.E. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*. 27(2), p. 155-164, November, 1994. <http://dx.doi.org/10.1016/0169-7552(94)90129-5>.
59. KDE e.V. *Reaktivate Released*. <http://www.konqueror.org/announcements/reaktivate.php>, HTML, July 9, 2001.

60. ---. *Konqueror FAQ*. <http://konqueror.kde.org/faq/>, HTML, 2005.
61. Khare, R. and Taylor, R.N. Extending the REpresentational State Transfer Architectural Style for Decentralized Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*. Edinburgh, Scotland, UK, May, 2004. <http://doi.ieeecomputersociety.org/10.1109/ICSE.2004.1317465>.
62. Koch, P.-P. *Browsers - Explorer 5 Mac*. <http://www.quirksmode.org/browsers/explorer5mac.html>, HTML, November 23, 2004.
63. Kristol, D.M. and Montulli, L. *HTTP State Management Mechanism*. Internet Engineering Task Force, Request for Comments Report 2109, February, 1997. <http://www.ietf.org/rfc/rfc2109.txt>.
64. Kwan, T.T., McGrath, R.E., and Reed, D.A. NCSA's World Wide Web server: design and performance. *Computer.* 28(11), p. 68-74, November, 1995. <http://dx.doi.org/10.1109/2.471181>.
65. Larsson, A. *The Life of An HTML HTTP Request*. <http://www.mozilla.org/docs/url_load.html>, Mozilla Foundation, HTML, October 8, 1999.
66. Lawrence, E. *Fiddler PowerToy - Part 1: HTTP Debugging*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebgen/html/IE_IntroFiddler.asp>, Microsoft Corporation, HTML, January, 2005.
67. Lock, A. Embedding Mozilla. In *Proceedings of the Free and Open Source Software Developers' European Meeting (FOSDEM 2002)*. Brussels, Belgium, February 16-17, 2002. <http://www.mozilla.org/projects/embedding/FOSDEMPres2002/contents.html>.
68. Markham, G. The Mozilla Foundation. In *Proceedings of the Free and Open Source Software Developers' European Meeting (FOSDEM 2005)*. Brussels, Belgium, February 26-27, 2005. <http://www.gerv.net/presentations/fosdem2005-mofo/>.
69. McFarlane, N. *Rapid Application Development with Mozilla*. 704 pgs., Prentice Hall, 2003. <http://mb.eschew.org/>.
70. Melton, D. *Greetings from the Safari team at Apple Computer*. <http://lists.kde.org/?l=kfm-devel&m=104197092318639&w=2>, Email, January 7, 2003.
71. Microsoft Corporation. *HTTP Protocol Stack (IIS 6.0)*. <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/a2a45c42-38bc-464c-a097-d7a202092a54.mspx>, Microsoft Windows Server 2003 TechCenter, HTML.
72. ---. *IIS 6.0 Operations Guide (IIS 6.0)*. <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/f74c464e-6d5d-403c-97e7-747cd798dde2.mspx>, Microsoft Windows Server 2003 TechCenter, HTML.
73. ---. *Configuring Isolation Modes (IIS 6.0)*. <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/ed3c22ba-39fc-4332-bdb7-a0d9c76e4355.mspx>, Microsoft Windows Server 2003 TechCenter, HTML.
74. ---. *What's Changed (IIS 6.0)*. <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/7b037954-441d-4037-a111-94df7880c319.mspx>, Microsoft Windows Server 2003 TechCenter, HTML.
75. ---. *Application Isolation Modes (IIS 6.0)*. <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/7b037954-441d-4037-a111-94df7880c319.mspx>, Microsoft Windows Server 2003 TechCenter, HTML.
76. ---. *ISAPI Extension Overview*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/78f84895-003d-4631-8571-97042c06a4b8.asp>, IIS Web Development SDK, HTML.
77. ---. *ISAPI Filter Overview*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/22e3fbfb-1c31-41d7-9dc4-efa83f813521.asp>, IIS Web Development SDK, HTML.
78. ---. *Creating ISAPI Filters*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iissdk/html/773e63f0-1861-47fc-ac85-fa92e799d82c.asp>, IIS Web Development SDK, HTML.
79. ---. *Important Changes in ASP (IIS 6.0)*. <http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/e1a77c5d-046e-4538-8d9d-b2996c3143d3.mspx>,

Microsoft Windows Server 2003 TechCenter, HTML.
80. ---. *Web Accessories*. <http://msdn.microsoft.com/workshop/browser/accessory/overview/overview.asp>, Internet Explorer - Browser Extensions, HTML.
81. ---. *Creating Custom Explorer Bars, Tool Bands, and Desk Bands*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/Shell/programmersguide/shell_adv/bands.asp>, Advanced Shell Techniques, HTML.
82. ---. *Implementing a Custom Download Manager*. <http://msdn.microsoft.com/workshop/browser/ext/overview/downloadmgr.asp>, Internet Explorer - Browser Extensions, HTML.
83. ---. *Adding Toolbar Buttons*. <http://msdn.microsoft.com/workshop/browser/ext/tutorials/button.asp>, Internet Explorer - Browser Extensions, HTML.
84. ---. *Adding Menu Items*. <http://msdn.microsoft.com/workshop/browser/ext/tutorials/menu.asp>, Internet Explorer - Browser Extensions, HTML.
85. ---. *Asynchronous Pluggable Protocols*. <http://msdn.microsoft.com/workshop/networking/pluggable/pluggable.asp>, Internet Explorer - Asynchronous Pluggable Protocols, HTML.
86. ---. *WebBrowser Object*. <http://msdn.microsoft.com/workshop/browser/webbrowser/reference/objects/webbrowser.asp>, Internet Explorer - WebBrowser, HTML.
87. ---. *Introduction to ActiveX Controls*. <http://msdn.microsoft.com/workshop/components/activex/intro.asp>, Internet Development, HTML.
88. ---. *Designing Secure ActiveX Controls*. <http://msdn.microsoft.com/workshop/components/activex/security.asp>, Internet Development, HTML.
89. ---. *Reusing the WebBrowser Control*. <http://msdn.microsoft.com/workshop/browser/webbrowser/webbrowser.asp>, Internet Explorer - WebBrowser, HTML.
90. ---. *How To: Implementing Cookies in ISAPI*. <http://support.microsoft.com/kb/q168864/>, HTML, July 1, 2004.
91. ---. *Windows XP Service Pack 2: What's New for Internet Explorer and Outlook Express*. <http://www.microsoft.com/windowsxp/sp2/ieoeoverview.mspx>, HTML, August 4, 2004.
92. ---. *Internet Explorer 6.0 Architecture*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcedsn40/html/coconInternetExplorer55Architecture.asp>, Microsoft Windows CE .NET 4.2, HTML, April 13, 2005.
93. ---. *Internet Explorer 5 for Mac*. <http://www.microsoft.com/mac/products/internetexplorer/internetexplorer.aspx>, HTML, December 19, 2005.
94. ---. *Developer Best Practices and Guidelines for Applications in a Least Privileged Environment*. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnlong/html/AccProtVista.asp>, HTML, September, 2005.
95. Mockus, A., Fielding, R.T., and Herbsleb, J. A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of the International Conference on Software Engineering*. p. 263-272, ACM Press. Limerick, Ireland, June, 2000.
96. Moore, D., Shannon, C., and Brown, J. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the Internet Measurement Workshop*. Marseille, France, November 6-8, 2002. <http://www.caida.org/outreach/papers/2002/codered/codered.pdf>.
97. Mozilla Corporation. *Mozilla Firefox 1.5 Release Notes*. <http://www.mozilla.com/firefox/releases/1.5.html>, HTML, November 29, 2005.
98. Mozilla Foundation. *NSPR: Module Description*. <http://www.mozilla.org/projects/nspr/about-nspr.html>, HTML, September 20, 2000.
99. ---. *SeaMonkey Milestones*. <http://www.mozilla.org/projects/seamonkey/milestones/index.html>, HTML, February 9, 2001.
100. ---. *Old Milestone Releases*. <http://www.mozilla.org/projects/seamonkey/release-notes/>, HTML, November 27, 2002.
101. ---. *Mozilla Embedding FAQ*. <http://www.mozilla.org/projects/embedding/faq.html>, HTML, December 8, 2004.
102. National Center for Supercomputing Applications. *Application Programmer's Interface for the NCSA Mosaic Common Client Interface (CCI)*. <http://archive.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-api.html>, HTML, March 31, 1995.
103. ---. *CGI: Common Gateway Interface*. <http://hoohoo.ncsa.uiuc.edu/cgi/intro.html>, HTML,

December 6, 1995.
104. ---. *NCSA HTTPd Tutorial: CGI Configuration*. <http://hoohoo.ncsa.uiuc.edu/docs/tutorials/cgi.html>, HTML, September 27, 1995.
105. ---. *Mosaic for X version 2.0 Fill-Out Form Support*. <http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html>, HTML, May 11, 1999.
106. ---. *NCSA Mosaic Version History*. <http://www.ncsa.uiuc.edu/Divisions/PublicAffairs/MosaicHistory/history.html>, HTML, November 25, 2002.
107. Netcraft. *Netcraft Web Server Survey*. <http://www.netcraft.com/survey/>, HTML, December 2, 2005.
108. Netscape. *JavaScript Guide*. <http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/>, HTML, 1996.
109. ---. *Client Side State - HTTP Cookies*. <http://wp.netscape.com/newsref/std/cookie_spec.html>, HTML, 1999.
110. Newmarch, J. Extending the Common Client Interface with User Interface Controls. In *Proceedings of the AusWeb95: The First Australian WorldWideWeb Conference*. p. AW016-04, Australia, May 1-2, 1995. <http://ausweb.scu.edu.au/aw95/integrating/newmarch/>.
111. Nielsen, H.F. *Threads and Event Loops*. <http://www.w3.org/Library/User/Architecture/Events.html>, HTML, July 13, 1999.
112. Oakland Software Incorporated. *Java HTTP Client - Comparison*. <http://www.oaklandsoftware.com/product_16compare.html>, HTML, April, 2005.
113. Oeschger, I. *API Reference: Netscape Gecko Plugins*. 2.0 ed. 190 pgs., Netscape Communications, 2002. <http://www.ics.uci.edu/~jerenkra/netscape-plugin.pdf>.
114. Orton, J. *neon HTTP and WebDAV client library*. <http://www.webdav.org/neon/>, HTML, 2005.
115. Parrish, R. XPCOM Part 1: An Introduction to XPCOM. *developerWorks*. February 1, 2001. <http://www-128.ibm.com/developerworks/webservices/library/co-xpcom.html>.
116. Peiris, C. What's New in IIS 6.0? (Part 1 of 2). *DevX.com*. August 20, 2003. <http://www.devx.com/webdev/Article/17085>.
117. Raggett, D. *HTML+ (Hypertext markup format)*. <http://www.w3.org/MarkUp/HTMLPlus/htmlplus_1.html>, HTML, November 8, 1993.
118. Ritchie, D.M. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*. 63(8 Part 2), p. 1897-1910, October, 1984.
119. Rosenberg, D. Adding a New Protocol to Mozilla. 2004. <http://www.nexgenmedia.net/docs/protocol/>.
120. Saksena, G. Networking in Mozilla. In *Proceedings of the O'Reilly Open Source Convention*. San Diego, California, 2001. <http://www.mozilla.org/projects/netlib/presentations/osc2001/>.
121. Schatz, B.R. and Hardin, J.B. NCSA Mosaic and the World Wide Web: Global Hypermedia Protocols for the Internet. *Science*. 265(5174), p. 895-901, August 12, 1994.
122. Schmidt, J. ISAPI. *Microsoft Internet Developer*. Spring, 1996. <http://www.microsoft.com/mind/0396/ISAPI/ISAPI.asp>.
123. Sergeant, M. Using Qpsmtpd. *O'Reilly SysAdmin*. September 15, 2005. <http://www.oreillynet.com/pub/a/sysadmin/2005/09/15/qpsmtpd.html>.
124. Shaver, M. *back; popular demand*. <http://shaver.off.net/diary/2003/01/15/back-popular-demand/>, HTML, January 15, 2003.
125. Shaw, M. and Garlan, D. *Software Architecture: Perpectives on an Emerging Discipline*. 242 pgs., Prentice Hall, 1996.
126. Silbey, M. *More details on Protected Mode IE in Windows Vista*. <http://blogs.msdn.com/ie/archive/2005/09/20/471975.aspx>, HTML, September 20, 2005.
127. Smith, J. Optimizing and Performance Tuning IIS 6.0. *Informit.com*. September 10, 2004. <http://www.informit.com/articles/article.asp?p=335881>.
128. Spolsky, J. *Things You Should Never Do, Part I*. <http://www.joelonsoftware.com/articles/fog0000000069.html>, Joel On Software, HTML, April 6, 2000.
129. Stein, G. and Erenkrantz, J. *Serf Design Guide*. <http://svn.webdav.org/repos/projects/serf/

trunk/design-guide.txt>, HTML, September 14, 2004.
130. Stenberg, D. *libcurl vs neon for WebDav?* <http://curl.haxx.se/mail/lib-2003-03/0208.html>, Email, March 19, 2003.
131. ---. *High Performance libcurl - hiper*. <http://curl.haxx.se/libcurl/hiper/>, HTML, December 26, 2005.
132. ---. *cURL - Frequently Asked Questions*. <http://curl.haxx.se/docs/faq.html>, HTML, January 5, 2006.
133. ---. *Programs Using libcurl*. <http://curl.haxx.se/libcurl/using/apps.html>, HTML, January 5, 2006.
134. Suryanarayana, G., Erenkrantz, J.R., Hendrickson, S.A., and Taylor, R.N. PACE: An Architectural Style for Trust Management in Decentralized Applications. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*. p. 221-230, Oslo, Norway, June, 2004.
135. Taylor, R.N., Medvidovic, N., Anderson, K.M., E. James Whitehead, J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*. 22(6), p. 390-406, June, 1996.
136. Thau, R.S. *Notes on the Shambhala API*. <http://mail-archives.apache.org/mod_mbox/httpd-dev/199507.mbox/%3c9507051409.AA08582@volterra%3e>, Email, July 5, 1995.
137. ---. Design considerations for the Apache Server API. *Computer Networks and ISDN Systems*. 28(7-11), p. 1113-1122, May, 1996. <http://dx.doi.org/10.1016/0169-7552(96)00048-7>.
138. The Apache HTTP Server Project. *Apache MPM event*. <http://httpd.apache.org/docs/2.2/mod/event.html>, HTML, December 5, 2005.
139. ---. *Overview of new features in Apache 2.2*. <http://httpd.apache.org/docs/2.2/new_features_2_2.html>, HTML, December 13, 2005.
140. The Apache Portable Runtime Project. *Apache Portable Runtime Project*. <http://httpd.apache.org/dev/guidelines.html>, HTML, November 20, 2004.
141. The Apache Software Foundation. *Apache API notes*. <http://httpd.apache.org/docs/1.3/misc/API.html>, HTML, October 13, 2003.
142. ---. *The Apache HTTP Server Project*. <http://httpd.apache.org/>, HTML, 2004.
143. ---. *Apache HTTP Server Reaches Record Eight Consecutive Years of Technical Leadership*. <http://www.prnewswire.com/cgi-bin/stories.pl?ACCT=SVBIZINK3.story&STORY=/www/story/05-11-2004/0002172126&EDATE=TUE+May+11+2004,+02:15+PM>, HTML, May 11, 2004.
144. ---. *HttpClient*. <http://jakarta.apache.org/commons/httpclient/>, HTML, 2005.
145. ---. *HttpClientPowered*. <http://wiki.apache.org/jakarta-httpclient/HttpClientPowered>, HTML, 2005.
146. ---. *Apache Axis*. <http://ws.apache.org/axis/>, HTML, 2005.
147. Trudelle, P. *XPToolkit: Straight-up Elevator Story*. <http://www.mozilla.org/xpfe/ElevatorStraightUp.html>, HTML, February 21, 1999.
148. Turner, D. and Oeschger, I. *Creating XPCOM Components*. 2003. <http://www.mozilla.org/projects/xpcom/book/cxc/>.
149. Udell, J. The Event-Driven Internet. *Byte.com*. December 3, 2001. <http://www.byte.com/documents/s=1816/byt20011128s0003/1203_udell.html>.
150. Vamosi, R. Security Watch: In defense of Mozilla Firefox. *CNET Reviews*. September 23, 2005. <http://reviews.cnet.com/4520-3513_7-6333507-1.html>.
151. Vatton, I. *Amaya Home Page*. <http://www.w3.org/Amaya/>, HTML, May 3, 2004.
152. Wang, D. *HOWTO: Use the HTTP.SYS Kernel Mode Response Cache with IIS 6*. <http://blogs.msdn.com/david.wang/archive/2005/07/07/HOWTO_Use_Kernel_Response_Cache_with_IIS_6.aspx>, HTML, July 7, 2005.
153. Web Host Industry Review. Microsoft to Rewrite IIS, Release Patches. *Web Host Industry Review*. September 26, 2001. <http://www.thewhir.com/marketwatch/iis926.cfm>.
154. Whitehead, E.J. and Wiggins, M. WEBDAV: IETF Standard for Collaborative Authoring on

the Web. *IEEE Internet Computing*. p. 34-40, September/October, 1998.

155. Wilson, B. *Browser History: Netscape*. <http://www.blooberry.com/indexdot/history/netscape.htm>, HTML, September 30, 2003.
156. Woolley, C. Bucket Brigades: Data Management in Apache 2.0. In *Proceedings of the ApacheCon 2002*. Las Vegas, 2002. <http://www.cs.virginia.edu/~jcw5q/talks/apache/bucketbrigades.ac2002.ppt>.
157. Zawinski, J. *Grendel Overview.* <http://www.mozilla.org/projects/grendel/announce.html>, HTML, September 8, 1998.
158. ---. *java sucks*. <http://www.jwz.org/doc/java.html>, HTML, 2000.