

UNIVERSITY OF CALIFORNIA,  
IRVINE

Computational REST: A New Model for Decentralized, Internet-Scale Applications

DISSERTATION

submitted in partial satisfaction of the requirements  
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Justin Ryan Erenkrantz

Dissertation Committee:  
Professor Richard N. Taylor, Chair  
Professor Debra J. Richardson  
Professor Walt Scacchi

2009

Portions of Chapters 2,3,4,5 adapted from “From Representations to Computations: The Evolution of Web Architectures,” in Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (September, 2007) © ACM, 2007.

<http://doi.acm.org/10.1145/1287624.1287660>

Used with permission under Section 2.5 “Rights Retained by Authors” of the ACM Copyright Policy

All other content © 2009 Justin Ryan Erenkrantz



## **DEDICATION**

to

Mom and Dad

who always gave their unwavering love  
even when their son chose the hard road

## TABLE OF CONTENTS

	Page
LIST OF FIGURES	vi
LIST OF TABLES	viii
ACKNOWLEDGEMENTS	ix
CURRICULUM VITAE	xii
ABSTRACT OF THE DISSERTATION	xv
INTRODUCTION	1
CHAPTER 1: Architectural Styles of Extensible RESTful Applications	8
Software Architecture and Frameworks	8
Software Architecture in the World Wide Web	9
Representational State Transfer	10
Selecting Appropriate REST-based Applications	14
Framework Constraint Prism	15
REST Constraints	18
Architectural Characteristic Matrix	19
Origin Servers	19
User Agents	40
Libraries and Frameworks	70
Constructing RESTful Application Architectures	83
Discussion	89
CHAPTER 2: Dissonance: Applications	96
mod_mbox	96
Subversion	98
Lessons learned	106
CHAPTER 3: Dissonance: Web Services	108
Web Services: Approaches	109
Web Services: Examples	110
Web Services: SOAP	110
Web Services: “RESTful” Services	113
Web Services: Observations	116
Cookies	117
AJAX	119
Mashups	123
CHAPTER 4: CREST: A new model for the architecture of	
Web-based multi-agency applications	126
Recap of Lessons Learned	126
Continuations and Closures	128
Computational Exchange	129
A Computational Exchange Web	131
CREST Axioms	135
CREST Considerations	141
CREST Architectural Style	150
CHAPTER 5: Revisiting Dissonance with CREST	151
Explaining mod_mbox	151

Explaining Subversion	152
Explaining Web Services	154
Explaining Cookies	155
Explaining AJAX and Mashups	156
Evaluation	157
CHAPTER 6: CREST Framework and Experience	159
Example Application: Feed Reader	160
Design Considerations: Feed Reader	163
Additional Related Work	167
CHAPTER 7: Conclusion	169
Recap / Summary	169
Future Work	171
Future Work: Recombinant Web Services	171
REFERENCES	175
APPENDIX A: Architectural Characteristics of RESTful Systems	188
APPENDIX B: Selected Feed Reader Widget Source Code	189

## LIST OF FIGURES

	Page	
Figure 1.1	Process view of a REST-based architecture	10
Figure 1.2	RESTful Architectural Constraint Prism	15
Figure 1.3	Origin Server Timeline with Netcraft Market Share Data	19
Figure 1.4	Apache HTTP Server Internal Architecture	25
Figure 1.5	Apache HTTP Server Internal Architecture	26
Figure 1.6	Internet Information Services (IIS5 Compatibility Mode)	36
Figure 1.7	Internet Information Services 6.0 Architecture	36
Figure 1.8	Web Browser Reference Architecture	41
Figure 1.9	Web Browser Timeline	41
Figure 1.10	Mozilla Architecture Breakdown	49
Figure 1.11	Mozilla Milestone 9 Architecture	50
Figure 1.12	Mozilla Concrete Architecture - circa 2004	52
Figure 1.13	Mozilla Architecture	53
Figure 1.14	Mozilla Architecture	53
Figure 1.15	Microsoft Internet Explorer for Windows CE	58
Figure 1.16	Microsoft Internet Explorer Architecture	58
Figure 1.17	Konqueror Architecture	64
Figure 1.18	Safari Architecture	66
Figure 1.19	Form Browser Example	87
Figure 2.1	mod_mbox architecture	99
Figure 2.2	mod_mbox URL structure	100
Figure 2.3	Idealized network architecture for Subversion	102
Figure 2.4	Initial realized architecture	103
Figure 2.5	Batch request architecture	105
Figure 2.6	Successful pipelined network flow	106
Figure 3.1	SOAP example	109
Figure 3.2	REST example	109
Figure 3.3	Google Maps	120
Figure 3.4	Yahoo! Mail - an AJAX application	121
Figure 3.5	Yahoo! Mail process timeline	123
Figure 3.6	AP News + Google Maps Mashup	124
Figure 3.7	Process view of AP News + Google Maps Mashup	124
Figure 4.1	An example of a closure in JavaScript	129
Figure 4.2	The resulting window produced by JavaScript closure	129
Figure 4.3	Example CREST program (in Scheme)	133
Figure 4.4	Example CREST program (in JavaScript)	133
Figure 4.5	Example CREST remote program (computational view)	133
Figure 4.6	Example CREST spawn service (in Scheme)	134
Figure 4.7	Messages exchanged when using spawn-centric service	134
Figure 4.8	Example SPAWN mailbox URL	136
Figure 4.9	Example of dynamic protocol adaptation in CREST	140
Figure 4.10	Word count example with an intermediary	141
Figure 4.11	CREST URL example (expanded)	143

Figure 4.12	CREST URL example (condensed)	143
Figure 6.1	Screenshot of CREST-based Feed Reader application	160
Figure 6.2	Feed Reader Architecture	161
Figure 6.3	Feed Reader Computations (overview)	161
Figure 6.4	Feed Reader Computations (detail)	162
Figure 6.5	Weak CREST Peer	163
Figure 6.6	Exemplary CREST Peer	163



## LIST OF TABLES

		Page
Table 1.1	Summary of REST constraints in terms of domain and induced properties	13
Table 1.2	REST Architectural Constraints	18
Table 1.3	REST Architectural Constraints: NCSA httpd	23
Table 1.4	REST Architectural Constraints: Early Apache HTTP Server	30
Table 1.5	REST Architectural Constraints: Apache HTTP Server	34
Table 1.6	REST Architectural Constraints: IIS	40
Table 1.7	REST Architectural Constraints: Mosaic	44
Table 1.8	REST Architectural Constraints: Early Netscape	48
Table 1.9	REST Architectural Constraints: Mozilla and Firefox	56
Table 1.10	REST Architectural Constraints: Internet Explorer	63
Table 1.11	REST Architectural Constraints: Konqueror	65
Table 1.12	REST Architectural Constraints: Safari	70
Table 1.13	REST Architectural Constraints: libwww	72
Table 1.14	REST Architectural Constraints: libcurl	75
Table 1.15	REST Architectural Constraints: HTTPClient	77
Table 1.16	REST Architectural Constraints: neon	79
Table 1.17	REST Architectural Constraints: serf	83
Table 3.1	Generic ROA Procedure	114
Table 3.2	HTTP Cookie Example	117
Table 4.1	Core CREST axioms	135
Table 4.2	Summary of CREST Considerations	142

## ACKNOWLEDGEMENTS

I have been blessed to stand on the shoulders of giants. Without each of them and their lasting influences, I would just be a sliver of who I am today and will be tomorrow.

Thanks to Tim for helping me to discover my true calling; thanks to Dan and Jerry for showing me that I could make a living doing this computer stuff; thanks to Andy for expanding my horizons.

Thanks to Steve and Domingos for introducing me to the joys of teaching.

Thanks to Roy for introducing me to the incredible world of open source and academia. Thanks to Aaron who was my fellow traveler in the early days of our collective Apache experiences. The rest of the gang at eBuilt - Josh, Phil, Mark, Seth, Steve, Neil, Bill, Joe, Eli - you set the standard by which I've measured all other teams.

Thanks to Greg, Ryan, Roy, Jeff, Jim, Bill, Sander, Cliff, Aaron, Brian, Manoj, Paul, Colm, Rudiger and all of the other Apache HTTP Server developers. There is simply no better collection of developers that will ever assemble together on a project.

Thanks to Aaron, Bill, Ben, Bertrand, Brian, Brett, Dirk, Doug, Geir, Greg, Hen, Henning, Jim, Ken, Sam, Shane, Stefano, Ken, Ben - you've made serving on Apache's Board of Directors a blast.

Thanks to Greg for so many late nights of conversation and being the better half of Serf.

Thanks to Karl, Jim, Fitz, Ben, C-Mike, Greg, Brane, Sander, Garrett, Hyrum, and the other Subversion developers. We set out to replace CVS and we've done so much more.

Thanks to Martijn, Sander, Garrett, Jens, Paul, Ben, Pier, Allan, Andy, Thom,

Brane, Leo, Matt, Jason, Mike and Janus - we had a tremendous ride together at Joost. Let's do it again sometime.

Thanks to Ken, Neno, Peyman, Jim, Roy, and Rohit for setting the bar so ridiculously high before I even entered the halls of academia. Thanks to Eric, Girish, Jie, and John for the laughs and showing me how to survive the world of being a graduate student. Thanks to Scott for getting me through those early dog days when we were still trying to find our way. Thanks to Joe for being a constant source of sunshine through the years.

Thanks to Yuzo for always being a friend and a source of wisdom. Thanks to Kari who has probably forgotten that she helped out with my very first project in this group years ago. Thanks to Debi for all that you do behind the scenes. Thanks to Kiana, Nancy, Steve, Laura, Jessica and all of the folks who have kept ISR humming over the years.

Thanks to Debra who has shown me how to juggle a million tasks and keep everyone content. Thanks to Walt for helping me to take a step back and see the larger picture when I just couldn't find it.

Thanks to Michael - if our paths didn't cross, this dissertation and research wouldn't be half as interesting. All of our passionate discussions and conversations have resulted in something far better than either one of us could ever have dreamed.

Thanks to Dick for always knowing what buttons to push and when to push them. In all my travels, I have never found anyone else who trusts another (let alone graduate students!) enough to let them find their own way even if we question what everyone else (including ourselves!) takes for granted. You have set an example that I will always strive to live up to every day.

Thanks to the American taxpayer! This material is based upon work supported by the National Science Foundation under Grant Numbers 0438996 and 0820222. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

# CURRICULUM VITAE

Justin Ryan Erenkrantz

## Education

Doctor of Philosophy (2009)

University of California, Irvine  
Donald Bren School of Information and Computer Sciences  
Department of Informatics  
Advisor: Dr. Richard N. Taylor  
Dissertation: *Computational REST: A New Model for Decentralized, Internet-Scale Applications*

Master of Science (2004)

University of California, Irvine  
Donald Bren School of Information and Computer Sciences  
Major Emphasis: Software

Bachelor of Science (2002) *Cum Laude*

University of California, Irvine  
Information and Computer Science

## Professional Experience

1/2007-                 Senior Software Engineer, Joost, Leiden, The Netherlands  
1/2006-1/2007        Engineering Intern, Google, Mountain View, California  
9/2002-9/2009        Graduate Student Researcher, Institute for Software Research,  
University of California Irvine  
4/2000-3/2002        Junior Software Engineer, eBuilt, Irvine, California  
7/1998-9/1999        Seasonal Associate, Ingram Micro, Santa Ana, California  
7/1996-5/1998        C++ Programmer, Renaissance Engineering, Dayton, Ohio

## Publications

### Refereed Journal Articles

- [1] "Architecting trust-enabled peer-to-peer file-sharing applications", by Girish Suryanarayana, Mamadou H. Diallo, Justin R. Erenkrantz, Richard N. Taylor. ACM Crossroads, vol. 12, issue 4, August 2006.
- [2] "An Architectural Approach for Decentralized Trust Management", by Girish Suryanarayana, Justin R. Erenkrantz, Richard N. Taylor. IEEE Internet Computing, vol. 9, no. 6, pp. 16-23, November/December, 2005.

### Refereed Conference and Workshop Publications

- [3] "From Representations to Computations: The Evolution of Web Architectures", by Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, Richard N. Taylor.

- Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 255-264, Dubrovnik, Croatia, September 2007.
- [4] "Architectural support for trust models in decentralized applications", by Girish Suryanarayana, Mamadou H. Diallo, Justin R. Erenkrantz, Richard N. Taylor. Proceedings of the 28th International Conference on Software Engineering, pp. 52-61, Shanghai, China, May 2006.
- [5] "ArchEvol: Versioning Architectural-Implementation Relationships", by Eugen Nistor, Justin R. Erenkrantz, Scott A. Hendrickson, André van der Hoek. Proceedings of the 12th International Workshop on Software Configuration Management, Lisbon, Portugal, September 5-6, 2005.
- [6] "PACE: An Architectural Style for Trust Management in Decentralized Applications", by Girish Suryanarayana, Justin R. Erenkrantz, Scott A. Hendrickson, Richard N. Taylor. Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture, Oslo, Norway, June, 2004.
- [7] "Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community", by Justin R. Erenkrantz, Richard N. Taylor. Proceedings of the 1st Workshop on Open Source in an Industrial Context, Anaheim, California, October, 2003.
- [8] "Release Management Within Open Source Projects", by Justin R. Erenkrantz. Proceedings of the 3rd Workshop on Open Source Software Engineering, Portland, Oregon, May 2003.
- [9] "Beyond Code: Content Management and the Open Source Development Portal", by T.J. Halloran, William L. Scherlis, Justin R. Erenkrantz. Proceedings of the 3rd Workshop on Open Source Software Engineering, Portland, Oregon, May 2003.

#### **Invited Publications**

- [10] "Rethinking Web Services from First Principles", by Justin R. Erenkrantz, Michael Gorlick, Richard N. Taylor. Proceedings of the 2nd International Conference on Design Science Research in Information Systems and Technology, Extended Abstract, Pasadena, California, May 2007.

#### **Non-Refereed Publications**

- [11] "Architectural Styles of Extensible REST-based Applications", by Justin R. Erenkrantz. Institute for Software Research, University of California, Irvine, Technical Report UCI-ISR-06-12, August 2006.
- [12] "Web Services: SOAP, UDDI, and Semantic Web", by Justin R. Erenkrantz. Institute for Software Research, University of California, Irvine, Technical Report UCI-ISR-04-3, May 2004.

## **Service**

Program Committee, HyperText 2004  
Program Committee, WoPDaSD 2008  
Program Committee, OSS 2009  
Webmaster, ICSE 2006  
Internet Chair, ICSE 2011

## **Awards and Honors**

Member, Phi Beta Kappa Society  
Member, Golden Key National Honor Society  
UC Irvine Outstanding Service by an Undergraduate  
UC Irvine Campuswide Honors Program  
Julian Feldman Scholarship Recipient  
Dan and Jean Aldrich Scholarship Nominee  
Member, National Honor Society

## **Associations and Activities**

Member, Association for Computing Machinery (1998-present)  
Member, IEEE Computer Society (2005-present)  
Member, The Apache Software Foundation  
Director, The Apache Software Foundation (2005-present)  
President, The Apache Software Foundation (2007-present)  
Treasurer, The Apache Software Foundation (2005-2007)  
Vice-President, ACM, UC Irvine student chapter (2000-2001)  
Contestant, ACM Programming Contest, UC Irvine student chapter (1998-2000)  
Member, Undergraduate Computing Facility (1998-2002)  
Representative, Associated Graduate Students, UC Irvine (2003-2004)

## **ABSTRACT OF THE DISSERTATION**

Computational REST: A New Model for Decentralized, Internet-Scale Applications

By

Justin Ryan Erenkrantz

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2009

Professor Richard N. Taylor, Chair

REpresentational State Transfer (REST) guided the creation and expansion of the modern web. The reformations introduced with REST permitted the web to achieve its goal as an internet-scale distributed hypermedia system. Yet, the web has now seen the introduction of a vast sea of shared and interdependent services. Despite the expressive power of REST, these new services have not consistently realized the anticipated benefits from REST.

In order to better understand the unwritten axioms necessary to realize these anticipated benefits, we survey the history and evolution of the web's infrastructure - including Apache HTTP Server, Firefox, and Squid. We also recount our experiences developing such systems and the challenges we faced due to the lack of thorough design guidance. We then critically examine these new services from the vast sea - including Service-oriented architectures, RESTful Web Services, and AJAX - to glean previously undocumented lessons about how these services are constructed and why they do not consistently realize the benefits expected from REST.

Based on this, this dissertation presents a new architectural style called Computational



REST (CREST). This style recasts the web from a model where content is the fundamental measure of exchange to a model where computational exchange is the primary mechanism. This crucial observation keys a number of new axioms and constraints that provide new ways of thinking about the construction of web applications. We show that this new style pinpoints, in many cases, the root cause of the apparent dissonance between style and implementation in critical portions of the web's infrastructure. CREST also explains emerging web architectures (such as mashups) and points to novel computational structure. Furthermore, CREST provides the necessary design guidance to create new web applications which have not been seen before. These applications are characterized by the presence of recombinant services which rely upon fine-grained computational exchange to permit rapid evolution.

# INTRODUCTION

**Background.** In 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 those early days, the web evolved chaotically and, to the extent there was any guidance at all, it was weakly focused on client/server communications, the mechanics of content transfer, and user interfaces. However, within a space of just a few years, exponential growth and numerous design flaws threatened the future of the early Web - leading to the introduction of the Representational State Transfer architectural style (REST) [Fielding, 2000].

As a software architectural style, REST introduces a set of principal design decisions that ideally governs the Web. In return for following these architectural decisions, certain properties are expected. The introduction of REST, by way of the HTTP/1.1 protocol, restored comparative order to the Web by articulating the necessary axioms that must be present to permit the continued growth of the web. Yet, as we illustrate in this dissertation, dissonance with any architectural style inevitably occurs in the real-world as certain design decisions dictated by the style are violated either knowingly or unknowingly. The impact of the violations may range from relatively minor to severe. In the presence of such violations, it is rational to expect that specific advantages that should be derived from the style can be lost.

It is our view that the software designer's proper role is to be able to articulate the drawbacks of such dissonance and to be able to knowingly choose which tradeoffs are acceptable against the ideal set forth by the style. This dissertation is our attempt to catalog the continued evolution of the web with two aims: identifying how real systems have strayed

from REST and articulating what specific tradeoffs were made along the way even if the original designers did not consciously realize which tradeoffs were being made. From these observations, we are able to identify a recurring theme in the dissonance: REST was primarily focused on the exchange of content and does not provide suitable guidance to designers participating in today's Web. Again, this is unsurprising as REST was intended to serve as an architectural style for distributed hypermedia systems. However, as we discuss throughout this thesis, today's Web has evolved far beyond the mere exchange of content. We believe that this calls for a new architectural style which better addresses the exchanges we see and shall continue to see on the Web.

More specifically, REST characterizes and constrains the macro interactions of the active elements of the web: servers, caches, proxies, and clients. This reformation permitted the Web to scale to its existing heights today where there are almost 240 million websites [Netcraft, 2009] and over 1.5 billion users [Miniwatts Marketing Group, 2009]. However, REST is silent on the architecture of the individual participant; that is, the components, relationships, and constraints within a single active participant. Therefore, as we will discuss, designers looking to construct such participants (such as servers, caches, proxies, and clients) were forced to independently relearn vital lessons that were neither properly articulated nor centrally captured. Moreover, a new threat to the Web arose as rampant experimentation and unforeseen Web applications threatened to rollback the improvements and reformations introduced with REST and HTTP/1.1. We initially began exploring the hypothesis that to maintain the fidelity of REST's principles at the level and scalability of the web requires previously unspecified constraints on both the architecture of those individual participants and the system-level architecture of the web itself. As

described in this dissertation, this investigation led us to a radical reconception of the web which can serve to characterize and guide the next step in the web's evolution.

**Research Question.** Throughout this dissertation, we can phrase our motivating question as: *What happens when dynamism is introduced into the Web?* In this characterization, we define dynamism as phenomena that must be explained as a manifestation of change, whether through interpretation or alteration of the interpreter. As we discover through our investigations of this question in this dissertation, we ultimately find that *the underlying architecture of the Web shifts, from a focus on the exchange of static content to the exchange of active computations.*

**Insights.** In order to come to that conclusion, we surveyed in detail a number of key web-based infrastructure systems (see Chapter 1 starting on page 8 - including among others: Apache HTTP Server, Mozilla Firefox, and libwww). In Chapter 2 starting on page 96, we draw upon our personal experience as developers struggling to build dynamic web applications. In Chapter 3 starting on page 108, we then turn to emerging web services to evaluate how they handle dynamism. From these investigations, we have drawn the following insights:

- Web Infrastructure (discussed in Chapter 1 starting on page 8): With little explicit coordination among developers during this period, critical web infrastructure applications evolved rapidly to support dynamism - both architectural and content-focused.
- Mobile code (discussed in Chapter 1 starting on page 8 as well as Chapter 4 starting on page 126): Due to the improvements in the JavaScript engines, the modern browser is far more powerful and capable today than it was in the mid-'90s. Distributed mobile code systems can be built on top of existing Web infrastructure.

- `mod_mbox` (discussed in Chapter 2 starting on page 96): Resources (as denoted by an URL) can represent more than just static content and can refer to various 'representations' generated on-the-fly to suit a particular domain (in this instance, web-based access to very large mail archives).
- `subversion/serf` (discussed in Chapter 2 starting on page 96): Decoupling communication and representation transformations internally within a user-agent's architecture can minimize latency, reduce network traffic, and improve support for caching. Also, it is feasible to deploy protocol-level optimization strategies that do not conflict with REST or existing protocols (in this case, many fewer requests were needed for the same operations). However, extreme care must be taken to not violate documented protocol expectations so as not to frustrate intermediaries (such as caching proxies).
- SOAP-based Web Services (discussed in Chapter 3 starting on page 108): Offering fine-grained services is unquestionably a noble goal as it promotes composition and first-party and third-party innovation. However, due to implementation deficiencies (such as lack of idempotency support and improper intermingling of metadata and data in SOAP messages), SOAP-based Web Services (and its sibling 'Service Oriented Architectures') are incapable of realizing the promise of fine-grained, composable services without fundamentally violating the REST axioms that permitted the web to scale. SOAP-based Web Services, if widely adopted, would rollback the scalability improvements introduced with HTTP/1.1.
- RESTful Web Services (discussed in Chapter 3 starting on page 108): A lack of clear design guidance for construction of "RESTful" services that are not closely tied to 'content' services (such as adding or removing documents) make for often madden-

- ingly inconsistent and incomplete interfaces and, even though they handle the bulk of traffic for “web services,” RESTful Web Services have yet to reach their full potential.
- Cookies (discussed in Chapter 3 starting on page 108): Even relatively minor alterations to a protocol may grant a 'toehold' to wide-spread architectural dissonance. The alternatives at the time were not fully articulated; therefore, web site designers took the easy (and better articulated through Netscape’s developer documentation) approach. This failure violated the statelessness interaction axiom of REST and raises inconsistencies with the semantics of the “back” button featured in today’s browsers.
  - AJAX and Mashups (discussed in Chapter 3 starting on page 108): AJAX and mashups illustrate the power of computation, in the guise of mobile code (specifically code-on-demand), as a mechanism for framing responses as interactive computations (AJAX) or for “synthetic redirection” and service composition (mashups). No longer must 'static' content to be transported from an origin server to a user agent - we now transfer 'incomplete' representations accompanied by domain-specific computations applied client-side to reify the 'content' with which a user interacts. The modern browser has matured into a capable execution environment - it can now, without the help of third-party “helpers,” piece together XML and interpret JavaScript to produce sophisticated client-side applications which are low latency, visually rich, and highly interactive. Additionally, AJAX-based “mashups” serve as the computational intermediary (proxy) in an AJAX-centric environment.

**CREST Architectural Style.** In order to support this shift, we construct a new architectural style called CREST (Computational REST). As discussed in much more detail in Chapter 4 starting on page 126, there are five core CREST axioms:

- CA1. A resource is a locus of computations, named by an URL.
- CA2. The representation of a computation is an expression plus metadata to describe the expression.
- CA3. All computations are context-free.
- CA4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.
- CA5. The presence of intermediaries is promoted.

**CREST Considerations.** We also encounter several recurring themes that must be addressed by a computation-centric Web and are related to the axioms above:

- Computations and their expressions are explicitly *named*. (CA1, CA2)
- *Services* may be exposed through a variety of URLs which offer perspectives on the same computation. (CA1); interfaces may offer complementary supervisory functionality such as debugging or management. (CA4)
- Functions may be added to or removed from the binding environment over *time* or their semantics may change. (CA4)
- Computational loci may be *stateful* (and thus permit indirect interactions between computations), but must also support *stateless* computations. (CA3)
- Potentially autonomous *computations* exchange and maintain state (CA2, CA3); A rich set of stateful relationships exist among a set of distinct URLs. (CA1)
- The computation is *transparent* and can be inspected, routed, and cached. (CA5)
- The *migration* of the computation to be physically closer to the data store is supported thereby reducing the impact of network *latency*. (CA2)

These themes are discussed in more detail in Chapter 4 starting on page 126.

**Explaining Dynamism in the Web.** The shift to supporting dynamism on the web is but an example of a more general form—network continuations—the exchange of the representations of the execution state of distributed computations. It is the presence and exchange of continuations among web participants, in their various forms, that induces new constraints among and within participants. With this in mind, as discussed in Chapter 5 starting on page 151, both prior complications in the structure of individual clients and recent elaborations of the web such as AJAX or mashups are accounted for by a single fundamental mechanism: network continuations as web resources—an insight codified in the principles of CREST.

**CREST Framework.** To both facilitate the adoption of CREST and to explore the implications and consequences of the style, we have constructed a CREST framework (discussed in detail in Chapter 6 starting on page 159) that allows us to build applications in this style. Utilizing this framework, we have constructed a feed reader application which offers novel computational and compositional aspects.

**Contributions.** In summary, this dissertation provides the following contributions:

- analysis of the essential architectural decisions of the World Wide Web, followed by generalization, opens up an entirely new space of decentralized, Internet-based applications
- recasting the web as a mechanism for computational exchange instead of content exchange
- a new architectural style to support this recasting (CREST)
- demonstrating how CREST better explains architectural dissonance
- a framework for building applications backed by CREST



# CHAPTER 1

## **Architectural Styles of Extensible RESTful Applications**

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 fifteen years of real-world architectural evolution from which to base our examinations. Our aim in this chapter 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.

### **1.1. 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, 1996]. 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 [Suryanarayana, 2006], 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.

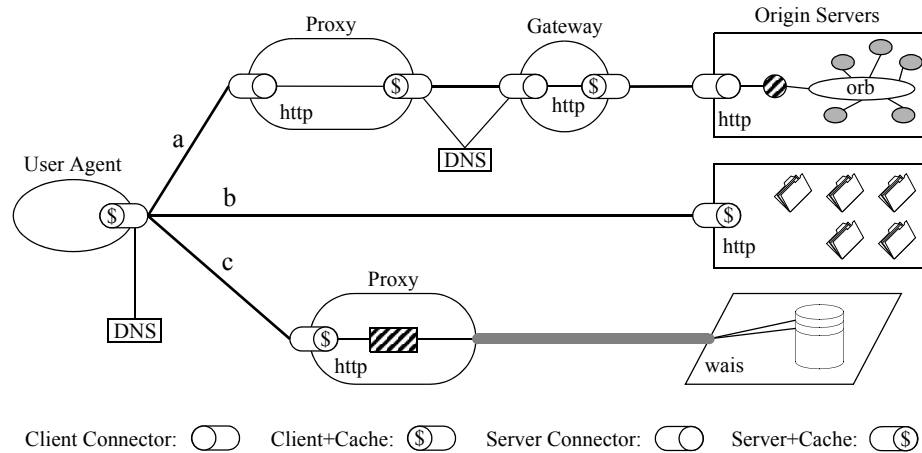
An architecture framework is software that helps to bridge the gap between a specific architectural style (or family of styles) and an implementation platform (e.g., programming language, core set of libraries, or operating system). This makes it easier for application developers to correctly (and compatibly) 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. As we will examine in this chapter, this is a natural tradeoff: frameworks may provide little support but be very lightweight, or be heavyweight and complex but provide many services.

## **1.2. 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



**Figure 1.1: Process view of a REST-based architecture at one instance of time (From [Fielding, 2002])**

Fielding's diagram of the 'process view' of a REST-based architecture is presented in Figure 1.1 on page 10. We will now introduce the REST architectural style. After introducing REST, we will begin selecting systems to survey.

### 1.3. Representational State Transfer

The Representational State Transfer (REST) architectural style minimizes latency and network communication while maximizing the independence and scalability of component implementations [Fielding, 2002]. 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. There are six core REST design axioms:

### **1.3.1. The key abstraction of information is a resource, named by an URL.**

Any information that can be named can be a resource: a document or image, a temporal service (e.g., “today's weather in Amsterdam”), a collection of other resources, a moniker for a nonvirtual object (e.g., a person), and so on.

### **1.3.2. The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes.**

Hence, REST introduces a layer of indirection between an abstract resource and its concrete representation. Of particular note, the particular form of the representation can be negotiated between REST components.

### **1.3.3. All interactions are context-free.**

This is not to imply that REST applications are without state, but that each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

### **1.3.4. Only a few primitive operations are available.**

REST components can perform a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST—for instance, all resources exposed via HTTP are expected to support each operation identically.

### **1.3.5. Idempotent operations and representation metadata are encouraged in support of caching.**

Caches are important to the goal of reducing latency. The metadata included in requests and responses permits REST components (such as user agents, caching proxies) to make

sound judgements of the freshness and lifespan of representations. Additionally, the repeatability (idempotence) of specific request operations (methods) permits representation reuse.

### **1.3.6. The presence of intermediaries is promoted.**

Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the client and the origin server.

Building upon these axioms, REST relies upon specific aspects of the distributed hypermedia domain and adds specific constraints in order to induce a set of properties. These are presented in Table 1.1 on page 13. It should be noted that while HTTP/1.1 is an instantiation of REST, it does suffer from mismatches with REST. One of these mismatches, the use of cookies, is discussed at length in Section 3.6 on page 117.

**Table 1.1: 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	<b>Metadata</b> describing <b>representation</b> 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 ( <b>Extensible methods</b> )	User Agent can talk to any Origin Server
A document may have multiple valid depictions with differing metadata	Distinction between abstract <b>resource</b> & transferred <b>representation</b> Metadata can be sent by user agent that indicates preferences ( <b>Internal transformation</b> )	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)	Even if direct paths are not available between two nodes, indirect paths may be available through REST intermediaries
	<b>Proxy</b> node (User Agents)	
	No assumption of persistent connection or routing; Hop-by-hop only Any <b>state</b> 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
	<b>Cacheability</b> 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

## 1.4. Selecting Appropriate REST-based Applications

Due to the ubiquitous deployment of the World Wide Web, there are plenty of critical systems which we can examine. In order to limit our selection, we constrained our selection to those systems which reside at the level closest to the protocol—that is, they directly implement the HTTP/1.1 protocol. Leveraging REST’s classifications, we will divide our discussion into three main categories: origin servers, user agents, and frameworks. However, as we will discuss, some origin servers also fulfill the responsibilities of a proxy or gateway. With respect to the specific selections we make in each category, we will attempt to choose a representative sample of the broad range of 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 extensibility is predicated on the diverse nature of web applications. The architectures we survey provide the glue by interfacing with the larger Web through protocol implementations and pass along the constraints of the Web on to its extensions to form complete applications driven by end-users.

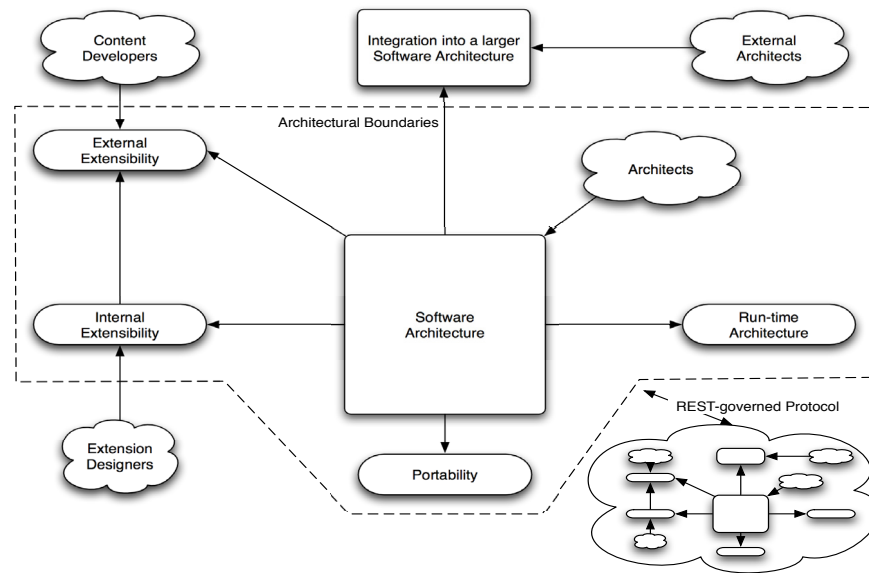
A vast range of applications have emerged that use the WWW in innovative ways - ranging from electronic-commerce sites to collaborative news sites. The specific content requirements often differ for each individual application. Instead of constructing an origin server or user agent from scratch each 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)
- Client-side scripting languages (such as JavaScript)
- HTML forms

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.

## 1.5. Framework Constraint Prism



**Figure 1.2: 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 archi-



tures is shown in Figure 1.2 on page 15. 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.

## 1.6. REST Constraints

We earlier presented a summary of the domain aspects, REST constraints, and induced behavior in Table 1.1 on page 13. 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 1.2 on page 18.

**Table 1.2: 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	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?

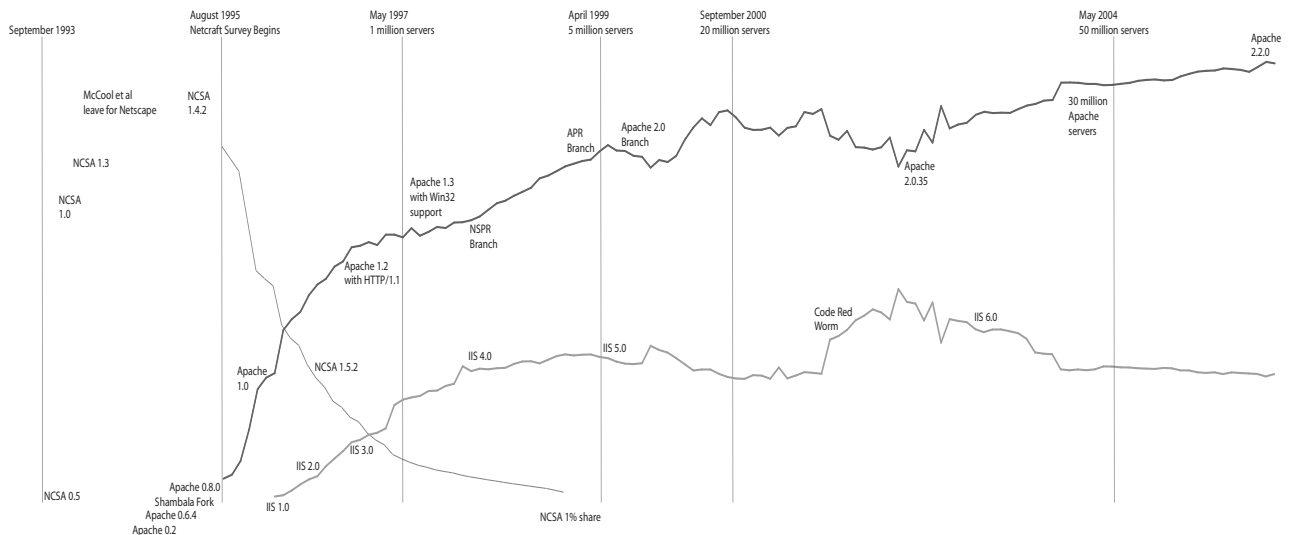
## 1.7. Architectural Characteristic Matrix

A matrix summarizing all of the architectural characteristics of these selected systems is presented in Appendix A.

## 1.8. Origin Servers

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, 5.2.3]



**Figure 1.3: Origin Server Timeline with Netcraft Market Share Data from [Netcraft, 2009]**

In common usage on the web, this is characterized by an HTTP server. Figure 1.3 on page 19 presents a timeline of market share as determined by Netcraft's Web Server Survey ([Netcraft, 2009]) for the three origin servers we will now discuss:

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

### **1.8.1. NCSA HTTP Server**

One of the early origin servers for the Web was produced at the National Center for Super-computing Applications (NCSA) at the University of Illinois at Urbana-Champaign[Kwan, 1995]. 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 at NCSA. After McCool left to join Netscape in 1994, NCSA development largely ceased with a few later ultimately unsuccessful efforts by NCSA personnel 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, 1994]. 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, 1994][Kwan, 1995].

**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, 1999]. 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 84.

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 execution 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] Therefore, CGI script authors were advised to hold the execution 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 1.3: 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.

### **1.8.2. 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][Mockus, 2000], 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 1.3 on page 19, 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, 2009], and has been the market leader for over nine years [The Apache Software Foundation, 2004]. 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]

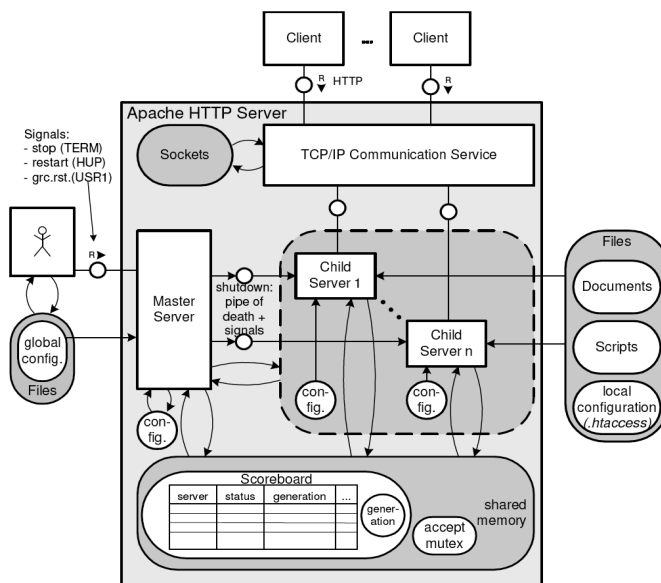
Due to its lineage from NCSA httpd codebase, there are a lot of surface similarities between the two codebases. In stark contrast to NCSA httpd, however, the internals of the Apache HTTP Server are characterized by an extremely modularized design with almost all aspects of functionality available to 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.

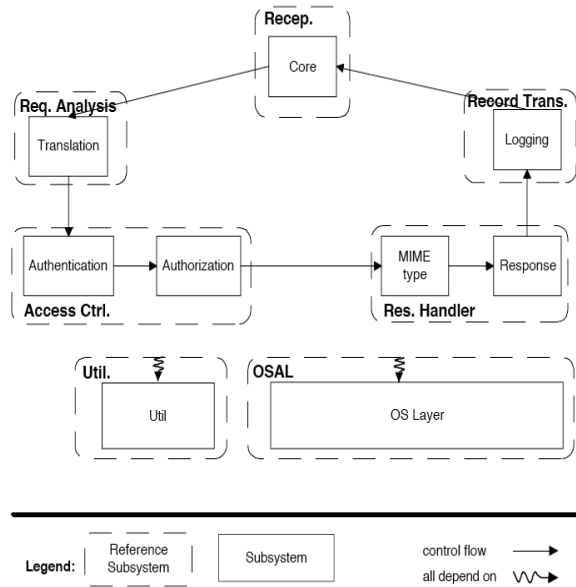
The principal mechanisms behind this re-architecture were introduced in the “Shambala” fork by Robert S. Thau. These changes were merged into the mainline Apache codebase to

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]. 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 1.4 on page 25[Gröne, 2002][Gröne, 2004]. Hassan and Holt present another description of Apache’s architecture in Figure 1.5 on page 26[Hassan, 2000].



**Figure 1.4: Apache HTTP Server Internal Architecture (From [Gröne, 2004] 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



**Figure 1.5: Apache HTTP Server Internal Architecture (From [Grosskurth, 2005])**

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.

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 remaining 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][The Apache Software Foundation, 2003]. 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].

**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]. 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 accommodated 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 1.4: 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] - 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]

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 noticeably 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 primitives 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].

**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]. 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 1.5: 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

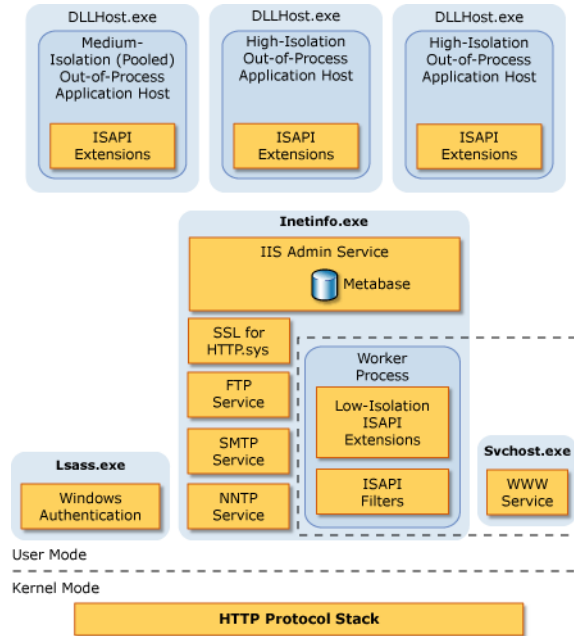
*Lessons learned.* Portability concerns led to a new portability layer and new run-time architecture policy layer; the absence of an internally extensible RESTful representation required the shoe-horning of filters; early design assumptions of where a node fits in the overall REST architecture challenged as the system evolves; use in different server-based applications.

### **1.8.3. Microsoft Internet Information Server (IIS)**

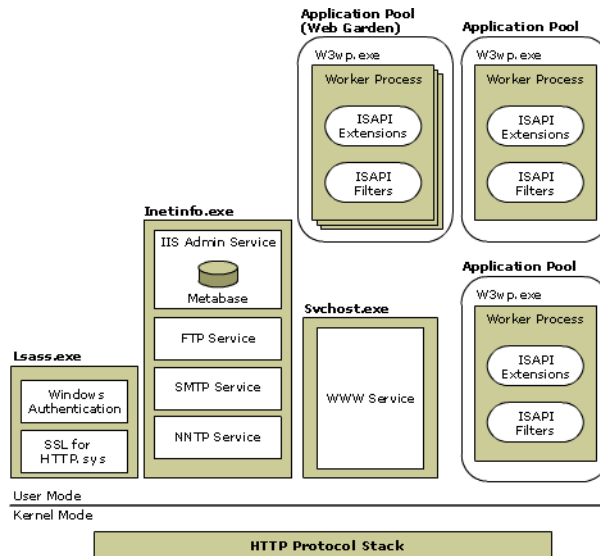
Microsoft first released their HTTP server named Internet Information Server (IIS) in February, 1996. This initial version of IIS was only available on Windows NT 3.51. Over time, it was updated to work on newer releases of the Windows platform. The early releases of IIS featured basic HTTP and FTP serving 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[Moore, 2002][Cook, 2005].

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



**Figure 1.6: Internet Information Services (IIS5 Compatibility Mode) (From [Microsoft Corporation])**



**Figure 1.7: Internet Information Services 6.0 Architecture (From [Microsoft Corporation])**

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] [Wang, 2005]. 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]

**Run-time architecture.** IIS presents the administrator with two run-time architectural models to choose from. Depicted in Figure 1.6 on page 36 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] 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] [Web Host Industry Review, 2001].

To increase reliability, IIS 6.0 introduces a new run-time architectural option called *Worker Process Isolation Mode*, depicted in Figure 1.7 on page 36. This model defines a collection of *application pools* that are assigned to a specific web site - a fault in one web site will only jeopardize the application pool it resides in. [Microsoft Corporation] 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]

**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. ISAPI modules would be executed inside the server process not outside the server process like CGI [Schmidt, 1996]. 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]. 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]

- 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]. 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] Any requests to these inactive services, even if they are otherwise installed, will result in an HTTP error being returned to the user.

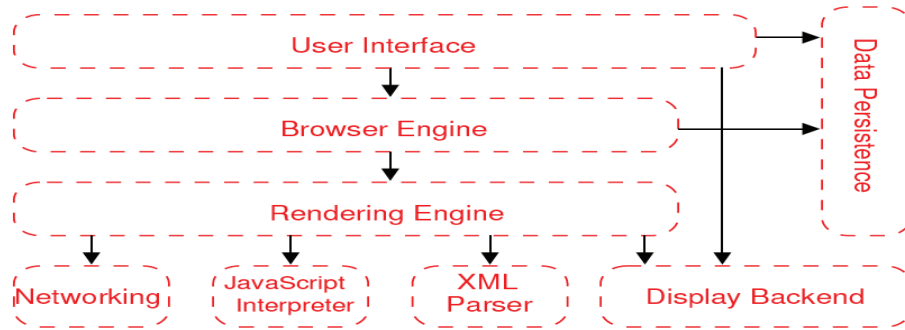
**Table 1.6: REST Architectural Constraints: IIS**

Constraint	Imposed Behavior
Representation Metadata	Request: Fetch request header with ‘:’ Response: Add headers with manual delimiting[Microsoft Corporation, 2004]
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

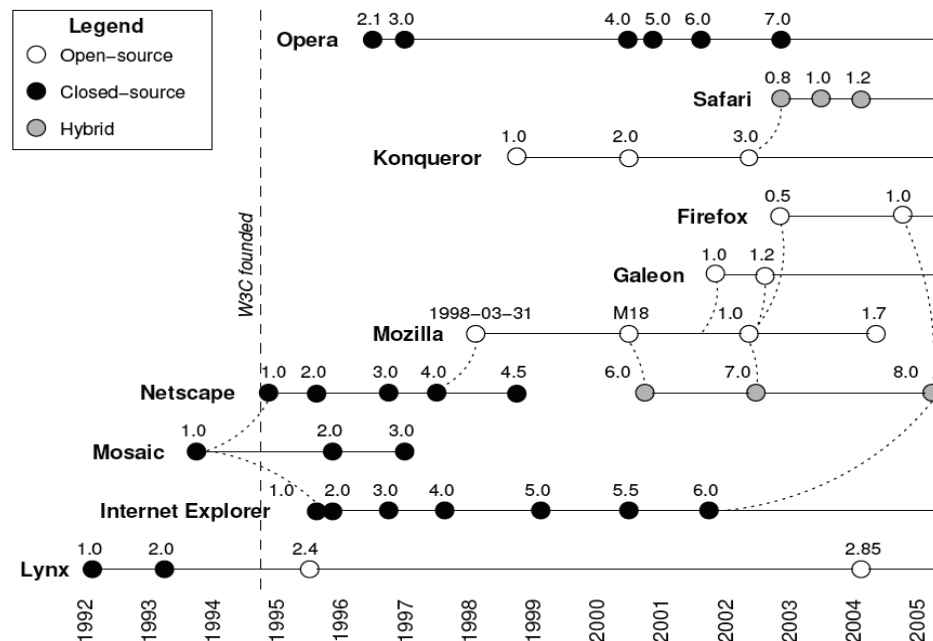
*Lessons Learned.* Lack of separation in run-time architecture presented serious security risks; Installing and activating unnecessary components by default can be dangerous; Security constraints can restrict range of functionality that can be provided.

## 1.9. User Agents

Fielding defines a user agent as:



**Figure 1.8: Web Browser Reference Architecture (From [Grosskurth, 2005])**



**Figure 1.9: Web Browser Timeline (From [Grosskurth, 2005])**

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, 5.2.3]

Grosskurth and Godfrey used automated architectural recovery processes to define a reference architecture for web browsers depicted as Figure 1.8 on page 41[Grosskurth, 2005]. Besides producing a reference architecture, they also presented a timeline that covers the

early history of web browsers as depicted in Figure 1.9 on page 41. In an earlier work, Grosskurth and Echiabi extracted software architectures for several other web browsers[Grosskurth, 2004]. 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.

### **1.9.1. 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]. 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, 1995].

#### ***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, 1994]. 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][Schatz, 1994]. 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] Other work with CCI would allow control of the user interface within Mosaic[Newmarch, 1995]. Notably, CCI did not allow for drawing of unknown media types within the Mosaic windows.

**Table 1.7: 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 competitor - 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] 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, 1996].

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] 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] 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] 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].

**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]:

- 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]. 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 88.

The other key feature that Netscape Navigator introduced was cookies.[Netscape, 1999][Kristol, 1997] As discussed in Section 3.6 on page 117, 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 1.8: REST Architectural Constraints: Early Netscape**

Constraint	Imposed Behavior
Representation Metadata	Requests: Metadata represented as content[Oeschger, 2002] 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]:

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:

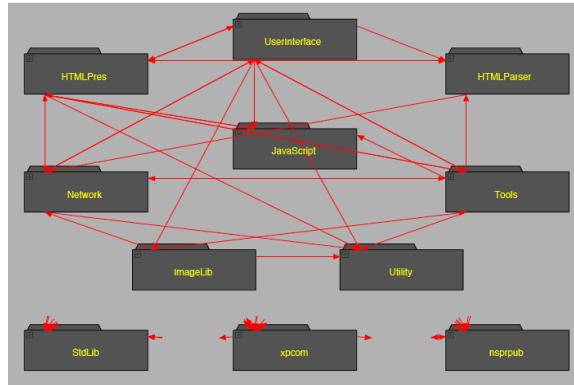
**They decided to rewrite the code from scratch.**

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]. 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.

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

**Figure 1.10: Mozilla Architecture Breakdown (From [Godfrey, 2000])**

**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, 2000]. Milestone 9 was first released to the public in August 26, 1999 and represented a web browser, mail client, news reader, and chat



**Figure 1.11: Mozilla Milestone 9 Architecture (From [Godfrey, 2000])**

engine in one integrated application[Mozilla Foundation, 2002]. Mozilla’s aim with Milestone 9 was to introduce a new networking layer called Necko[Mozilla Foundation, 2001]. 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, 1999]

Godfrey and Lee’s recovered architecture diagram for Mozilla Milestone 9 is presented in Figure 1.11 on page 50. A breakdown of Mozilla’s architectural systems is presented Table 1.10 on page 49. 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, 2000]. 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.

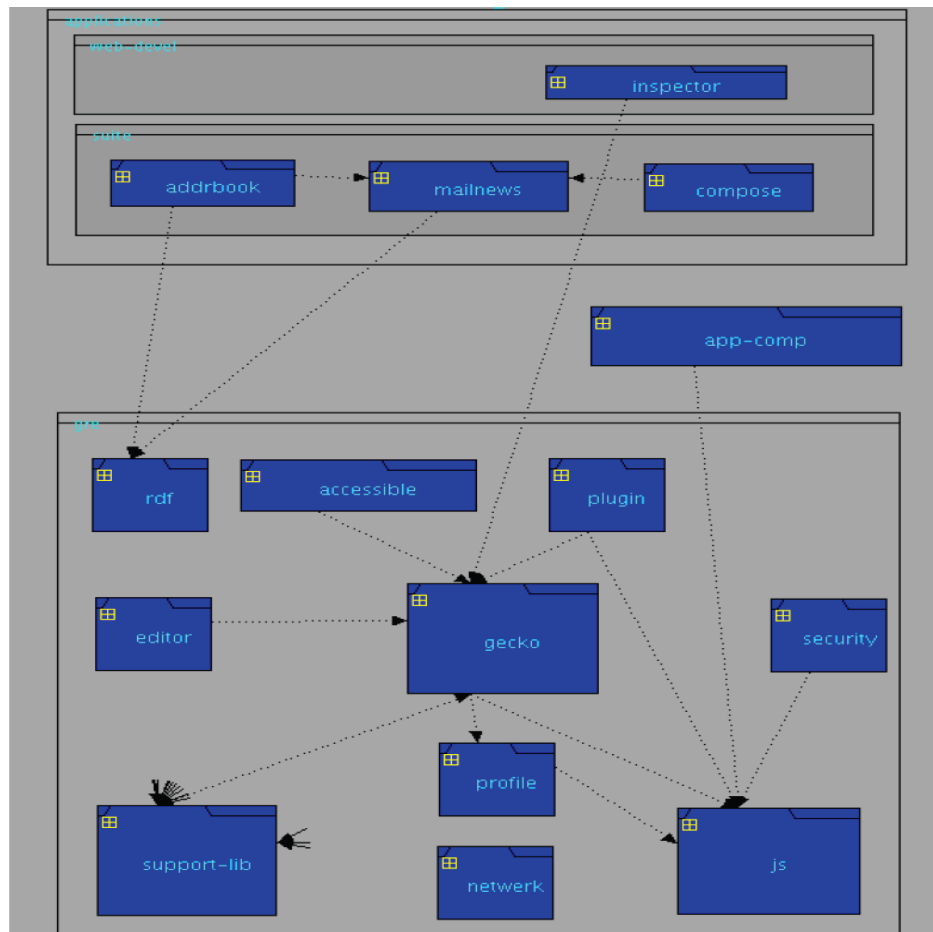
**Analysis of Mozilla's Architecture.** Godfrey summarized their architectural observations about Mozilla that “either its architecture has decayed significantly in its relatively short lifetime, or it was not carefully architected in the first place”[Godfrey, 2000]. As can be seen in the figure, the 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]:

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 question-

ing: 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.



**Figure 1.12: Mozilla Concrete Architecture - circa 2004 (From [Grosskurth, 2004])**

*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.

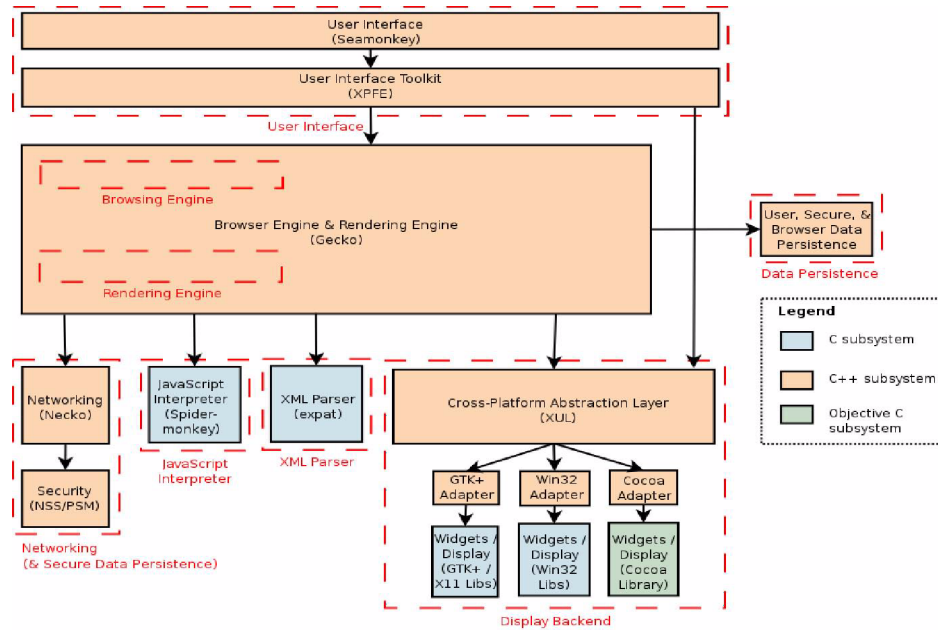


Figure 1.13: Mozilla Architecture (From [Grosskurth, 2004] Figure 8)

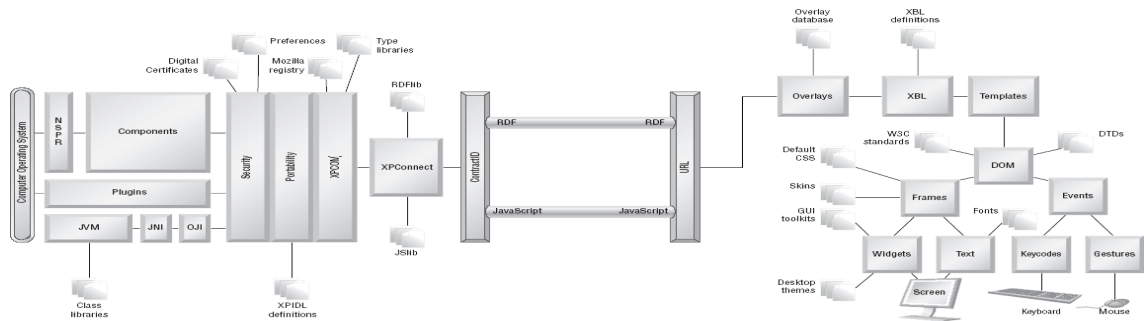


Figure 1.14: Mozilla Architecture (From [McFarlane, 2003])

### CURRENT MOZILLA ARCHITECTURE

In 2004, Grosskurth and Echiabi returned to Mozilla to assess the architecture's progress [Grosskurth, 2004]. By this time, Mozilla had launched a spin-off product called Firefox. Firefox differed from Mozilla in that it 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 Echiabi's resulting concrete architecture is presented in Figure 1.12 on page 52. They also fit this recovered Mozilla architecture into a reference architecture for

web browsers as presented in Figure 1.13 on page 53. 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 1.14 on page 53[McFarlane, 2003].

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 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]. 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]

**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, 2003]. 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]. Building upon XPCOM with user interface extensions like XPFE[Trudelle, 1999], 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].

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, 2004]. 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]. 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]. 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]. 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]

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]. Therefore, Gecko-derived applications tend to be variants on Mozilla but are not functionally much different than Mozilla.

**Table 1.9: 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

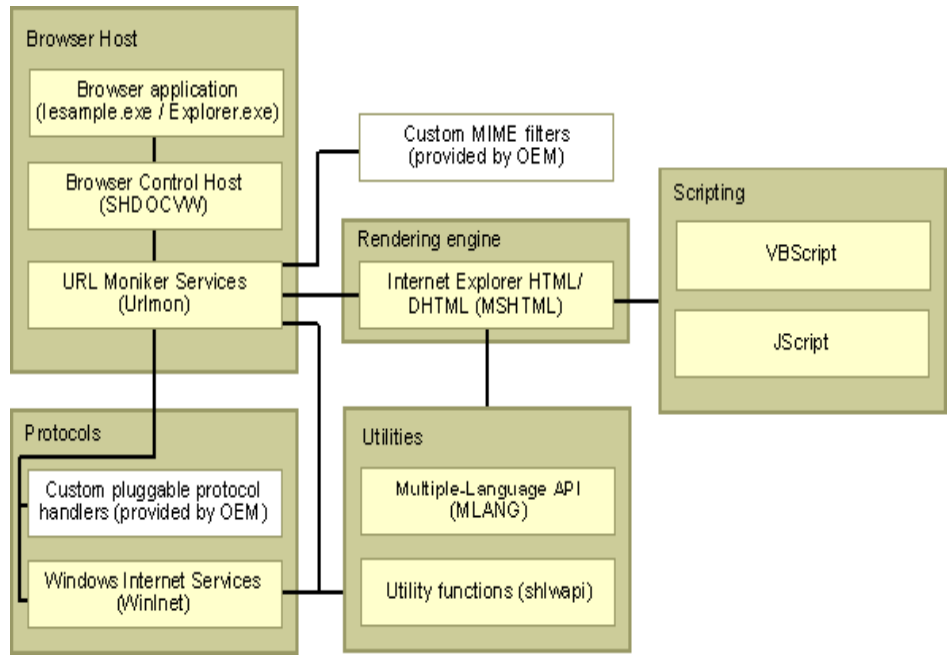
### **1.9.2. Microsoft Internet Explorer**

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

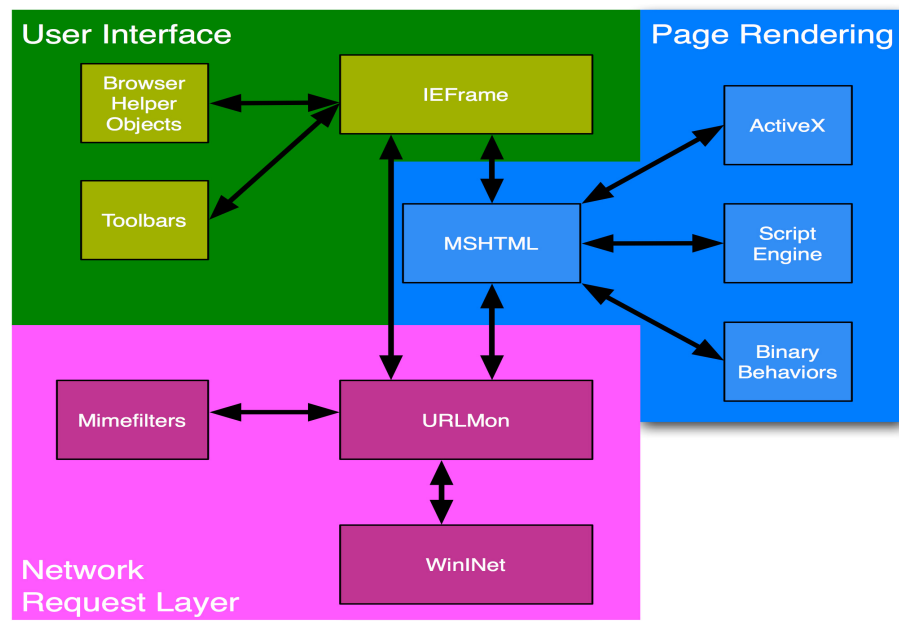
browser from a company called Spyglass. Spyglass, in turn, was the commercial variant of NCSA Mosaic for Windows-based platforms. However, in the years since the first release of Internet Explorer, the corresponding code base and architecture greatly evolved to the point where it now has little architectural 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 1.15 on page 58, is a public architectural description of Microsoft Internet Explorer for Windows CE available on the Microsoft Developer Network (MSDN) website.[Microsoft Corporation, 2005] Another source for architectural information about Internet Explorer is contained within recent presentations



**Figure 1.15: Microsoft Internet Explorer for Windows CE (From [Microsoft Corporation, 2005])**



**Figure 1.16: Microsoft Internet Explorer Architecture (Adapted from [Chor, 2005])** given by Microsoft’s Internet Explorer development team discussing the impact of security on IE’s architecture.[Chor, 2005] This particular architectural representation of Internet Explorer is reproduced in Figure 1.16 on page 58.

**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]. 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]

In Internet Explorer 5.0, this functionality was broadened to allow modification by COM objects and events[BowmanSoft, 2001]. 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]. 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]

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].

Through the URLmon component, developers can also register an *asynchronous pluggable protocol* to register a new protocol or MIME filter [Microsoft Corporation]. 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 initiates 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]. 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].

**External Extensibility and Run-time Architecture.** The run-time architecture of Internet Explorer is presented in Figure 1.16 on page 58. 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] 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]. 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] 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]. 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]. The main problems identified were that various architectural flaws and deficiencies com-

bined 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]

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] 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 1.10: 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.

### **1.9.3. Konqueror**

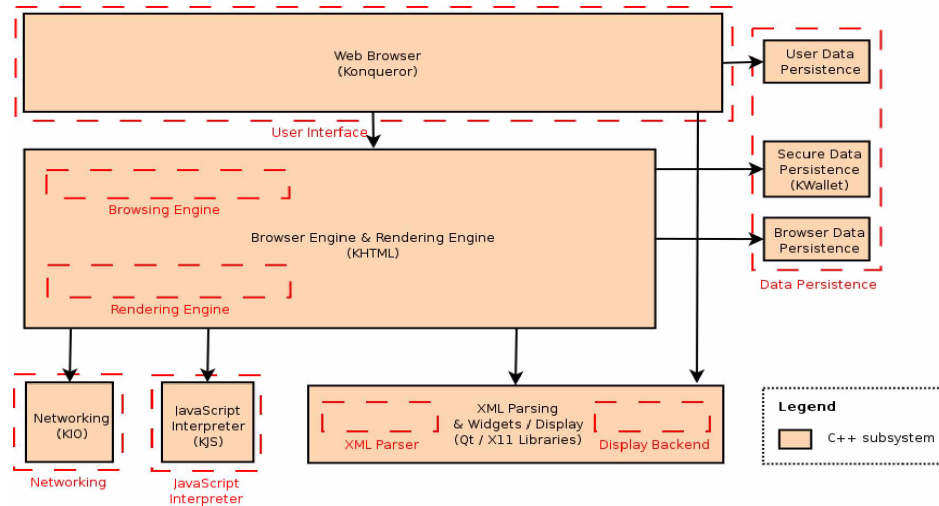
Konqueror from the K Desktop Environment project (KDE) is one of the few user agents that can not trace its heritage to the NCSA Mosaic codebase. Instead, Konqueror evolved from the file manager for the KDE environment. The Konqueror name itself is a subtle reference to the other browsers. The KDE developers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

Konqueror's architecture, as extracted by Grosskurth and Echiabi, is presented in Figure 1.17 on page 64.



**Figure 1.17: Konqueror Architecture (From [Grosskurth, 2004] Figure 12)**

**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 environment. 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 repre-

sentation (such as for HTML, JavaScript, and CSS). Through a Konqueror plugin and operating system emulation, Konqueror can also support ActiveX controls.[KDE e.V., 2001] 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]. Through KParts, a developer can render media elements inside the Konqueror window[Granroth, 2000]. 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.

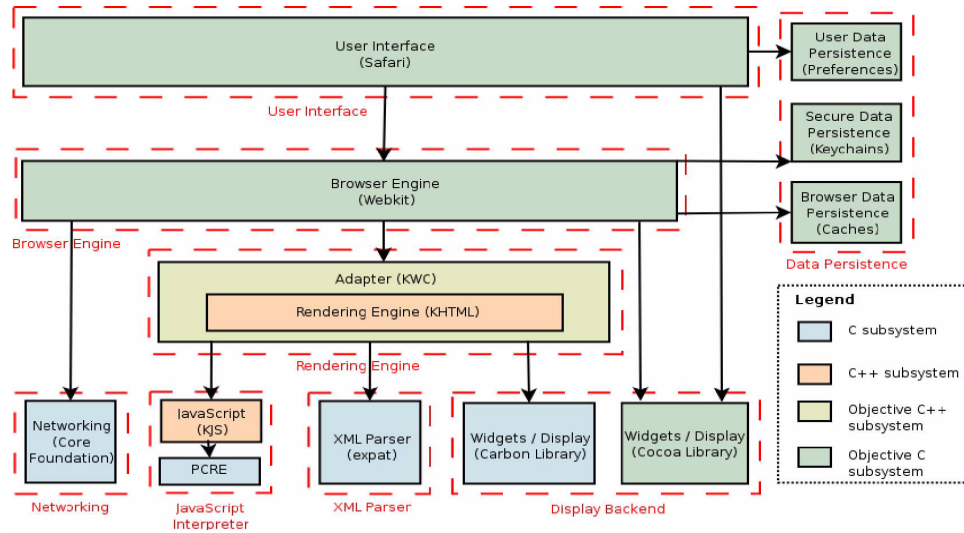
**Table 1.11: REST Architectural Constraints: Konqueror**

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

**Table 1.11: REST Architectural Constraints: Konqueror**

Constraint	Imposed Behavior
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 1.18: Safari Architecture (From [Grosskurth, 2004] Figure 14)**

### 1.9.4. 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]. 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 1.18 on page 66, 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]

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]. 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]

**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 primitives 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.”[Shaver, 2003][Baker, 2003]

**Internal Extensibility.** Safari supports two kinds of extensibility mechanisms: Netscape-compatible plug-ins and WebKit[Apple Computer Inc., 2005]. 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]

On top of support for Netscape plug-ins, Safari also offers a set of extensions in Objective-C called WebKit[Apple Computer Inc., 2005][Apple Computer Inc., 2005]. Extensions written for WebKit allows use of Apple’s bundled development tools for easy construction.[Apple Computer Inc., 2005] 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] Contrary to statements which have stated otherwise, the iTunes player on Mac OS X does not use WebKit.[Hyatt, 2004] Since Web-

Kit 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 1.12: 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.

## 1.10. 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

necessities for these applications. Again, we will limit ourselves to the criteria presented in “Selecting Appropriate REST-based Applications” on page 14.

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.

### **1.10.1.libwww**

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

**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]. 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

---

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



be found in Chapter 7 of Bass [Bass, 1998]. 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]. 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] 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 assumptions throughout the framework that raise serious challenges when designing applications with libwww for other REST connector types.

**Table 1.13: 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

**Table 1.13: REST Architectural Constraints: libwww**

Constraint	Imposed Behavior
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.

### **1.10.2.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]. 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].

**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 programming 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 transfer files through URLs and supports all of the underlying protocols that libcurl supports. A selection of applications that use libcurl include[Stenberg, 2006]:

- 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 multi-platform 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 1.14: 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.

### 1.10.3.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] 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].

**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]:

- 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 1.15: 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
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.

#### 1.10.4. 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]. Web Distributed Authoring and Versioning (WebDAV) is an official extension of HTTP which facilitates multiple authors collaboratively editing REST resources[Whitehead, 1998][Goland, 1999][Clemm, 2002]. 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] 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 addition 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]. 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 1.16: REST Architectural Constraints: neon**

Constraint	Imposed Behavior
Representation Metadata	Request: Add metadata fields to request structure Response: Register callbacks for specific metadata names



**Table 1.16: REST Architectural Constraints: neon**

Constraint	Imposed Behavior
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.

### **1.10.5.Serf**

Serf is an HTTP client framework that is inspired by the Apache HTTP Server's design[Stein, 2004]. 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 dynamically and in a specific order. This name can trace its origins through Apache HTTP Server[Woolley, 2002] and, from there, back to the libwww-ada95 library[Fielding, 1998]. In turn, this layering concept is related to Unix STREAMS[Ritchie, 1984].

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]

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].

We believe that serf can resolve these performance problems and remove the need for custom methods solely for performance reasons[Fielding, 2005].

**Table 1.17: 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

## 1.11. 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.

### **1.11.1.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]

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] 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 execution 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] 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:

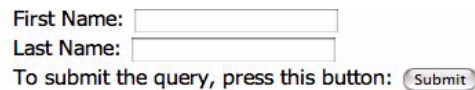
The extra path information, as given by the client. In other words, scripts can be accessed by their virtual pathname, followed by extra information at the end of this path. The extra information is sent as `PATH_INFO`. This information should be decoded by the server if it comes from a URL before it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

### **1.11.2.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 1.19 on page 87.

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.

```
<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>
```



First Name:   
Last Name:   
To submit the query, press this button:

**Figure 1.19: Form Browser Example (HTML snippet and screenshot)**

**Deployment.** NCSA Mosaic for X 2.0, released in November 1993, was one of the first browsers to support FORM tags.[Andreessen, 1993][National Center for Supercomputing Applications, 1999] A specification of forms was first included in HTML+ announced in November 1993 as well[Raggett, 1993]. 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.

### **1.11.3.JavaScript**

As discussed in “Early Netscape Navigator Architecture” on page 45, 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 control the behavior of the browser.[Champeon, 2001] 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, 2003] As mentioned with “Current Mozilla Architecture” on page 53, 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] 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] Earlier works such as KnowNow's JavaScript-based micro-servers presaged this later work.[Udell, 2001][Khare, 2004] AJAX and mashups are discussed in more detail in Section 3.7 on page 119.

## **1.12.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.

### **1.12.1.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 cre-

ators, 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.

### **1.12.2. 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 concep-

tually 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 filters*). 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.

### **1.12.3.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.

#### **1.12.4.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. Retrofitting 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.

#### **1.12.5. 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 re-architectures. 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, 2009]. Mozilla Firefox has made improvements in its market share in the last year by try-

ing 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]



## CHAPTER 2

### Dissonance: Applications

Despite their apparent simplicity, the REST principles have proven difficult to understand and apply consistently, leaving many web systems without the benefits promised by REST, including sites that do not support caches or require stateful interactions. To better understand why, we now recount our personal experiences building and repairing two web systems: `mod_mbox`, a service for mail archiving, and `Subversion`, a configuration management system.

#### 2.1. `mod_mbox`

As core contributors to the open-source Apache HTTP Server, we required a scalable, web-based mail archive to permanently record the design rationales and decisions made on the project mailing lists. In addition to the lists of the Apache HTTP Server Project (which include developer-to-developer traffic, user-to-user assistance, automated source change notifications, and issue tracker notices), we wished to deploy a mail archive capable of archiving all of the lists across the entire Apache Software Foundation. In total, there are over 650 lists, some of which receive over 1,000 messages a month and have been active for over a decade. To date, the archive houses over 5 million messages and receives nearly 2 million requests per month. From the outset, it was clear that we would need to explicitly consider scalability as a primary factor in order to support an archive of this size and traffic.

### **2.1.1. Alternative mail archivers**

At the time we started the project in the middle of 2001, no archivers fully met our requirements. Some systems, such as MHonArc [Hood, 2004], converted each incoming message into HTML and recomputed the archive and list index as new messages arrived. This constant index regeneration proves problematic for high-traffic archives - the time to recompute the index and regenerate the HTML for all messages in the archive (usually split out into monthly segments) often takes longer than it would for new messages to arrive in peak hours. Therefore, an alternative strategy would be to delay the generation of the HTML until absolutely necessary - this follows RA2 in creating dynamic representations of the mailing list archive. Additionally, it would be important to introduce a cache (RA5) to help facilitate indexing and retrieval to minimize the latency in producing these dynamic representations.

Other web-based archivers, such as Eyebrowse [CollabNet, 2006], generated message links whose ordering reflected the sequence in which messages were loaded into the archive. Therefore, the first message loaded into the global archive was assigned an identifier roughly analogous to 1, the second message received 2, and so on. In order to retrieve the message from the archive, the exposed resource would be in the form of `http://www.example.com/archive/viewMessage?id=50`. However, in the event of a hardware failure or other database corruption, the archive would have to be regenerated and there would be no guarantee of consistency of identifiers between database loads. This inconsistency would mean that any prior links to the archive (such as those from our own code or other emails) would be stale or, even worse, refer to a different message than originally intended. Therefore, it was critical to identify a persistent naming scheme for resource (RA1) that would not invalidate links at a later date.

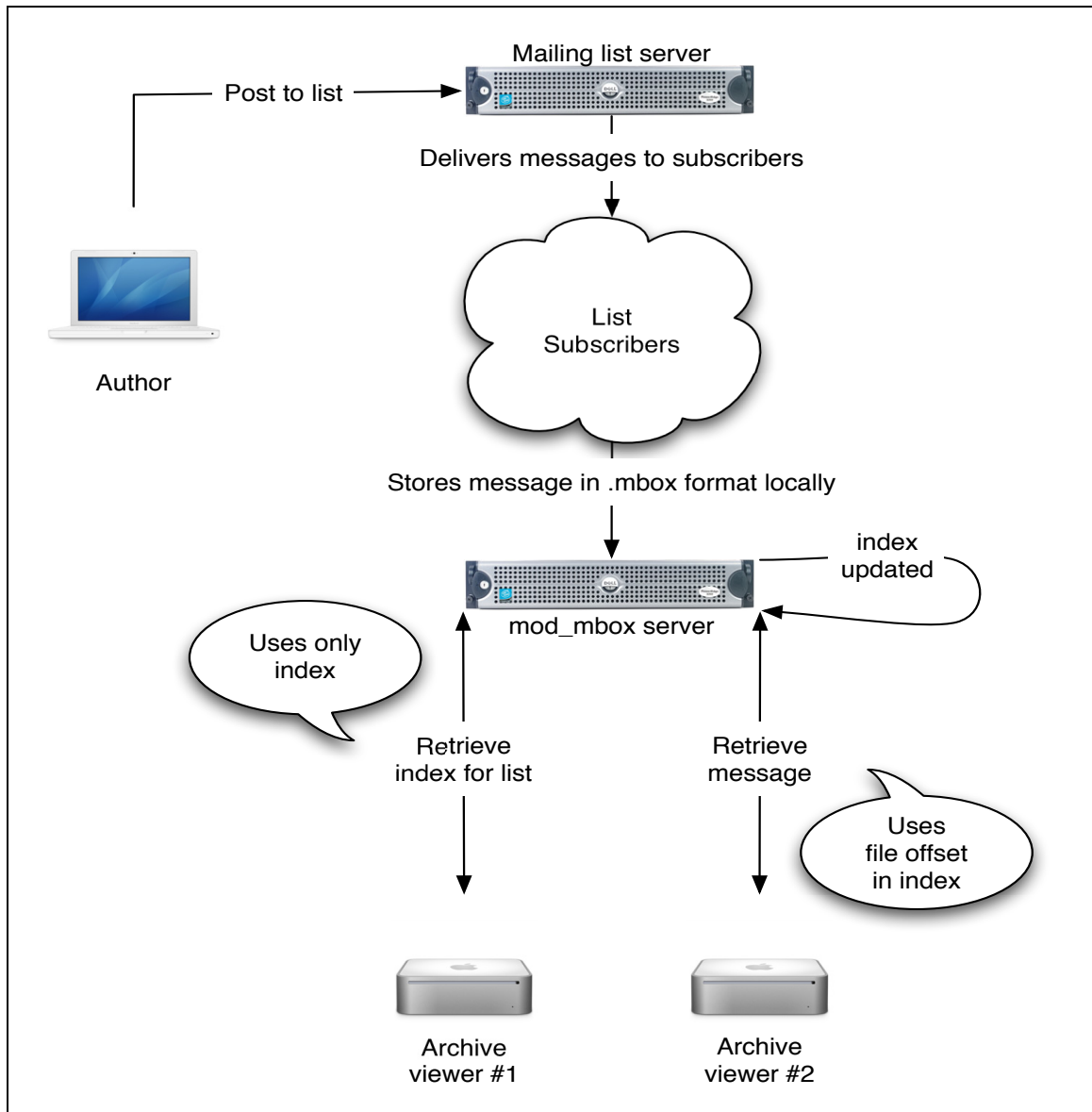
### **2.1.2. mod\_mbox Architecture**

Informed by our experience with the Apache HTTP Server, we were confident that we could create a new web-based archiver, `mod_mbox` [The Apache Software Foundation, 2006], based on the REST principles, that would not suffer from the shortcomings of other archivers. However, as we discovered, REST alone was neither sufficiently expressive nor definitive. `mod_mbox` required two additional constraints beyond those dictated by REST: dynamic representations of the original messages and the definition of a consistent namespace. The architecture of `mod_mbox` is depicted in Figure 2.1 on page 99.

Instead of creating HTML representations as messages arrive, `mod_mbox` delays that transformation until a request for a specific message is received. On arrival, only a metadata entry is created for a message *M*. Only later, when message *M* is fetched from the archive, does an Apache module create an HTML representation of *M* (with the help of *M*'s metadata entry). This sharp distinction between the resource and its representation (RA1, RA2) minimizes the up-front computational costs of the archive—allowing `mod_mbox` to handle more traffic. As depicted in Figure 2.2 on page 100, to achieve a consistent namespace, `mod_mbox` relies upon *M*'s metadata (the Message-ID header). Consequently, if the metadata index is recreated, the URLs of the resources (messages) remain constant—guaranteeing the long-term persistence of links. After adopting these constraints, `mod_mbox` scaled to archive all of the mailing lists for The Apache Software Foundation (in excess of 5 million messages to date) with consistent and persistent links.

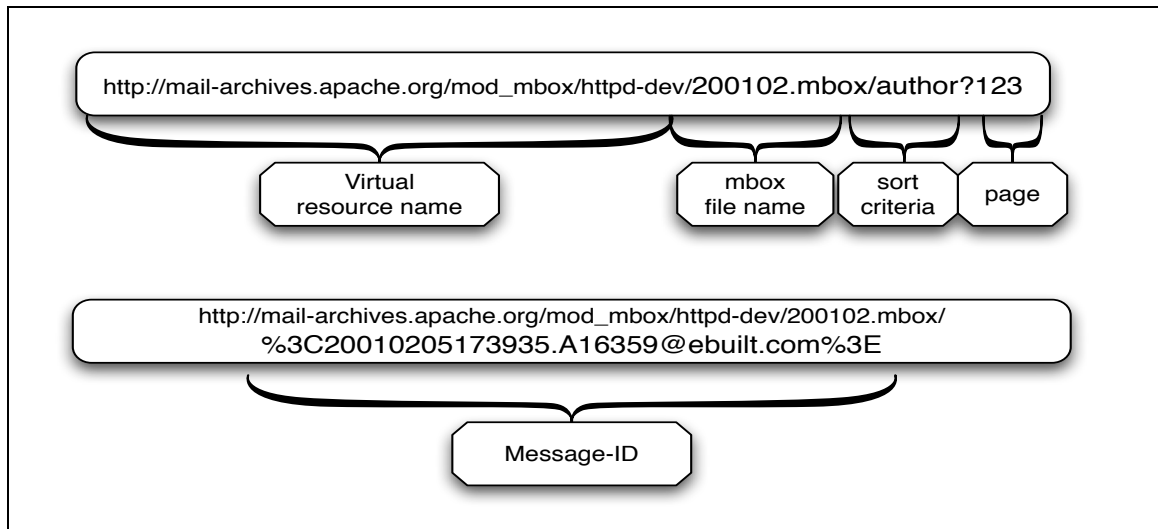
## **2.2. Subversion**

Subversion [CollabNet, 2008], a source code manager designed to be a compelling replacement for the popular CVS, made a decision early on to use WebDAV, a set of



**Figure 2.1: mod\_mbox architecture**

extensions to HTTP, for repository updates and modifications. Hence, by conforming to the REST constraints, the Subversion developers (of which this author is one of the core contributors) expected that Subversion's implementation would be low latency, amendable to caching, and support the easy construction and introduction of intermediaries (RA5, RA4, RA6).



**Figure 2.2: mod\_mbox URL structure**

### 2.2.1. Subversion, WebDAV, and HTTP/1.1

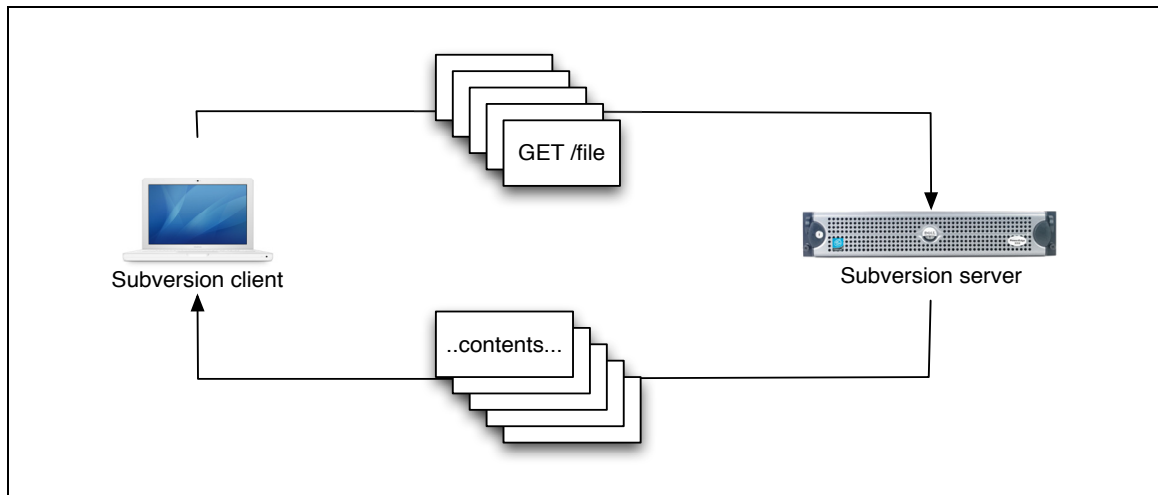
Fundamental to Subversion is a global, monotonically-increasing revision number denoting the state of the repository, hence each file in Subversion is uniquely identified by the combination of the global revision number and the path in the repository. Additionally, in order to permit certain off-line operations and to optimize certain network operations, Subversion requires having a pristine copy of each file locally on the client in a private location. Therefore, each Subversion client when it initially checks out the repository must retrieve an unmodified version of the file, and on subsequent updates, the difference between the last retrieved pristine file and the youngest (latest) version is applied locally. This ensures that the client always has an up-to-date copy and also optimizes later network retrieval by only requiring differentials to be sent rather than the complete full-text content every time.

As part of exposing the repository via WebDAV, the Subversion developers created a set of mappings to permit clients to retrieve specific versions of a file. In Subversion, this decision was made to expose all of the necessary parameters explicitly in the URL space.

Therefore, the file `baz` in the directory `foo/bar` at revision `1234` would be exposed via WebDAV as `/path/to/repos/!svn/ver/1234/foo/bar/baz`. As a shorthand mechanism, the youngest (latest) version would be exposed as `/path/to/repos/foo/bar/baz`.

In order to retrieve a collection of files in this mapping (say all of the files in the `foo/bar` directory), WebDAV and HTTP/1.1 do not directly support fetching multiple files in the same request - therefore, the request must be performed individually for each file or a new custom method must be introduced. The lack of batch requests was acknowledged in the early days of standardizing the HTTP protocol, but was not clearly articulated within REST itself. As a mechanism to reduce the latency effects of many small requests, “pipelining” was introduced. In this mechanism, clients issue multiple HTTP/1.1 requests without waiting for the responses. This process will optimize the network pipeline by not requiring the responses to be fully processed by the client before the next request is issued. Of note, HTTP/1.1 does not support out-of-order responses; therefore, the server must respond serially to the requests. Yet, as we will see, taking full advantage of this pipelining mechanism is crucial when trying to adhere to the REST principles.

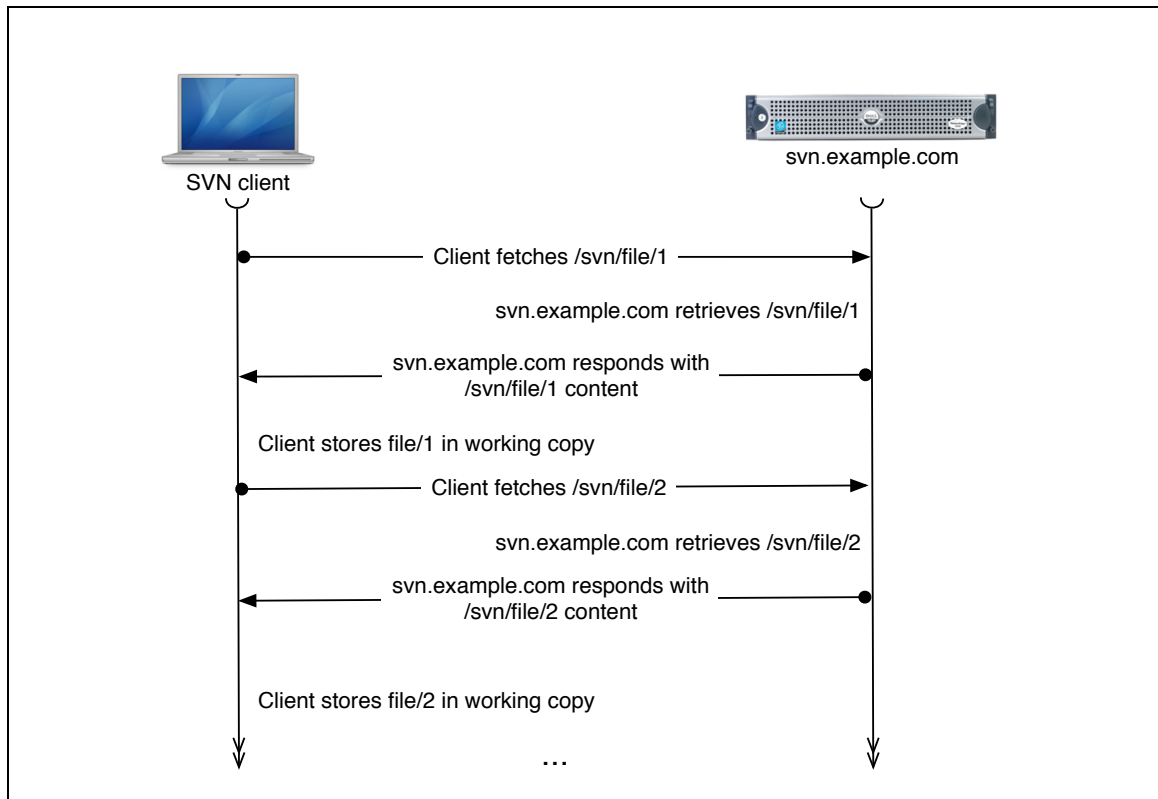
The idealized process demonstrated in Figure 2.3 on page 102 shows the conceptual task that must be performed: the user agent must request all necessary files and the origin server will respond with the content of those files so that the client can store them on disk. The challenge we faced with Subversion is how to perform this task efficiently without compromising the REST principles.



**Figure 2.3: Idealized network architecture for Subversion**

### 2.2.2. Subversion's initial network architecture

The earliest implementation of Subversion utilized the network as depicted in Figure 2.4 on page 103. To achieve the quickest turnaround on implementing a WebDAV client, the Subversion developers adopted the existing `neon` client library to implement the protocol semantics of WebDAV. This library was chosen because it offered most of the functionality necessary to implement a rich WebDAV client. The update functionality in Subversion over `neon` was implemented over one network connection and performed simple `GET` requests of the versioned resources. The use of `GET` requests meant that it was relatively straightforward to introduce a standard caching intermediary which would help reduce the load on the master Subversion server (origin server). However, one notable omission from the `neon` library is that it did not support HTTP/1.1 pipelining. Therefore, all responses must be fully processed before the next request could be issued. Unsurprisingly, this made the checkout/update performance of Subversion unacceptable as the latency characteristics caused by the time that the network pipeline was left idle had a substantial impact on the overall performance.



**Figure 2.4: Initial realized architecture**

This poor network performance can be more generally attributed to not taking into due consideration the physical characteristics of the underlying network connection [Smith, 2009][Khare, 2003]. Per [Smith, 2009], latency can be defined as the propagation delay (time to transmit one bit from source to destination and back again) plus the data unit size (how much data is sent at one time) divided by the bandwidth (how much data can be sent simultaneously). In an application that does not keep the network pipeline constantly active—such as Subversion’s initial design—but rather has substantial time lapses between packet transmissions (since the client must fully process the last response before issuing a new request), network latency can have a substantial impact on the observed performance. In a physically co-located network where the propagation delay can be measured in nano-seconds, the latency effects on the application are minimal. However, when using trans-continental links where the propagation delay is on the order of 40ms or inter-continental



links where propagation delays can reach up to 180ms, latency can have a dramatic effect on the overall application performance. Therefore, upon proper reflection and due consideration of network latency properties, it should not be surprising that the real-world network performance of Subversion was so initially underwhelming.

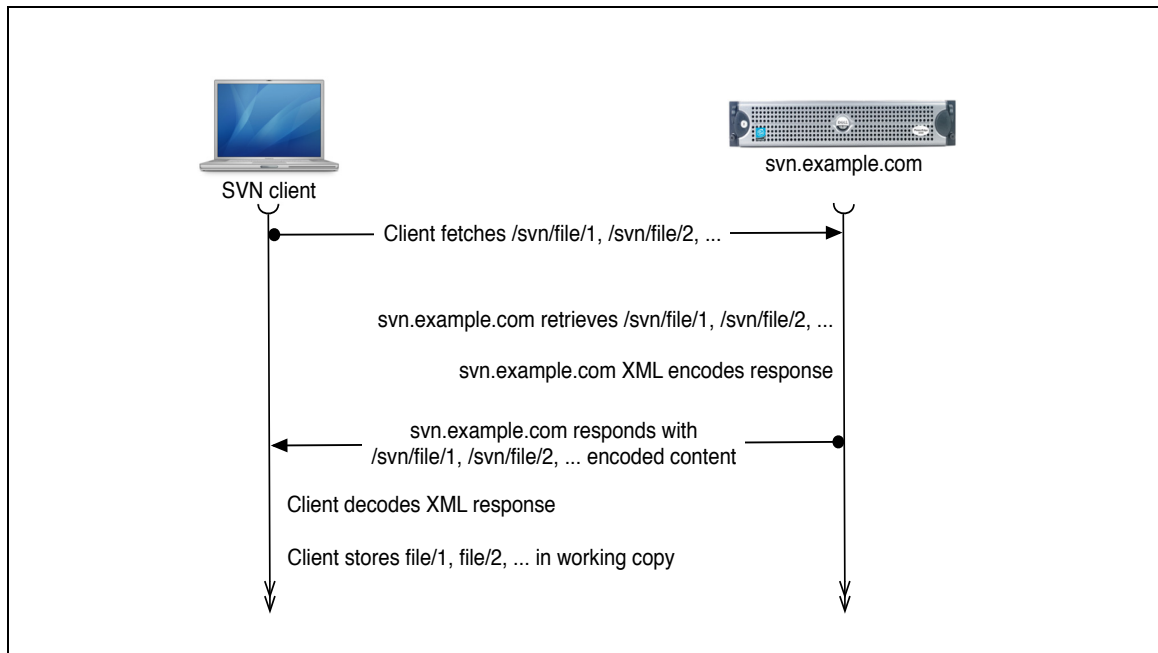
### **2.2.3. Subversion's batch request architecture**

In order to improve serial performance and reduce network roundtrip latency, Subversion developers decided to implement a *custom* WebDAV method.<sup>1</sup> The updated architecture is reflected in Figure 2.5 on page 105. Rather than request each resource individually, the new Subversion client issued a single bulk request for all needed resources at one time. As expected, this change greatly improved reduced the network latency and made the update operation much faster. However, this custom WebDAV method required that all of the data (including the content stored in Subversion—which, of course, could be binary and of arbitrary size) be XML-encoded. XML encoding has been shown to increase transfer volumes by approximately 33% [Nottingham, 2004] as well as being computationally expensive to construct and decode [Davis, 2002]. Therefore, while the introduction of this custom method did improve the overall performance, the introduction of XML encoding introduced substantial inefficiencies of its own into the network.

As a further consequence of adopting a custom method, it was no longer possible to deploy a standard caching intermediary to help reduce the load on the master Subversion server. As a result, a new custom intermediary had to be written and deployed to help reduce the load on Subversion servers [Erenkrantz, 2002].

---

1. Actually, the REPORT method was (re)used which had largely undefined semantics in the WebDAV standard.

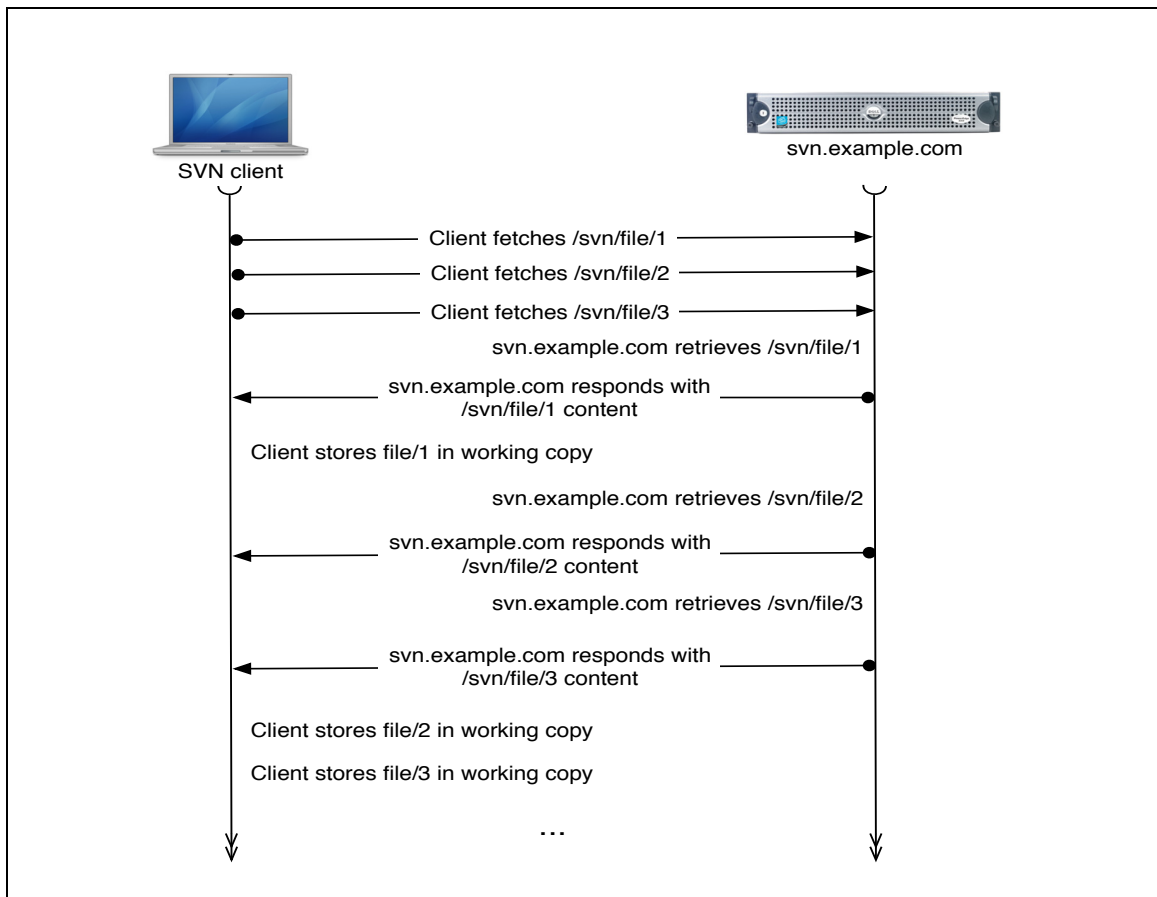


**Figure 2.5: Batch request architecture**

#### 2.2.4. Subversion's revised architecture with `serf`

Due to the inefficiencies presented by the XML encoding and the additional requirement for custom intermediaries, the Subversion developers wrote an alternate client framework, `serf`, replacing the `neon` library previously used. As discussed previously in this chapter, the critical omission from the initial architecture was the `neon` could not support pipelining. Therefore, `serf` would implement a model conducive to supporting pipelining along with asynchronous requests and responses in addition to adhering to the REST principles and constraints. (For more information on `serf`, please see Section 1.10.5 on page 80.) The network architecture for Subversion powered by `serf` is depicted in Figure 2.6 on page 106. In addition to supporting pipelining, `serf` also permitted the Subversion client to multiplex the update process across multiple connections further reducing the impact of network latency. Now that Subversion could take advantages of pipelining, there was little need for the custom WebDAV method and the client could again simply retrieve the content via the `GET` method. Since the client used the standard

HTTP/1.1 protocol semantics again, simple caching intermediaries could be reintroduced into the network without requiring custom intermediaries. Plus, by removing the custom method, the overhead imposed by XML encoding could also be reverted. Additionally, further optimizations were introduced which reduced the number of requests in typical operations without violating any standard semantics or jeopardizing intermediaries. All of these changes allow the new Subversion client to scale gracefully yet adhere to the REST principles.



**Figure 2.6: Successful pipelined network flow**

## 2.3. Lessons learned

As we have seen in this chapter, `mod_mbox` scaled in part because it delayed representation generation until the last possible moment. Consumers of a service are usually better

positioned to specify exactly the representations characteristics that they want rather than having a service provider determining beforehand which characteristics will be presented. Subversion had scaling problems because it did not account for the bandwidth-delay product of network connections and had no provision for optimizing the network pipe. By migrating the computations (in the form of custom methods) from the client to the server, the performance was improved. However, this migration was done in a way that led to other inefficiencies which frustrated intermediaries. Further improvements to Subversion to decouple communication and representation transformations internally minimized latency effects, reduced network traffic, and improved support for caching. Also, Subversion was able to deploy protocol-level optimizations that did not conflict with HTTP/1.1.

## CHAPTER 3

### Dissonance: Web Services

“Web services” have emerged to expose finer-grained computational resources than the coarser-grained content resources of the traditional view of the Web. There are two popular approaches to web services: one that relies upon SOAP and “service-oriented architectures,” and another that conforms to REST principles and “resource-oriented architectures.” We will compare, contrast, and present examples of these approaches. Additionally, we discuss the impact of a relatively minor alteration to a REST-based protocol (cookies) and explore how this minor alteration created architectural dissonance in the Web. Finally, we assess the emergence of computational exchange on the Web in the form of AJAX and mashups.

As we illustrate throughout this chapter, the REST style, by itself, has proven insufficient to guide or characterize the introduction of these services. “Service-oriented architectures” only pay limited lip service to the REST style underpinning the web and more closely represent the antiquated client-server architectural style and is ill-suited in its current form for Internet-scale deployment. Even the “resource-oriented architectures” which claim to conform to REST are required to navigate a set of hidden and under-articulated principles in order to succeed. Despite the knowledge that cookies are harmful, they still perform vital tasks today. And, REST has little to offer to help us understand what role AJAX and mashups play in today’s web. This lack of guidance has resulted in a striking dissonance between the idealized architecture embodied by REST and the realized Web of today.

### 3.1. Web Services: Approaches

Figure 3.1 on page 109 and Figure 3.2 on page 109 compare SOAP-based web services (often referred to as “service-oriented architectures” [Papazoglou, 2003][Colan, 2004]) and “RESTful”-based web services (sometimes referred to as “resource-oriented architectures” [Richardson, 2007]). One clear distinction between the two approaches is their use of resources [Mitchell, 2002]. The “RESTful” HTTP/1.1 request of Figure 3.2 on page 109 leverages resources by providing all of the requisite information in the resource name (requesting `/Reservations/itinerary` with a query string of `id=FT35ZBQ`) while the SOAP request of Figure 3.1 on page 109 adds a level of indirection in naming the resource. This particular SOAP request asks the `/Reservations` resource to invoke the `retrieveItinerary` function with the `reference` argument being set to `FT35ZBQ`. These seemingly inconsequential differences leads to several architectural dissonance points with REST.

```
POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml
Content-Length: ...

<?xml version="1.0" ?>
<env:Envelope...>
  <env:Body>
    <m:retrieveItinerary...>
      <m:reference>FT35ZBQ</m:reference>
    </m:retrieveItinerary>
  </env:Body>
</env:Envelope>
```

**Figure 3.1: SOAP example (modified from Example 12a in [Mitra, 2003])**

```
GET /Reservations/itinerary?id=FT35ZBQ HTTP/1.1
Host: travelcompany.example.org
```

**Figure 3.2: REST example (modified from Example 12b in [Mitra, 2003])**

## **3.2. Web Services: Examples**

There are many examples of both SOAP-based Web Services and RESTful services. In fact, the two are not necessarily exclusive - many sites provide SOAP and RESTful interfaces to their services simultaneously. The primary audience of these interfaces is not end-users interacting through a user-agent, but developers who wish to incorporate a “service” into their own application. Among a plethora of examples, eBay provides services to query current auctions [eBay, 2007], Flickr provides services to upload pictures and tag photos [Yahoo, 2007], and Amazon.com facilitates access to their centralized storage repository, S3, through services [Amazon Web Services, 2006].

## **3.3. Web Services: SOAP**

Tracing its origins from the prior XML-RPC specification [Winer, 1999], Simple Object Access Protocol (SOAP) was introduced as a lightweight “protocol” for exchanging structured information on the web [Gudgin, 2003]. The formulation and development of SOAP-based web services has been a complex undertaking with vigorous discussion within the community as to whether such services are even feasible and how to manage their complexity [Cardoso, 2007][W3C, 2007]. “Service-oriented architectures,” with SOAP as its foundation, have emerged in part as a response to the problems of service exchange [Papazoglou, 2003]. Many specifications and standards were formulated to enable composition, discovery, and evolution of these “service-oriented” architectures [Fu, 2004][Bellwood, 2002][Ryu, 2008], but, as we shall now detail, these standards are not built on a suitable technical underpinning and, if widely deployed, would threaten the scalability of the web.

To be clear, SOAP is not a protocol but a descriptive language expressed via XML transferred via another transport protocol such as HTTP. Alternative protocols such as SMTP can be used with SOAP, but the prevailing use of SOAP is in conjunction with HTTP. To that end, one of the stated reasons for the adoption of HTTP by early SOAP advocates was that by using HTTP, SOAP implementations could get through corporate firewalls easily [Prescod, 2002]. The use of HTTP as a transport mechanism was not made out of any particular understanding or adherence to the REST axioms which governed the HTTP/1.1 protocol. SOAP owes more to the legacy of remote procedure calls (RPC) and remote method invocation (RMI) than that of hypermedia systems in which the web was designed to accommodate [Davis, 2002]. Given this heritage, it is not surprising that the grid community has adopted SOAP-based web services as it is easy to represent the opaque RPC-like services provided by a grid into the SOAP model [Foster, 2002]. However, in practice, SOAP corrupts the integrity of the REST architecture, and most problems due to SOAP can be traced to architectural mismatches with REST and HTTP. Two such problems are elaborated here.

As discussed in Chapter 1, idempotent operations (RA5) are a fundamental REST concept. To recap, per the RFC, certain HTTP methods (such as `GET`) are declared to be idempotent: if a `GET` is performed multiple times on a static resource, the results must be identical; or on dynamic resources, the return representations should be valid for a specific time frame (expressed via expiration metadata). Other HTTP methods (such as `POST`) are non-idempotent: if a `POST` is performed multiple times on the same resource, the side effects are undefined by the HTTP specification.



Therefore, it is possible for a HTTP/1.1 intermediary serving as a caching proxy to always cache the returned representation of any `GET` method while it can not cache representations generated in response to `POST` requests. This allows the caching proxy to achieve its goal of reducing overall network latency by serving prior representations the next time it sees a request for the same resource. However, in practice, most SOAP interactions always employ the `POST` method of HTTP regardless of whether or not the result of the SOAP interaction could be cached. Consequently, no intermediary (RA6) can know whether a service call will be idempotent without specific semantic knowledge of the actions taken by the SOAP receiver. Within REST, the protocol alone defines whether the operation is idempotent, without any relationship to the resource.

Tunneling a separate SOAP envelope within HTTP also violates REST's separation of data and metadata (RA2). SOAP encapsulates the entire message—including all SOAP metadata—in the body of the HTTP request as an XML entity. In order to properly parse (and understand) the request, a SOAP intermediary must examine the *entire* body of the HTTP request. In contrast, an HTTP proxy need only examine the metadata found in the request line and request headers and can pass the request body through without any inspection, as it is irrelevant for routing or caching. By hiding the routing and metadata for SOAP inside the body of the HTTP request, a SOAP intermediary must peek inside the body to ensure proper routing—a clear violation of RA2, the strict separation of metadata (the HTTP headers) and representation (the SOAP message payload).

The (apparent) lack of idempotency and the necessity of deep inspection of requests in SOAP interactions is a significant obstacle to intermediaries seeking to intelligently cache SOAP messages sent over HTTP. A caching SOAP intermediary requires “proprietary

knowledge” of the SOAP action in order to determine whether or not the response is cacheable. Compared to existing HTTP/1.1 caching implementations (such as Squid[Rousskov, 1999]), a SOAP cache would have to read and parse the entire XML request body (which has no theoretical size limit, but has been shown to introduce extreme latency into the request processing[Davis, 2002]), often negating the latency benefits of a cache. Therefore, such SOAP caches must always be custom developed and deployed with intimate knowledge of the SOAP requests and responses and could only offer, at best, negligible performance improvements. So, it is infeasible to create a generic reusable caching proxy for a SOAP-centric environment. In practice, this omission threatens the scalability of SOAP services because, unlike standard HTTP content servers, it is not possible to generically off-load the traffic by introducing “dumb” caches in front of the SOAP servers. Not being able to strategically deploy caches is a tremendous threat to the scalability of the overall Web if SOAP-based services were to proliferate.

### **3.4. Web Services: “RESTful” Services**

Many content providers expose REST-compliant APIs for their products. As there is no commonly accepted litmus test for REST compliance, we term those services that explicitly acknowledge REST in their description as “RESTful.” To help create “RESTful” services, Richardson and Ruby have recently presented a process for creating “resource-oriented architectures” and is described in Table 3.1 on page 114 [Richardson, 2007, pg. 216]. Other than this material, there is little concrete guidance available to service devel-

opers. Consequently, as we will see, there is a wide range of variation in the structure and semantics within the available “RESTful” services.

**Table 3.1: Generic ROA Procedure from [Richardson, 2007, pg. 216]**

1. Figure out the data set
2. Split the data set into resources
<b>For each kind of resource:</b>
3. Name the resources with URIs
4. Expose a subset of the uniform interface
5. Design the representation(s) accepted from the client
6. Design the representation(s) served to the client
7. Integrate this resource into existing resources, using hypermedia links and forms
8. Consider the typical course of events: what’s supposed to happen?
9. Consider error conditions: what might go wrong?

While RESTful services have not yet completely displaced their SOAP counterparts, they do have well-known advantages. In a 2003 talk, an Amazon spokesperson mentioned that 85% of their service requests employed their RESTful API, not SOAP [O’Reilly, 2003], and that querying Amazon using REST was roughly six times faster than with the SOAP equivalents [Trachtenberg, 2003].

A fundamental property of these RESTful services is that they interact well with generic HTTP/1.1 caching schemes (as outlined in [Wang, 1999]). Read-only operations are performed via `GET` requests, so these responses are cacheable by intermediaries without any special knowledge. However, write-able operations in RESTful services are conducted via `POST` requests, so these responses can not be cached. Yet, the design of most backing systems for high-traffic sites offering RESTful services tends towards extremely high availability (that is, read operations succeed) rather than immediate consistency (that is, write

operations can take time to take effect) [DeCandia, 2007][Chang, 2006]. This means that these designs are geared to always successfully respond to GET requests while delaying the effects of a POST request - in other words, reading dominates writing on the Web. In addition, traffic analysis at one large university shows that 77-88% of the web traffic is ideally cacheable, yet only 37-50% of the traffic is cached in practice - so there are efficiencies still to be gained from new caching strategies [Saroiu, 2002]. Therefore, the web as a whole is optimized for substantially higher read volumes than write volumes in order to achieve the requisite high scalability requirements seen in practice [Vogels, 2009].

A critical mechanism for meeting the high scalability requirements of the Web is the strategic introduction of caches [Rousskov, 1999]. Since these caches can be deployed anywhere within the network, Leighton identifies three places that a cache can be successfully deployed to reduce latency: the first mile, middle mile, and last mile [Leighton, 2009]. The first mile is operated by the agency owning the origin server so that they can distribute the traffic load amongst their web servers. Commonly referred to as “reverse proxies” or “gateways”, software like Squid [Rousskov, 1999] and Varnish [Grenness, 2008] or hardware appliances like F5’s BIG-IP [MacVittle, 2009], can be deployed with minimal changes to the application to perform load balancing and caching in this first mile. In the middle mile, a third-party (commonly referred to as a “content-delivery networks” or CDNs[Pallis, 2006] - one example is Akamai[Dilley, 2002]) operates a private set of caches in an attempt to help reduce latency. Some content-delivery networks even permit deployment of Web-based applications to their edge nodes [Davis, 2004], or use custom topologies to optimize their fetching algorithms [Katsaros, 2009]. Finally, for the last mile, a “forward proxy” (again, a system like Squid can be used here) to cache the traffic

of user-agents before the requests leave the local network. All three strategies combine to help produce the incredible scaling characteristics of the Web.

Despite their prevalence and popularity, these implementations of REST services are uneven at best—especially in comparison to other alternatives from the same provider. eBay’s REST API is extremely limited, permitting just read-only operations (such as queries). More advanced functions, such as posting new items, are available only through their SOAP API. Others have REST interfaces that are roughly equivalent to their SOAP counterparts; with Flickr, photos can be uploaded, tagged, and deleted in both interfaces with only minimal differences. Finally, there are examples, such as Amazon’s S3, for which their RESTful APIs are a superset of their SOAP counterpart.

### **3.5. Web Services: Observations**

Overall, we observe that the closer the service semantics are to those of content, the more likely the service is to have a rich REST API. For Amazon’s storage service, a mapping to the content-centric nature of REST is straightforward and free of complications. REST principles RA4 and RA5 are well-preserved in Amazon’s interface. On the other hand, eBay’s service model is strongly tilted toward SOAP. How to explain the differences? In part, the division between REST and SOAP may reflect a lack of design guidance; how can services that are not content-centric, such as auction bidding (which are prominently displayed in eBay’s SOAP interface but absent from the REST interface) be cleanly constructed in a REST-consistent manner? We opine that web services whose focus is nontraditional resources are clearly under-served with REST alone and that their developers lack adequate design guidance. Consequently, it is unsurprising that service providers offer alternatives to fill this gap.

**Table 3.2: HTTP Cookie Example**

Intent	HTTP Protocol (adapted from [Kristol, 1997])
User identifies self via a form	POST /acme/login HTTP/1.1  <i>form data</i>
Server sends a cookie to reflect identity	HTTP/1.1 200 OK Set-Cookie: CUST-ID="WILE_E_COYOTE"; path=/; expires=Sun, 17-Jan- 2038 12:00:00 GMT;
User returns cookie in a later request	POST /acme/pickitem HTTP/1.1 Cookie: CUST-ID="WILE_E_COYOTE"  <i>form data</i>

Yet, more importantly, without the scalability provided by intelligent caching (such as offered by off-the-shelf proxies and commercial CDNs), SOAP-based web services will simply not be able to scale to meet the demands of the Web. Service-oriented architectures will continue to be relegated to an isolated sphere of influence because SOAs will be incapable of delivering the performance to form the critical backbone of the Web. We must take away as a critical lesson that any true web services architectures must not inhibit the scalability of the Web.

### 3.6. Cookies

Besides RESTful APIs and SOAP, cookies [Kristol, 1997] are another common mechanism employed by developers to support services that span more than one request. Comparatively lightweight, cookies are a means for a site to assign a “session” identifier to a “user.” To start the process (illustrated in Table 3.2 on page 117), an origin server provides a cookie in response to a request via the “Set-Cookie” HTTP header. Inside of this cookie

are several attributes, the most important of which is an opaque identity token, a path representing where the cookie should subsequently be presented, and an expiration date.

This addition of an identifier assigned by the server to a client makes it “easier” for web developers to later remember the user who receives a specific cookie. As in our example, by assigning a user the cookie `CUST-ID="WILE_E_COYOTE"`, the website does not have to later guess who the user is on subsequent accesses. Since the cookie is presented in every subsequent request, the server will know that they are the `WILE_E_COYOTE` customer just by inspecting the new request and not require any separate identity information. However, there are serious security and privacy implications of cookies and “unless this type of system [cookies] is implemented carefully, it may be vulnerable to exploitation by unscrupulous third parties.” [Stein, 2003] Companies like Quantcast can perform statistical inferences based solely on cookie and clickstream information across many sites to infer an individual’s demographic information (such as age, sex, income) even in the absence of personally-identifying information [Quantcast, 2009]. In the early days of cookies, browsers would explicitly ask a user before accepting a cookie; however, due to the proliferation of cookies, browsers now silently accept cookies by default. According to one survey, over 25% of all sites use cookies [E-Soft Inc., 2007]. Cookies have become an integral part of the web in spite of the security and privacy concerns [Ha, 2006].

Besides the security and privacy implications, cookies also have poor interactions with the history mechanisms of a browser [Fielding, 2000]. Once a cookie is received by a user agent, the cookie should be presented in all future requests to that server until the cookie expires. Therefore, cookies do not have any direct relationship to the sequence of requests that caused the cookie to be set in the first place. Hence, if the browser’s back button is

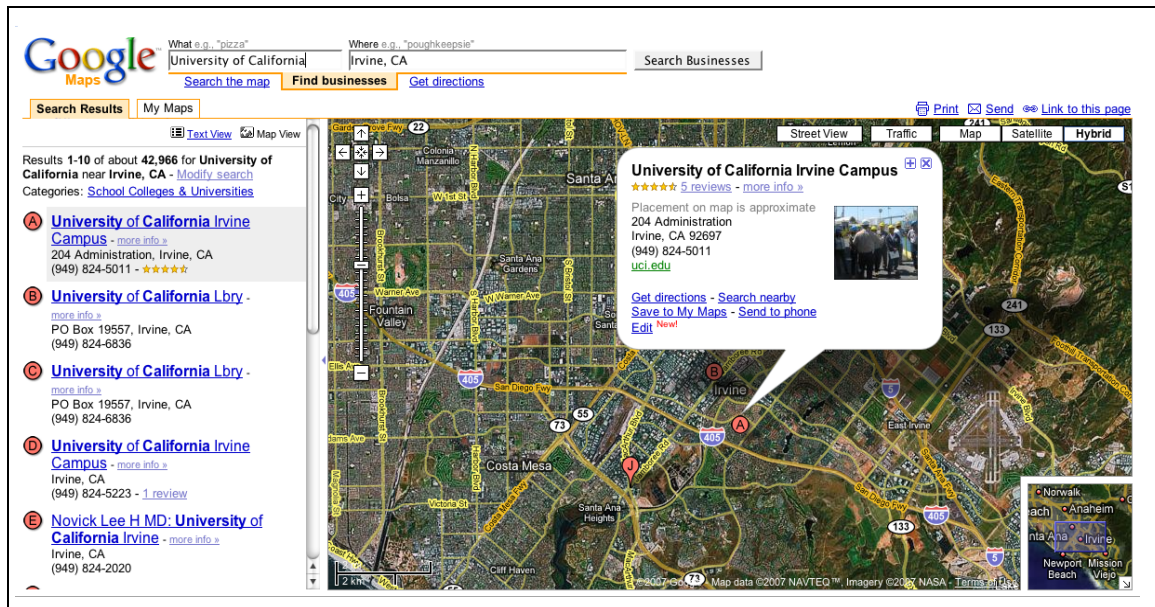
used to navigate in the page history prior to the initial setting of the cookie, the cookie will still be sent to the server. This can lead to a “cookie that misrepresents the current application context, leading to confusion on both sides” [Fielding, 2000]. This has led to a number of practical problems that user-agent vendors are still attempting to deal with gracefully [Pettersen, 2008].

### **3.7. AJAX**

We next take a look at ways in which service composition and encapsulation have been addressed through REST—albeit not under the traditional banner of web services. Emerging classes of Web applications extend the notion of a REST interface in interesting ways. One early example is Google Maps [Google, 2007] (shown in Figure 3.3 on page 120), which employs an application model known as AJAX [Garrett, 2005], consisting of XHTML and CSS, a browser-side Document Object Model interface, asynchronous acquisition of data resources, and client-side JavaScript. Using these techniques, developers have created vibrant client-side applications that are low latency, visually rich, and highly interactive.

As discussed at length in **Chapter 2**, early browser architectures relegated everything (even image viewing) to external helper applications which would be executed when “unknown” content (as indicated by the MIME type of the representation) arrived. Later on, browsers gained the ability to support third-party plugins - systems such as Java applets and Flash SWF applications rely upon this functionality. These plugins are able to receive a “canvas” within the browser’s window to paint the content and interact with the user. Due to security sandboxing restrictions, these plugins can not always easily interact with the rest of the elements on a page. Additionally, these plugins are required to be writ-





**Figure 3.3: Google Maps**

ten in the language of choice of the browser - which means that these plugins have to handle portability and cross-browser implementation issues without any direct assistance.

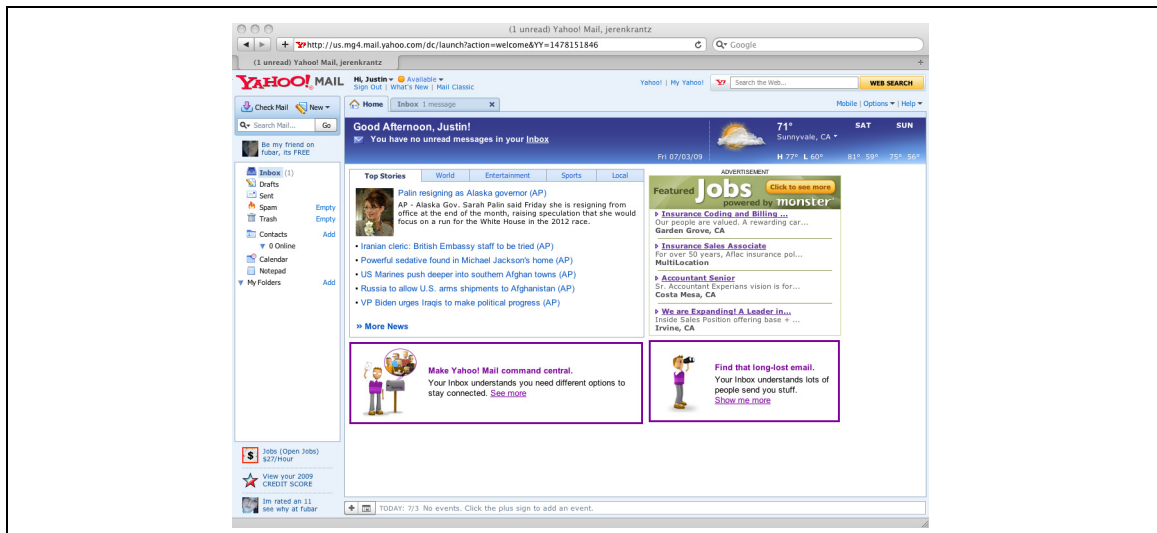
In comparison to plugins, AJAX relies upon only what is provided with modern browsers: JavaScript, CSS support, and DOM models. There have also been a number of developer frameworks created to help create AJAX-powered sites [Mesbah, 2008] as well as methodologies for testing AJAX-powered sites [Mesbah, 2009]. Given that no third-party plugins were required and the proliferation of AJAX frameworks, this enabled the proliferation of rich applications that did not require the user to install any third-party software. In fact, the popularity of AJAX has been beneficial to the competitiveness and openness of the Web - as it has restarted the innovation cycle amongst browser developers by forcing them to optimize their JavaScript and CSS engines [Google, 2008][Gal, 2006][Stachowiak, 2008].

From an architectural perspective, AJAX expands on an area for which REST is deliberately silent—the interpretation environment for delivered content—as content interpreta-

tion and presentation is highly content- and application-specific. However, instead of running the helper in a different execution environment, AJAX blurs the distinction between browser and helper by leveraging client-side scripting to download the helper application dynamically and run it within the browser's execution environment.

Dynamically downloading the code to the browser moves the computational locus away from the server. Instead of performing computations solely on the server, some computations (for example, presentation logic) can now be executed locally. By reducing the computational latency of presentation events, AJAX makes possible a new class of interactive applications with a degree of responsiveness that may be impossible in purely server-side implementations.

To illustrate the impact of latency and network traffic of AJAX, let us consider a popular site for webmail powered by AJAX - Yahoo! Mail depicted in Figure 3.4 on page 121.

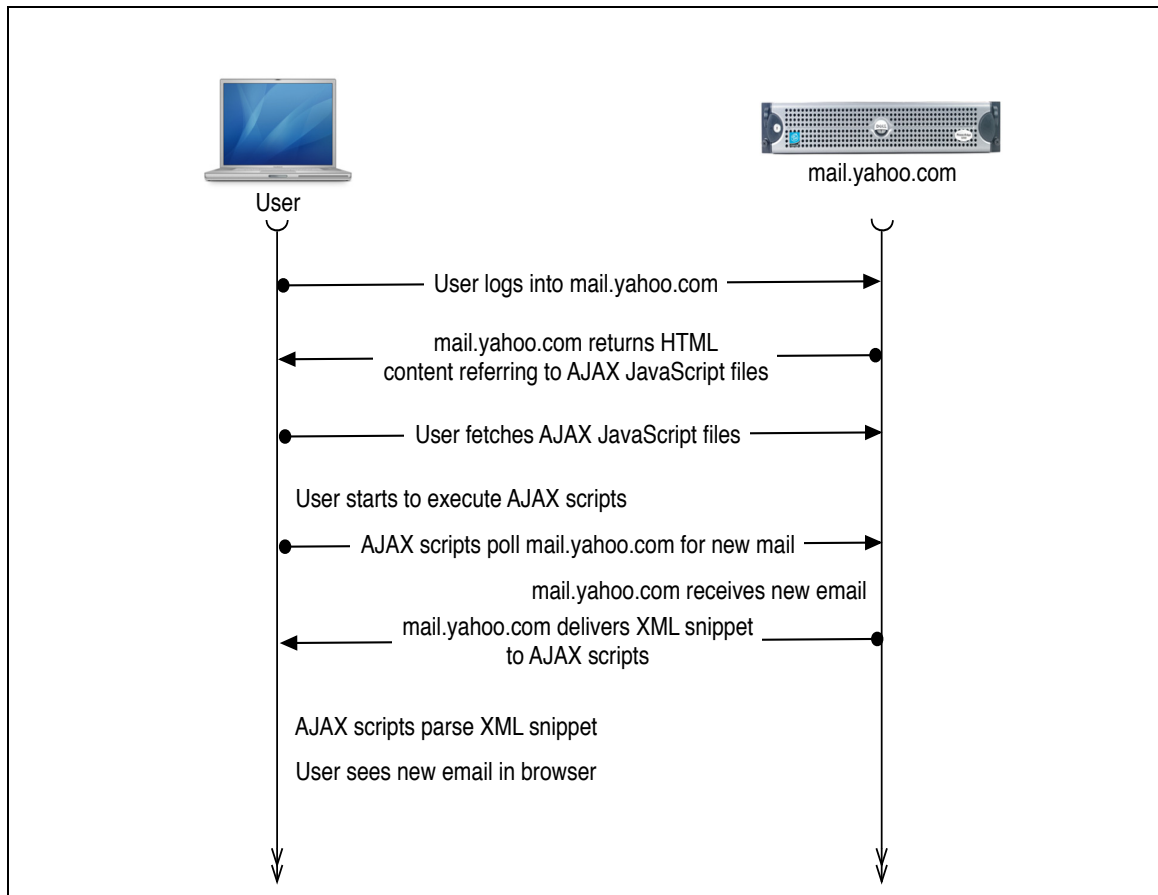


**Figure 3.4: Yahoo! Mail - an AJAX application**

A partial representation of the process view representing the interactions between the user's browser and the Yahoo! Mail site is depicted in Figure 3.5 on page 123. After successfully authenticating with the site, the local browser downloads a series of HTML and

JavaScript files. This JavaScript code contains all of the presentation logic and also starts frequently polling the Yahoo! servers as to whether new mail has arrived. When new mail arrives, an XML representation of the email is returned to the browser. The browser then processes that XML representation with the already-retrieved JavaScript code to display the new message to the user. At no point is the full representation that the user sees on their local screen transferred together - the presentation code arrives first, and then the data arrives as needed. Therefore, the important observation is that once the initial AJAX application is running, the bytes transferred between server and client are simply the minimal data necessary to populate the interface. In this way, AJAX amortizes the network traffic over time by transferring the presentation logic immediately in one burst and then delays the retrieval data until it is requested. This separation can greatly reduce the load on the servers as it does not need to combine the data with the presentation layer and send it to the user, and it also decreases the network latency by minimizing what must be transferred.

To restate, the architectural innovation of AJAX is the transfer, from server to client, of a computational context whose execution is “resumed” client-side. Thus, we begin to move the computational locus away from the server and onto other nodes. REST’s goal was to reduce server-side state load; in turn, AJAX reduces server-side computational load. AJAX also improves responsiveness since user interactions are interpreted (via client-side JavaScript) at the client. Thus AJAX, respecting all of the constraints of REST, expands our notion of resource.



**Figure 3.5: Yahoo! Mail process timeline**

### 3.8. Mashups

Mashups, another computation-centric REST-based service, are characterized by the participation of at least three distinct entities:

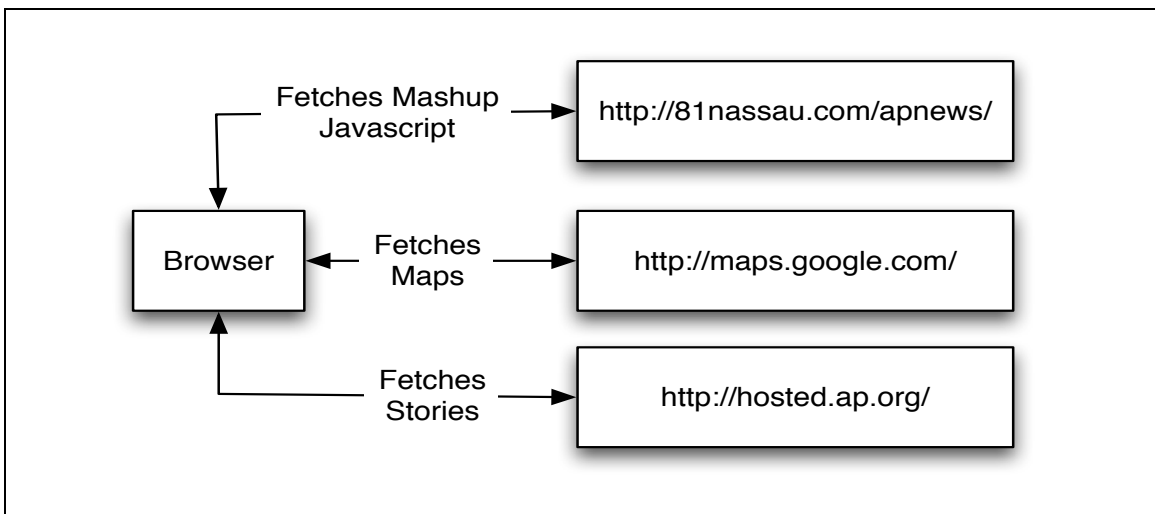
- A web site, the mashup host  $M$
- One or more web sites,  $c_1c_2\dots c_n$ ,  $c_i \neq M$ , the content providers
- An execution environment  $E$ , usually a client web browser

The mashup host  $M$  generates an “active” page  $P$  for the execution environment  $E$ .  $P$  contains references (URLs) to resources maintained by content providers  $c_1c_2\dots c_n$  and a set of client-side scripts in JavaScript. The client-side scripts, executing in  $E$ , interpret user actions and manage access to, and presentation of, the combined representations served by  $c_1c_2\dots c_n$ .

Mashups where Google Maps is one of the content providers  $c_i$  are especially popular; examples include plotting the location of stories from the Associated Press RSS feed [Young, 2006] (shown in Figure 3.6 on page 124 and an idealized process view shown in Figure 3.7 on page 124<sup>1</sup>), and Goggles, a flight simulator [Caswell-Daniels, 2007].



**Figure 3.6: AP News + Google Maps Mashup**



**Figure 3.7: Process view of AP News + Google Maps Mashup**

1. It must be noted that the process view is not completely faithful to the actual implementation of the mashup. In the actual version, the RSS feeds are pre-parsed and re-packaged by the 81nassau.com server rather than the client directly fetching from hosted.ap.org.

*Mashups offer a fresh perspective on REST intermediaries.* Conventionally, an intermediary intercepts and interprets all request and response traffic between client and server. In contrast, the mashup host  $M$  (the origin server), constructs a virtual “redirection” comprising a set of client-side scripts that reference resources hosted elsewhere at web sites (content providers)  $c_i, c_i \neq M$ . Thereafter, the client interacts only with the content providers  $c_i$ . Thus, mashup host  $M$  “synthesizes” a virtual compound resource for the client. Though a mashup could be implemented entirely server-side by  $M$ , it would increase server load and (for highly interactive mashups) latency. Mashups illustrate both the power of combining multiple resources under computational control to synthesize a new resource and the benefits of moving the locus of computation away from the origin server.

## CHAPTER 4

### **CREST: A new model for the architecture of Web-based multi-agency applications**

We see the web realigning, from applications that are content-centric to applications that are computation-centric: where delivered content is a “side-effect” of computational exchange. In a computation-centric web, distribution of service and the composition of alternative, novel, or higher-order services from established services is the primary goal. To help further the development of a computation-centric web, we need a new theory to both explain and guide its future.

#### **4.1. Recap of Lessons Learned**

Drawing from the work enumerated in Chapters 2-3, we have the following critical observations about building web applications from real systems:

- `mod_mbox`: Resources (as denoted by an URL) can represent more than just static content and can refer to various 'representations' generated on-the-fly to suit a particular domain (in this instance, web-based access to very large mail archives).
- `subversion/serf`: Decoupling communication and representation transformations internally within a user-agent's architecture can minimize latency, reduce network traffic, and improve support for caching. Also, it is feasible to deploy protocol-level optimization strategies that do not conflict with REST or existing protocols (in this case, many fewer requests were needed for the same operations). However, extreme care must be taken to not violate documented protocol expectations so as not to frustrate intermediaries (such as caching proxies).
- Web Services (SOAP): Offering fine-grained services is unquestionably a noble goal as it promotes composition and first-party and third-party innovation. However, due to implementation deficiencies (such as lack of idempotency support and improper inter-

mingling of metadata and data in SOAP messages), SOAP-based Web Services (and its sibling 'Service Oriented Architectures') are incapable of realizing the promise of fine-grained, composable services without fundamentally violating the REST axioms that permitted the web to scale. SOAP-based Web Services, if widely adopted, would rollback the scalability improvements introduced with HTTP/1.1.

- Web services (RESTful): A lack of clear design guidance for construction of “RESTful” services that are not closely tied to 'content' services (such as adding or removing documents) make for often maddeningly inconsistent and incomplete interfaces and, even though they handle the bulk of traffic for “web services,” RESTful Web Services have yet to reach their full potential.
- Cookies: Even relatively minor alterations to a protocol may grant a 'toehold' to widespread architectural dissonance. The alternatives at the time were not fully articulated; therefore, web site designers took the easy (and better articulated through Netscape's developer documentation) approach. This failure violated the statelessness interaction axiom of REST and raises inconsistencies with the semantics of the “back” button featured in today's browsers.
- AJAX and Mashups: AJAX and mashups illustrate the power of computation, in the guise of mobile code (specifically code-on-demand), as a mechanism for framing responses as interactive computations (AJAX) or for “synthetic redirection” and service composition (mashups). No longer must 'static' content to be transported from an origin server to a user agent - we now transfer 'incomplete' representations accompanied by domain-specific computations applied client-side to reify the 'content' with which a user interacts. The modern browser has matured into a capable execution environment - it can now, without the help of third-party “helpers,” piece together XML and interpret JavaScript to produce sophisticated client-side applications which are low latency, visually rich, and highly interactive. Additionally, AJAX-based “mashups” serve as the computational intermediary (proxy) in an AJAX-centric environment.



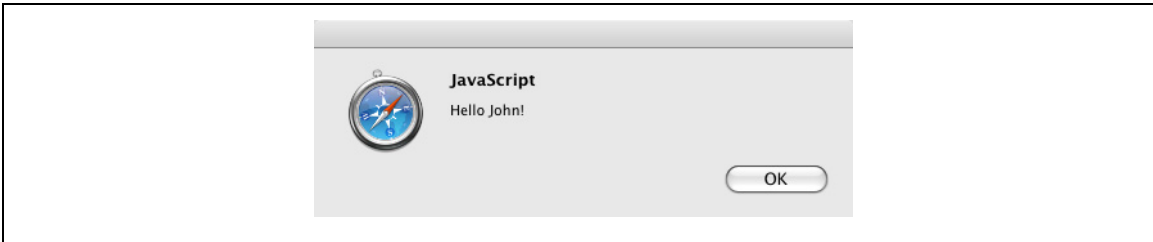
## 4.2. Continuations and Closures

AJAX and mashups both employ a primitive form of mobile code (JavaScript embedded in HTML resource representations) that expand the range of REST exchanges. However, far more powerful forms of computational exchange, based on a combination of mobile code and continuations, are available. A continuation is a snapshot (representation) of the execution state of a computation such that the computation may be later resumed at the execution point immediately following the generation of the continuation. Continuations are a well-known control mechanism in programming language semantics: many languages, including Scheme, JavaScript, Smalltalk, and Standard ML, implement continuations.

In these programming languages which support continuations, closures are the typical mechanism by which continuations are defined. Figure 4.1 on page 129 illustrates a closure in JavaScript. The function `closure_example` does not directly draw an alert box, but instead returns a function which draws an alert box. When the closure is evaluated, the user will see a pop-up box as depicted in Figure 4.2 on page 129. More precisely, a closure is a function with zero or more free variables such that the extent of those variables is at least as long as the lifespan of the closure itself. If the scope of the free variables encompasses only the closure then those variables are private to the function (they can not be accessed elsewhere from other program code) and persist over multiple invocations of the function (a value established for a variable in one invocation is available in the next invocation). Consequently, closures retain state (thereby sacrificing referential transparency) and may be used to implement state encapsulation, representation, and manipulation—the necessary properties for computational exchange powered by continuations.

```
function closure_example(name) {
  var alert_text = 'Hello ' + name + '!';
  var alert_closure = function() { alert(alert_text); }
  return alert_closure;
}
var v = closure_example('John');
v();
```

**Figure 4.1: An example of a closure in JavaScript**



**Figure 4.2: The resulting window produced by JavaScript closure in Figure 4.1**

### 4.3. Computational Exchange

We borrow liberally from a rich body of prior work on mobile code and continuations to articulate our view of computational exchange. An excellent survey and taxonomy of mobile code systems may be found in [Fuggetta, 1998] and, in particular, there are several examples of mobile code implementations based on Scheme. Halls' *Tubes* explores the role of “higher-order state saving” (that is, continuations) in distributed systems [Halls, 1997]. Using Scheme as the base language for mobile code, *Tubes* provides a small set of primitive operations for transmitting and receiving mobile code among *Tubes* sites. *Tubes* automatically rewrites Scheme programs in continuation-passing style to produce an implementation-independent representation of continuations acceptable to any Scheme interpreter or compiler. Halls demonstrates the utility of continuations in implementing mobile distributed objects, stateless servers, active web content, event-driven location awareness, and location-aware multimedia.

MAST is a Scheme variant for distributed and peer-to-peer programming that introduces first-class distributed binding environments and distributed continuations (in the spirit of Tubes) accompanied by a sound security model [Vyzovitis, 2002]. Like Tubes, MAST also provides primitives for mobility. MAST offers a developer fine-grain network control while supplying potent control and execution primitives.

Both Tubes and MAST achieve “computation mobility,” the movement of “live code” from one network host to another. Other language bases are also feasible. Tarau and Dahl achieve the same for a logic programming language `BinProlog`, again employing serialized continuations that are transferred from one host to another and then reconstituted [Tarau, 2001].

Mobile objects are a weaker form of computation mobility. Scheme appears in this context as `Dreme` in pursuit of distributed applications with little concern for process or network boundaries [Fuchs, 1995]. There, extensions to Scheme include object mobility, network-addressable objects, object mutability, network-wide garbage collection, and concurrency. `Dreme` also includes a distributed graphical user interface that relied upon continuations, rather than event-driven programming, to manage the interface and respond to user interactions and network events.

Continuations have an important role to play in many forms of web interactions and services. For example, Queinnec demonstrates that server-side continuations are an elegant mechanism to capture and transparently restart the state of ongoing evolving web interactions [Queinnec, 2000]; in other words, server/client interactions are a form of “web computation,” (represented by a program evaluated by the server) for which continuations are

required to suspend and resume stateful page tours or service-oriented sessions that are client-parameterized but generated server-side.

Matthews et al. extend this work, offering a set of automated transformations based on continuation-passing style, lambda lifting, and defunctionalization that serialize the server-side continuation and embed it in the web page returned to the client [Matthews, 2004]. When the client responds to the web page the (serialized) continuation is returned to the server where it is “reconstituted,” with the server resuming execution of the continuation. This is an example of computational exchange (from server to client and back again) that preserves context-free interaction and allows the server to scale by remaining largely stateless.

Finally, motivated by the richness of web interactions, browsing through data- and decision-trees, bookmarking, and backtracking, Graunke and Krishnamurthi explore bringing the same interaction techniques to non-web graphical user interfaces [Graunke, 2002]. They describe transformations, based on the continuation-passing style, that confer the power and flexibility of web interactions on graphical user interfaces.

## **4.4. A Computational Exchange Web**

To help illustrate the upcoming discussion with concrete examples, we have chosen Scheme as the language of choice for CREST.<sup>1</sup> In this particular computation-centric web, Scheme is the language of computational exchange and Scheme expressions, closures, continuations, and binding environments are both the requests and responses exchanged

---

1. Yet, it should be made clear that other languages could also suffice - to help demonstrate that, in this chapter, we will also illustrate some of the examples using JavaScript as well as Scheme.

over the web. Raising mobile code to the level of a primitive among web peers and embracing continuations as a principal mechanism of state exchange permits a fresh and novel restatement of all forms of web services, including serving traditional web content, and suggests the construction of new services for which no web equivalent now exists.

In the world of computational exchange, an URL now denotes a computational resource. There, clients issue requests in the form of programs (expressions)  $e$ , origin servers evaluate those programs (expressions)  $e$ , and the value  $v$  of that program (expression)  $e$  is the response returned to the client. That value (response)  $v$  may be a primitive value (1, 3.14, or "silly" for example), a list of values (1 3.14 "silly"), a program (expression), a closure, a continuation, or a binding environment (a set of name/value pairs and whose values may include (recursively) any of those just enumerated).

With CREST, there are two fundamental mechanisms of operation: `remote` and `spawn`. A `remote` computation evaluates the given program, closure, or continuation and (if there is a result) returns the resulting expression (which could be a new program, closure, or continuation) back to the original requestor. The other mechanism of operation is `spawn`, which is intended for installing longer-running custom computations. This mechanism allows a peer to install a new service and receive a new URL in response which permits communication with the newly installed service. This new URL can then be shared and messages can be delivered to the new service with the response wholly under the control of the newly installed computation.

To help illustrate the semantics of `remote`, we provide an example program (expression; rendered in the concrete syntax of Scheme in Figure 4.3 on page 133 and JavaScript in Figure 4.4 on page 133) issued by a client  $C$  to an URL  $u$  of origin server  $S$ . This program

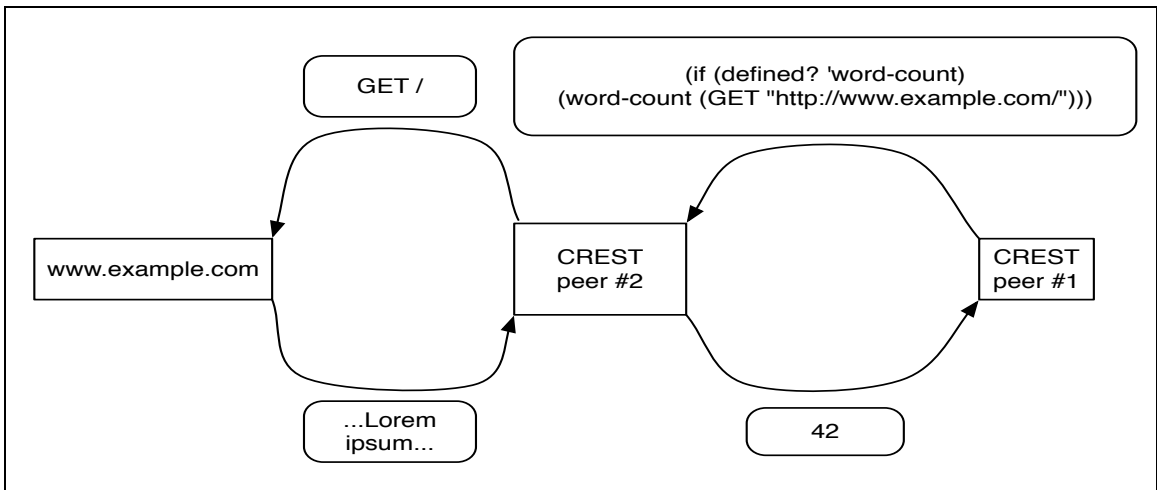
tests the execution environment of  $S$  for a function `word-count` (service discovery) and if the function (service) is available, fetches the HTML representation of the home page of `www.example.com`, counts the number of words in that representation (service composition), and returns that value to  $C$ . A diagram reflecting the division of computation reflected in this program is presented in Figure 4.5 on page 133. As shown by this example, in a computational exchange web, the role of SOAP can be reduced to a triviality, service discovery is a natural side-effect of execution, and service composition reduces to program (expression) composition.

```
(if (defined? 'word-count)
    (word-count (GET "http://www.example.com/")))
```

**Figure 4.3: Example CREST program (in Scheme)**

```
{
  if (wordcount) {
    return wordcount (GET ("http://www.example.com/"));
  }
}
```

**Figure 4.4: Example CREST program (in JavaScript)**



**Figure 4.5: Example CREST remote program (computational view)**

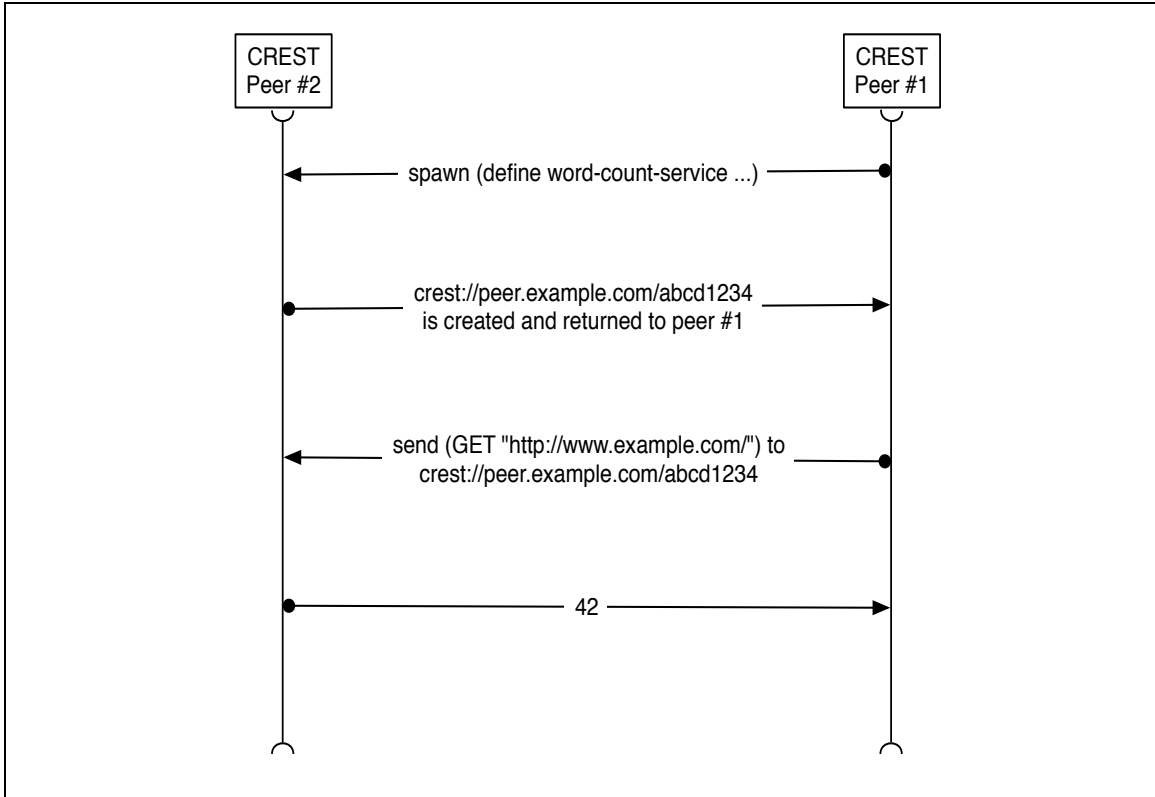
```

(define (word-count-service)
  (accept
   ((reply message-id ('GET url))
    (where (or (symbol? url) (string? url)))
    (! reply message-id (word-count (GET url))))
    (word-count-service))

  (_ (word-count-service)))) ; Ignore other message forms.

```

**Figure 4.6: Example CREST spawn service (in Scheme)**



**Figure 4.7: Messages exchanged when using spawn-centric service**

An example of a spawn-centric word-count service is provided in Figure 4.6 on page 134. In this example, the overall computational view remains the same as presented in Figure 4.5 on page 133, but with a crucial distinction—a new resource (represented by an URL) now exists, where no such resource existed before, that responds to word-count requests. In other words, CREST computations synthesize new services out of old by exposing URLs as computational resources rather than content resources. An example of

the sequence of computations and messages that are exchanged to install and use this service are presented in Figure 4.7 on page 134.

## 4.5. CREST Axioms

To provide developers concrete guidance in the implementation and deployment of computational exchange, we offer Computational REST (CREST) as an architectural style to guide the construction of computational web elements. There are five core CREST axioms:

**Table 4.1: Core CREST axioms**

CA1. A resource is a locus of computations, named by an URL.
CA2. The representation of a computation is an expression plus metadata to describe the expression.
CA3. All computations are context-free.
CA4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.
CA5. The presence of intermediaries is promoted.

### 4.5.1. A resource is a locus of computations, named by an URL. (CA1)

Any computation that can be named can be a resource: word processing or image manipulation, a temporal service (e.g., “the predicted weather in London over the next four days”), a generated collection of other resources, a simulation of an object (e.g., a spacecraft), and so on. Compared with REST, this axiom is not entirely inconsistent with the original REST axioms - many REST resources are indeed computation-centric (especially those presented under the guise of “RESTful Web Services”). However, CREST’s framing by explicitly emphasizing computation over information makes it far clearer that these are active resources intended to be discoverable and composable.



As discussed earlier, when a computational request arrives at an URL, two different modes of operation are possible: `remote` or `spawn`. If the closure is a `remote`, once the closure is received, the computation is realized within the context and configuration of the specific URL. In this way, the URL merely denotes “the place” where the received computation will be evaluated. Until the closure is received, the specified URL can be viewed as quiescent as there is no computation running but the URL merely presents the possibility of computation. If the evaluation of the received closure is successful, then any value (if there is one) returned by the closure on its termination will be sent back to the original requestor. In this way, `remote` is the CREST equivalent of HTTP/1.1’s GET. With a `remote` closure, there is no provision for communicating with the closure after it is exchanged. No outside party, including the client who originally submitted the `remote` request, can communicate in any way with the `remote` computation once it has begun evaluation.

```
http://www.example.com/mailbox/60d19902-aac4-4840-aea2-d65ca975e564
```

**Figure 4.8: Example SPAWN mailbox URL**

In contrast, if the closure received indicates a `spawn`, the locus of computation is revealed in a limited way in that a unique URL is then created to serve as a mailbox for that spawned computation. For example, Figure 4.8 on page 136 denotes one potential formulation for a mailbox URL whereby the hex string represents a universally unique identifier. In response to the `spawn` request, the requestor is returned this URL immediately. This particular URL may be used by the original requestor or provided to another node for its own use. With that mailbox URL in hand, any client may now send arbitrary messages to

the spawned closure via the mailbox. The executing closure which was provided with the initial `spawn` will read, interpret, and potentially respond to those messages.

#### **4.5.2. The representation of a computation is an expression plus meta-data to describe the expression. (CA2)**

Since the focus of CREST is computational exchange, it is only natural that the representations exchanged among nodes are amenable to evaluation. In this axiom, we follow Abelson and Sussman's definition of expression: "primitive expressions, which represent the simplest entities the [programming] language is concerned with" [Abelson, 1996]. While these exchanges may be simplistic in form (such as a literal representing static content or binary data), we expect that more complex expressions will be exchanged - such as closures, continuations, and binding environments. As discussed earlier in this chapter, closures and continuations are particularly well-suited for computational exchange as they are common programming language constructions for state encapsulation and transfer. By being able to also exchange binding environments (which associate variable names with their functional definitions or values), complex compositional computations can be easily exchanged.

CREST can also leverage its particularly active view of computation in order to produce more precise and useful representations. REST is nominally silent on the forms of exchanged representations but, to drive the application, implicitly requires the exchanged representations to be a form of hypermedia. In contrast, CREST relies upon computational expressions and the exchange of such to drive the application. To produce the appropriate representation, CREST employs explicit, active computations where REST relies upon a repertoire of interpreted declarative forms, such as the declared MIME types enumerated

by a User-Agent. CREST achieves far greater precision when negotiating a representation, for example, not only can a particular format be specified (such as JPEG) but also specific resolutions (thumbnails versus full images). This model of content negotiation can simply not be achieved in REST in a straightforward manner. Exploiting computation directly in the negotiations reduces its complexity and eliminates the complex parsing and decomposition of representations thereby improving encapsulation, isolation, and composibility.

#### **4.5.3. All computations are context-free. (CA3)**

Like REST, CREST applications are not without state, but the style requires that each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it. Prior representations can be used to help facilitate the transfer of state between computations; for example, a continuation (representation) provided earlier by a resource can be used to resume a computation at a later time merely by presenting that continuation. Again, REST has a similar restriction - however, the mechanism for interactions in a context-free manner was under-specified and offered little guidance for application developers. By utilizing computational semantics, continuations provide a relatively straightforward supported mechanism for achieving context-free computations.

#### **4.5.4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged. (CA4)**

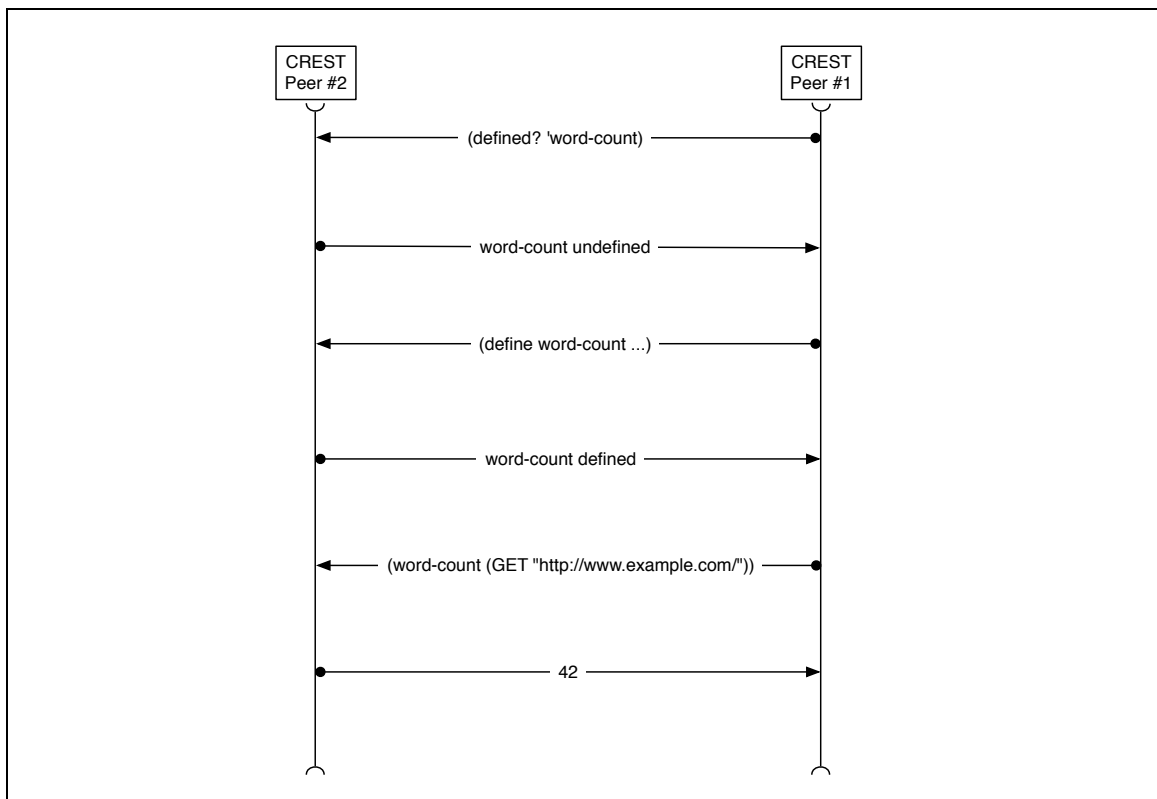
In HTTP/1.1 - the best known protocol instantiation of REST - the available operations (methods) of the protocol are documented in the relevant standards documentation (in this case, RFC 2616). With HTTP/1.1, the server may support additional methods that are not described in the standards, but there is no discovery mechanism available to interrogate

the server about which methods are supported on a particular resource. In CREST, since the node offers base programming language semantics, such discovery mechanisms are intrinsic and always available.

CREST nodes may define additional operations at a resource-level - that is, there may be operations (functions or methods) defined locally by the server which are pre-installed and are optimized for the server's specific environment. While these operations are exposed via CREST's computational model, these operations may actually be implemented in languages that are not directly amenable to CREST's computational exchange model (such as C, Java, or even raw machine code). For example, these locally defined functions may be front-ends to a proprietary database of credit scores, airline routings, or storehouse inventories. These mechanisms allow a particular provider to expose an optimized or value-added resources to the CREST nodes.

A critical feature lacking in the HTTP/1.1 protocol is that of two-way extensibility at runtime - new methods (such as those supported by extensions like WebDAV) can only be implemented on the server, but, other than implicit agreement or trial and error (handling the complete absence of a particular method), dynamic protocol adaptation is not feasible with HTTP/1.1. However, with a CREST-governed protocol - which relies upon providing a computational platform (in our examples, in the rendered form of Scheme) - protocol enhancements are merely a form of providing additional computations (such as new functions) on top of the existing computation foundation. Therefore, with CREST, if a specific method is not available, the participant can then submit the code to the resource which interprets that method exactly as the participant desires. An example of this dynamic protocol adaptation is provided in Figure 4.9 on page 140.

In other words, participant  $A$  can send a representation  $p$  to URL  $u$  hosted by participant  $B$  for interpretation. These  $p$  are interpreted in the context of operations defined by  $u$ 's specific binding environment or by new definitions provided by  $A$ . The outcome of the interpretation will be a new representation—be it a program, a continuation, or a binding environment (which itself may contain programs, continuations, or other binding environments). To reiterate, a common set of primitives (such as the base semantics of the computational substrate, such as the Scheme primitives) are expected to be exposed for all CREST resources, but each  $u$ 's binding environment may define additional resource-specific operations and these environments can be further altered dynamically.



**Figure 4.9: Example of dynamic protocol adaptation in CREST**

#### 4.5.5. The presence of intermediaries is promoted. (CA5)

Filtering or redirection intermediaries may also utilize both the metadata and the computations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the end user-agent and the origin server. The clear precursor of the full power of this axiom in a computational exchange web is in AJAX-based mashups, more fully explained in Section 3.8 on page 123. A derivative of our earlier word count example is given in Figure 4.10 on page 141. In this example, CREST peer #2 has configured itself to use `babelfish.example.com` to translate all outgoing requests into Portuguese. CREST peer #1's code has not changed, but CREST peer #2's new configuration alters the computation yielding a different result.

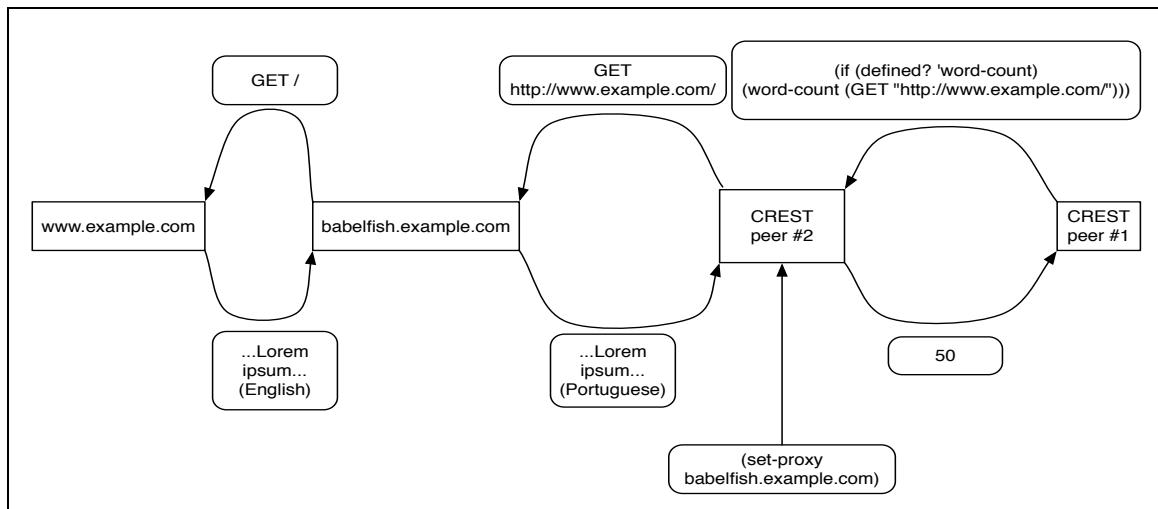


Figure 4.10: Word count example with an intermediary

## 4.6. CREST Considerations

As the REST experience demonstrates, it is insufficient to merely enumerate a set of architectural principles; concrete design guidance is required as well. To this end, we explore some of the consequences of the CREST axioms, cautioning that the discussion here is neither exhaustive nor definitive. Nonetheless, it draws heavily upon both our experiences

as implementors of web services and web clients and the lessons of the analyses of prior chapters. A summary of the considerations we feel are essential in guiding the application of the CREST axioms can be found in Table 4.2 on page 142.

**Table 4.2: Summary of CREST Considerations**

Names
Services
Time
State
Computation
Transparency
Migration and latency

#### **4.6.1. Names**

CREST can name specific computations (CA1) and exchanges their expressions between nodes (CA2). In determining how to achieve this exchange, we believe that the most logical solution is to physically embed the expressions directly in the URL. To go back to our earlier word count example, the `remote` request can be constructed as depicted in Figure 4.11 on page 143. By reusing the existing URL specification and embedding the computation directly in the URL, CREST provides a mechanism for embedding computations inside HTML content - allowing an existing user agent (such as Firefox) to interact successfully (and without its direct knowledge) with a CREST node. This explicitly permits CREST to be incrementally deployed on top of the current Web infrastructure without requiring wholesale alterations or adoption of new technology. To be more precise about this reuse of the URL format, if  $a$  is the ASCII text of a expression  $e$  sent by client  $c$

to URL  $P://S/u_0/.../u_{m-1}/$  of origin server  $S$  under scheme  $P$  then the URL used by  $c$  is  $P://S/u_0/.../u_{m-1}/a/$ .

To be clear, CREST URLs are not intended for human consumption, as they are the base mechanisms of computational exchange among CREST nodes; human-readable namings may be provided as needed by higher layers. Among computational nodes, the length of the URL or its encoding is irrelevant and ample computational and network resources are readily available among modern nodes to assemble, transmit, and consume URLs that are tens of megabytes long. In effect, the URL  $u=P://S/u_0/.../u_{m-1}/$  is the root of an infinite virtual namespace of all possible finite expressions that may be evaluated by the interpreter denoted by  $u$ . Finally  $u'$ , a moderately compact and host-independent representation of a computational exchange, may be recorded and archived for reuse at a later point in time (CA3). One possible compact representation of our word count example is provided in Figure 4.12 on page 143.

```
crest://server.example.com/(if(defined?'word-count')(word-count(GET "http://www.yahoo.com/")))
```

**Figure 4.11: CREST URL example (expanded)**

```
crest://server.example.com/word-count/www.yahoo.com/
```

**Figure 4.12: CREST URL example (condensed)**

### 4.6.2. Services

A single service may be exposed through a variety of URLs which offer multiple perspectives on the same computation (CA1). These interfaces may offer a different binding environment or offer complementary supervisory functionality such as debugging or management (CA4). Different binding environments (altering which functions are available) may be offered at different URLs which represent alternate access mechanisms to the same underlying computation. We envision that a service provider could offer a tier of



interfaces to a single service. For instance, a computation that performs image processing could be exposed as a service. In this hypothetical service offering, a free service interface is exposed which allows scaling up to a fixed dimensions (up to 1024x768) and conversions into only a small set of image formats (only JPEG and GIF). In addition to this free service, another interface could be exposed for a fee (protected via suitable access controls) which places no restrictions on the dimensions of the resized image and offers a wider range of support for various image formats (such as adding RAW or TIFF formats). In addition, these alternative URLs can be used to perform supervisory tasks for a particular service. For long running custom computations, as is the intention for `spawn`, an outside party might desire more insight into the progress and state of the computation. The outside party may also wish to suspend or cancel the computation. In this case, a unique “supervisory” URL `d` can be generated by the CREST interpreter in addition to the spawned computation’s mailbox. Clients can then direct specific `remote` closures to `d` in order to access special debugging and introspection functions. For example, the supervisory environment can provide current stack traces, reports of memory usage, and functions to monitor communications originating from the computation or messages arriving at the mailbox. If the environment chooses, it can also expose mechanisms to suspend or kill the computation. A `remote` closure delivered to this supervisory URL could then combine these debugging primitives to produce a snapshot of the spawned computation’s health. Or, if the supervisory environment supports it, a new `spawn` closure can be delivered to `d` which will automatically kill the computation if the original closure exceeds specific parameters.

### 4.6.3. Time

The nature and specifics of the locus of computation may also vary over time. For example, functions may be added to or removed from the binding environment over time or their semantics may change (CA4). One potential reason for this variation is that a service provider wishes to optimize the cost of providing the service depending upon the present computational load. Therefore, in a service representing a complex mathematical function, a provider can offer a more precise version of a function that uses more CPU time during off-peak hours. However, during peak hours when overall computational cycles are scarce, a less precise variant of the function can be deployed which uses less CPU time. Additionally, the interpreter may change as well for the sake of bug fixes, performance enhancements, or security-specific improvements.

Functions in the binding environment may also return different values for the same inputs over time. For example, a random number generator function would be required to vary its output in successive calls in order to be useful. Yet, there is nothing to prevent a locus from being stateful if it so desires. A URL representing a page counter computation that increments on each `remote` invocation would be stateful. It is important to understand that the locus, in addition to everything else may be stateful and that state, as well as everything else, is permitted to change over time.

### 4.6.4. State

It is vital to note that many distinct computations may be underway simultaneously within the same resource. A single client may issue multiple `remotes` or `spawns` to the same URL and many distinct clients may do the same simultaneously. While a computational locus may choose to be stateful (and thus permit indirect interactions between different com-

putations), it is important to also support stateless computations whereby these parallel computations do not have any influence or effect on any other instance within the same computational namespace (CA3). With stateless computational loci, independent parallelization of the evaluation is easily available and straightforward.

In order to address scalability concerns with stateful services (such as a database), specific consistency mechanisms must be introduced to try to regain parallelization. The coarsest-grained consistency mechanism would be to only allow one evaluation of the stateful service at a time as protected by a mutex (akin to the Java `synchronized` keyword). However, as discussed in Section 3.4 on page 113, the web as a whole tends to require optimizing for substantially higher read volumes than write volumes. Therefore, it is possible to introduce weak consistency models where writes are delayed or processed independently in order to permit highly parallelizable read operations [Lamport, 1978][Dubois, 1986][DeCandia, 2007][Chang, 2006].

#### **4.6.5. Computation**

REST relies upon an end-user's navigation of the hypermedia (through the links between documents) to maintain and drive the state of the overall application. In contrast, CREST relies upon potentially autonomous computations to exchange and maintain state (CA2, CA3). Given the compositional semantics offered with CREST, we expect that it will be common for one resource to refer to another either directly or indirectly. As a consequence, there may be a rich set of stateful relationships among a set of distinct URLs (CA1). This state will be captured within the continuations that are exchanged between services. A service provider will be able to give its consumers a continuation that permits later resumption of the service. In this way, the provider does not have to remember any-

thing about the consumer as all of the necessary information is embedded inside of the continuation. As long as the consumer is interested in persisting the stateful relationship, it needs to merely retain the continuation (embedded in an URL) to let the provider resume the state of the service at a later date.

#### **4.6.6. Transparency**

Since the computational mechanisms are transparently exposed in an URL, the computation can be inspected, routed, and cached. In this way, intermediaries and proxies are strongly embraced by CREST (CA5). This allows a service to scale up as needed by sharing network resources; a single origin server may now be dynamically reconstituted as a cooperative of origin servers and intermediaries (CA3, CA4). A proxy can be interjected into a request path in order to record all of the computations that are exchanged between parties. This information can then be examined online or post-mortem to depict traffic flow analysis. In turn, the intermediary can also do intelligent routing based on just the URL and status line information. This is in contrast to SOAP routers (discussed in Section 3.3 on page 110) which require inspecting the request body of the message in order to determine routing paths. Since the computations are also transparent, they can be altered by the intermediary. For instance, we envision an intermediary which can convert search requests against one search vendor's interface that can dynamically rewrite and reroute it to use another search vendor's interface by intelligent deep inspection of the binding environments.

More importantly, CREST also permits caching of computations to be performed by off-the-shelf existing caching modules. In particular, `remote` closures can have their representations (evaluation) reused by standard HTTP/1.1 caches. Since all of the computation

is specified directly in the URL, the cache can then compare the URL against all prior interactions that it has cached. If the response is permitted to be cacheable (as dictated by site policy or by the original service provider) and is still fresh according to the specified expiration parameters, then that prior interaction can be leveraged and returned to the requestor without contacting the original service provider. This feature permits the graceful introduction of caching into the CREST environment.

#### **4.6.7. Migration and latency**

Given the dominance of computation and computational transfer within CREST, applications should adopt a strategy of minimizing the impact of network and computational latency. This can best be achieved migrating computations to the place where it offers the best value. Consumers must be willing to dynamically stitch partial computations from service providers into a comprehensive whole to obtain desired results, as no one origin server may be capable of supplying all of the functional capability that the client requires. As discussed in Section 2.2.2 on page 102, the physical characteristics of the underlying network connection can have a substantial effect on latency. Recent investigations into cloud computing services have confirmed that if a computation over a set of data (such as aggregation or filtering) where the computation processing is physically separated from the data store (such that the data is located on another continent from where the filtering occurs), the performance will be dominated and subject to the variations caused by network latency [Palankar, 2008]. To remedy this behavior, CREST easily facilitates the exchange of the computation to be physically closer to the data store thereby reducing the impact of latency (CA2). The consumer merely needs to provide the computation to the data store service which permits the computation and the data to be closer together. For

computations which reduce the data set to a subset (filtering) or even a single value (such as a summation), this reduces the latency effects on the overall application dramatically.

With CREST, not only can the computation be moved closer to the data, but the computation can also be delayed until it is required. For rendering computations (where the data size increases as an effect of the computation - such as rendering an email message into HTML), CREST can serve here as well in reducing the impact of latency. As established by AJAX applications (discussed in Section 3.7 on page 119), computations can be shipped to a suitably-empowered user agent which reduces the overall latency of the application by only requiring minimal application-specific data feeds to be retrieved. The local computations already running within the user agent can then render the data locally.

In order to support these latency mitigation procedures, certain characteristics must be internally within a specific node as any latencies in-the-large will be reflected in latencies in-the-small; for example, a client becomes unresponsive while it fetches a resource, or an intermediary stalls while composing multiple representations from upstream sources. Therefore, both clients and origin servers must have mechanisms for reducing or hiding the impact of latency. Clients must be hospitable to concurrent computation and event-driven reactions (such as the nondeterministic arrival of responses). Since those responses may be continuations, origin servers, in an analogous manner, must be open to receiving previously generated continuations long after they were first created (on timespans of seconds to months) and restarting them seamlessly. Time is also fundamental to CREST nodes as both origin servers and clients require timers to bound waiting for a response or the completion of a computation. In addition, CREST nodes may employ timestamps and

cryptographic signatures embedded within a continuation to reliably determine its age and may refuse to resume continuations that are past their “expiration dates.”

## **4.7. CREST Architectural Style**

Broadly speaking, we can gather these guidelines and considerations together to codify a new architectural style that expands REST’s concept of state transfer to encompass computational transfer. Just as REST requires transparent state exchange, our new style, Computational REST (CREST), further requires the transparent exchange of computation. Thus, in the CREST architectural style, the client is no longer merely a presentation agent for content—it is now an execution environment explicitly supporting computation.

# CHAPTER 5

## Revisiting Dissonance with CREST

As discussed earlier in Chapter 2 starting on page 96, REST has proven difficult to apply even for those who are well-versed in the style. Chapter 3 starting on page 108 discusses how recent emergent architectures and applications on the Web are poorly explained solely by REST. Chapter 4 starting on page 126 introduces the CREST architectural style which aims to help developers consistently gain the benefits promised by REST. In order to properly assess the utility of CREST, we explore how well CREST retrospectively explains the systems introduced in Chapter 2 and Chapter 3.

### 5.1. Explaining `mod_mbox`

**Names.** As highlighted in Section 2.1 on page 96, `mod_mbox` took specific care in crafting its exposed namespace (CA1). Instead of exposing storage details in the namespace as other archivers did, `mod_mbox` only exposed content-specific metadata: the Message-ID header. This level of indirection shielded the web-level namespaces from irrelevant implementation decisions made at a lower level - such as the choice of backing database or message arrival sequences.

**Services.** In addition to retrieving specific messages, `mod_mbox` also added several resource-specific functions such as dynamic indexing and sorting of the archive (CA4). By specifying service names in the URL (CA1), `mod_mbox` would sort the archive index by author, date, or thread.



**Time.** One of the scaling challenges with mail archives is how to keep up with a steady stream of new messages entering the archive. As discussed previously, other archival systems would regenerate the index and all of the representations whenever any new message is received. This style of processing becomes unsuitable for high-volume archives as the re-generation would essentially degrade into a constantly never-ending process. Therefore, in contrast to other archivers, `mod_mailbox` chose to delay the creation of index and message representations until it is requested by a user. This permitted `mod_mailbox` to scale and handle time in a domain-appropriate manner.

**Migration and latency.** By being flexible about the creation of message representation formats, `mod_mailbox` was able to later evolve gracefully as subsequent development added an AJAX presentation interface to the archive. Again, instead of creating all of the necessary representations at indexing time, `mod_mailbox` could simply dynamically expose a new namespace and representation format suitable for AJAX-capable clients (CA2, CA3). Instead of rendering the message or index entirely into HTML on the server, we were able to trivially shift that rendering to the browser by delivering JavaScript to control the rendering client-side with the mail messages formatted into XML whenever an AJAX-capable client requested it. This migration also reduced latency considerations as less data had to be transferred between server and client and led to a more responsive application.

## 5.2. Explaining Subversion

**Migration and latency.** As discussed in Section 2.2 on page 98, Subversion initially suffered from network latency issues. With the explanatory powers of CREST, we view the first attempt to solve this latency issue (performing, on the server, the aggregation of

resources to check out) as moving the computational locus back from the client to the server. Unfortunately, while addressing the initial latency issue, this only served to increase the overall computational load on the server and made it such that simple intermediaries could not be deployed to reduce load. But, with considered changes to the client's framework, consistent with latency reduction, we could repair the deficiencies. A new client was deployed that addressed the issues of latency through independent transformational elements (buckets) and asynchronous network calls. Hence, the computational locus (the aggregation of resources to check out) could rightfully return to the client. The server's load is thereby lessened and intermediaries can be redeployed.

**Naming and Transparency.** We can see that Subversion dealt with the naming concern in a very straightforward manner with one glaring exception which had a dramatic impact on the transparency of the overall system. All of the necessary parameters to identify a unique piece of content was exposed directly in the URL (CA1) - such as `/path/to/repos/!svn/ver/1234/foo/bar/baz`. This transparency makes it relatively straightforward to leverage caching mechanisms. However, as discussed earlier, the usage of Web-DAV's `REPORT` method hid the real computational request inside the request body rather than the exposing it directly in the URL or using the HTTP/1.1 method field. This lack of transparency makes it impossible to intelligently and simplistically route and cache any requests using `REPORT`. Unfortunately, while its usage has been minimized through the changes discussed here, the `REPORT` method is still used within Subversion for checkout and update operations.

### 5.3. Explaining Web Services

**Services.** by using a well-defined binding environment (CA4) and continuations (CA3), CREST provides more substantial design guidance on how to create arbitrary dynamic services that are flexible and content independent. For instance, for a service that may return a large number of results (such as a list of all live auctions), the exchange of continuations (as generators) is more natural, responsive, and elegant than building pagination and chunking explicitly into the interface. On each round  $i$  the service  $S$  may return a ordered pair  $(a_i, C_i)$  where:

- $a_i = (a_{i,0}, \dots, a_{i,m_i})$  is a list of the representations of the next  $m_i$  out of  $n$  total live auctions and
- $C_i$  is the continuation that, when returned to  $S$  at sometime in the future, will generate the next set of  $(a_{i+1,0}, \dots, a_{i+1,m_{i+1}})$  live auction representations.

Note that continuation  $C_i$  need not be returned to the origin server immediately; many seconds or minutes may pass before the client returns the continuation for the next round of exchange. This allows the client to pace its consumption of the results, a freedom that pagination and chunking do not provide. Further, the client may pass a continuation  $C_i$  onto one or more fellow clients for other purposes; for example, parallel searching for auctions with particular items for sale.

**Naming and Transparency.** One major deficiency of SOAP-based Web Services is the improper intermingling of metadata and data in SOAP messages. From a naming perspective (contrary to CA1), the identification of the SOAP service to be invoked is hidden inside the data of the HTTP request body typically encoded inside of XML. Therefore, all routing decisions (if even offered) are necessarily very inefficient due to the parsing and

inspecting of the entire request body. Due to this intrinsic inefficiency, SOAP-based intermediaries are uncommon.

**Migration and latency.** SOAP-based services do not expose any primitive compositional functionality to consumers (CA4). This absence has a deleterious effect on the offering of fine-grained web services as SOAP-based services tend to be rather coarse-grained to overcome this deficiency. Without compositional semantics, it is not possible to remotely combine the result of a SOAP service and compose it with another remote SOAP service without undue exposure to the network latency conditions. Therefore, SOAP-based web services must offer a separate distinct service representing the hard-wired functional composition of the underlying services. Therefore, the consumer's compositional abilities are restricted to whatever combinations are offered by the service provider or suffer the effects of network latency when trying to do the service composition locally.

## 5.4. Explaining Cookies

**State.** Under CREST, cookies are now reinterpreted as a weak form of continuation (CA2). When a client wants to resume the transaction represented by the cookie (continuation), it simply returns the last cookie (continuation) to the server. The key distinction under CREST is that cookies (continuations) are bound to a specific, time-ordered, request sequence. A full continuation is, by definition, bound to a particular sequence of resource access; there is no ambiguity server-side. Thus the continuation restarts resource access at a particular known point in the sequence of all resource accesses (CA2, CA3). This stands in sharp contrast to the current use of cookies—*generic* tokens to all subsequent server requests.

**Time.** Cookies are also currently presented with an explicit expiration date (in practice, many sites set their cookies to the latest expiration time supported by 32-bit platforms: January 19, 2038 [Microsoft Corporation, 2006]). However, in CREST, no such expiration date must be supplied (though it may, at the discretion of the origin server) as the continuation itself contains all of the necessary state to resume the dialog of exchange. Finally, continuations, like cookies, can be hardened against tampering using digital signatures or even encryption to prevent security or service attacks (CA2).

## 5.5. Explaining AJAX and Mashups

**Migration and latency.** From the CREST perspective, mashups are nothing more than a closure (CA2) that makes reference to resources on multiple web sites  $w_1, w_2, \dots, w_n$ . Note that for CREST, there is no requirement that a web browser be the execution environment for the mashup. By using CREST, we can predict two future elaborations of mashups. A *derived* mashup is one in which one or more content provider web sites  $w_i$  are themselves mashups—with the lower-level mashups of the  $w_i$  executing on an intermediary (CA5) rather than a browser. CREST also speaks to a future web populated by *higher-order* mashups. Similar to a higher-order function in lambda calculus, a higher-order mashup is a mashup that accepts one or more mashups as input and/or outputs a mashup (CA2). This suggests a formal system of web calculus, by which web-like servers, clients, and peers may be cast as the application of identifiable, well-understand, combinators to the primitive values, functions, and terms of a given semantic domain. Thus, CREST hints at the existence of future formalisms suitable for the proof of REST and CREST properties.

## 5.6. Evaluation

How are we to evaluate the validity of CREST as an explanatory model of modern and emerging web behavior?

First, note that REST is silent on many issues of architectural mismatch, repeatedly neglects to offer explicit design guidance, lacks a bright line separating REST-feasible web services from those that are not, fails to predict novel services that are consistent with REST principles and is frequently silent on many issues of web behavior. In each of these cases, CREST fills the gap and provides detailed guidance and explanations where none existed previously.

This is particularly acute in the case of SOAP and web services. Why are developers so focused on ignoring REST constraints for the sake of web services? Developers are struggling toward service interactions far finer-grained than fetching hypermedia content. But, absent a comprehensive computational model, the only mechanism even remotely suggested by REST is parameterized request/response that relies on the ill-suited semantics of the GET and POST methods of HTTP. CREST tackles the problem directly, since content exchange is nothing but a side-effect of its primary focus: computational exchange. Further, it demonstrates why SOAP and the tower of web service standards and protocols stacked above it utterly fail; computational exchange requires the full semantics of powerful programming languages: conditionals, loops, recursion, binding environments, functions, closures, and continuations, to name only a few. Without these tools, web service developers are condemned to recapitulate the evolution of programming languages.

CREST identifies the precise reasons why the evolution to web services is so difficult, pinpoints the mechanisms that must be applied to achieve progress, and offers detailed

architectural and design guidance for the construction of service-friendly servers and clients. Thus, CREST offers guidance where REST and all others have failed so far. In future work, we intend to apply CREST to the entire spectrum of web services—recasting all major elements of the web services protocol stack in the light of computational exchange—as well as address other outstanding problems in web behavior, including content negotiation and effective caching in service-oriented architectures.

## CHAPTER 6

### CREST Framework and Experience

To both facilitate the adoption of CREST and to explore the implications and consequences of the style, we have constructed a CREST framework that allows us to build applications in this style. Our framework has two classes of peers: *exemplary* peers and *weak* peers. As depicted in Figure 6.6 on page 163, exemplary peers are standalone servers that have a rich set of computational services available. These exemplary peers utilize Scheme as the language in which the computations are written. In order to assist and expose third-party libraries, our Scheme implementation is written on top of Java [Miller, 2003] - so any Java frameworks are accessible from the exemplary peer. To foster interoperability with the existing Web, these exemplary peers can act as a HTTP/1.1 server (to expose the computations running on that peer to browsers) as well as an HTTP/1.1 client (to permit the local computations on that peer to fetch resources on a HTTP/1.1 server or another remote peer). On a modern Intel-class laptop with minimal performance tuning, our exemplary peers can serve dynamic computations in excess of 200 requests per second. In contrast to exemplary peers, weak peers (depicted in Figure 6.5 on page 163) are confined to the restrictions of a modern Web browser and rely upon JavaScript as the fundamental computational foundation. For ease of development and portability for our weak peers, our example applications use the Dojo JavaScript framework. Supported browsers include Mozilla Firefox, Safari, Google Chrome, and Internet Explorer. Mobile devices such as an Apple iPhone and Google Android phone are also supported as weak peers (through their built-in browser applications.)



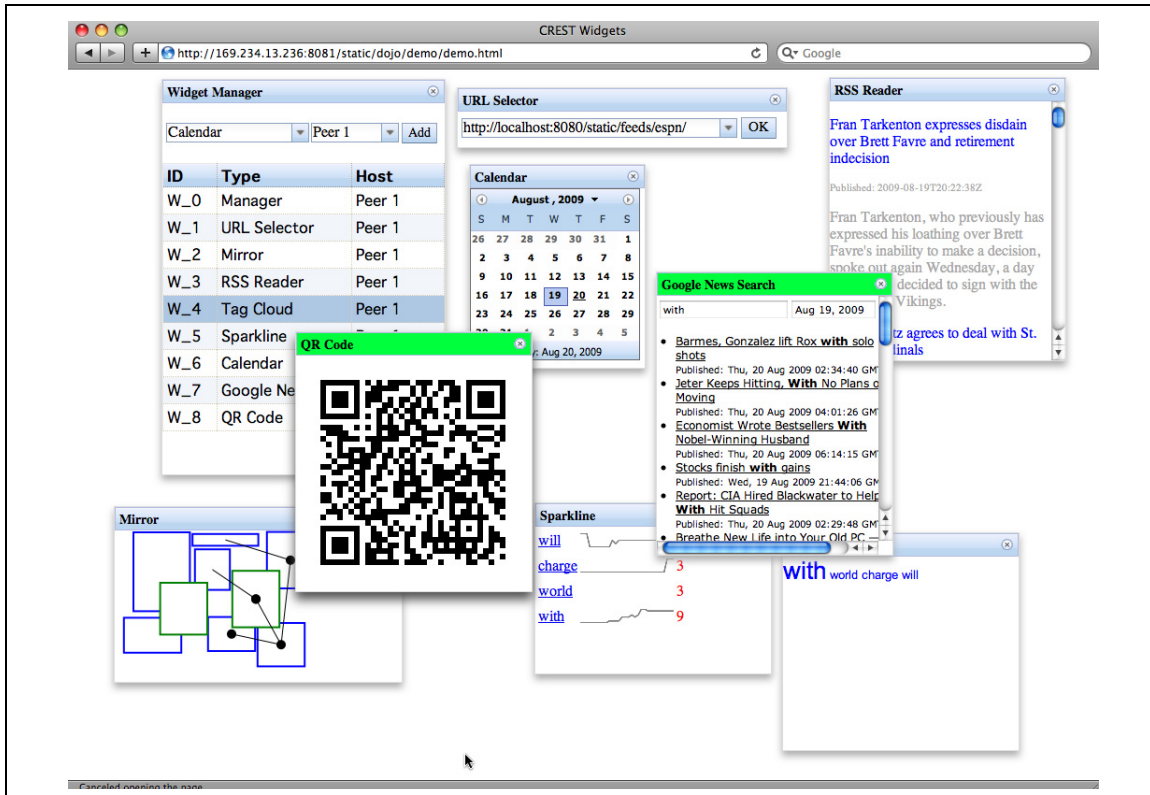
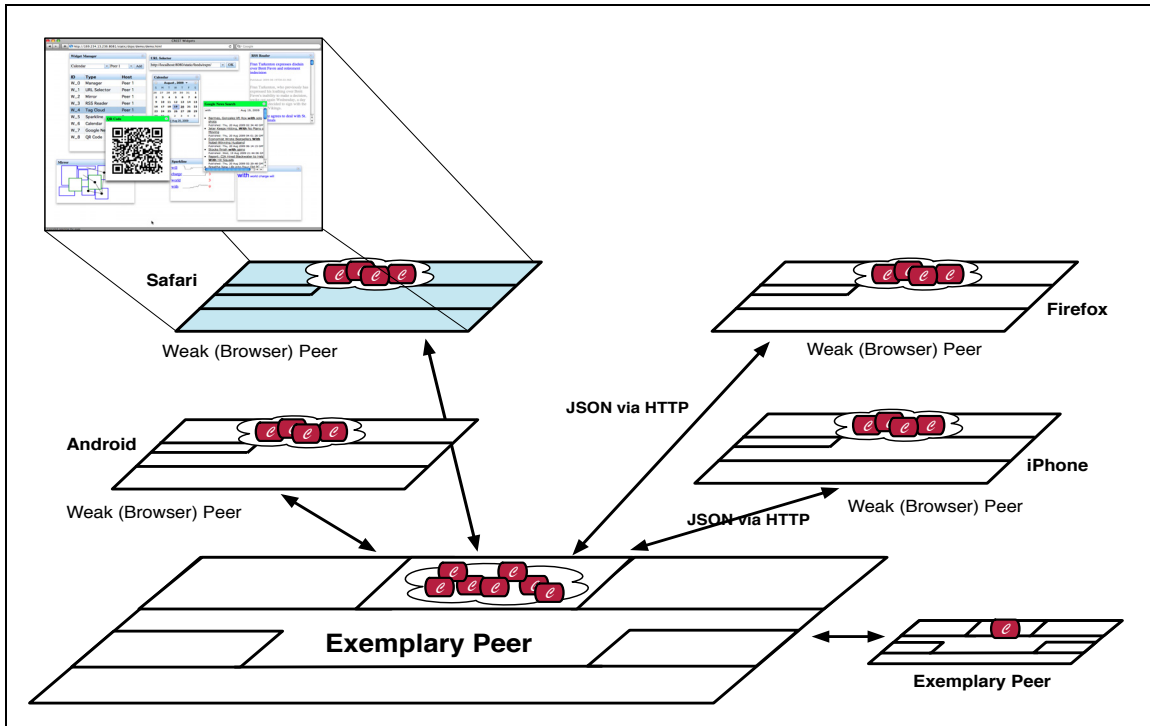


Figure 6.1: Screenshot of CREST-based Feed Reader application

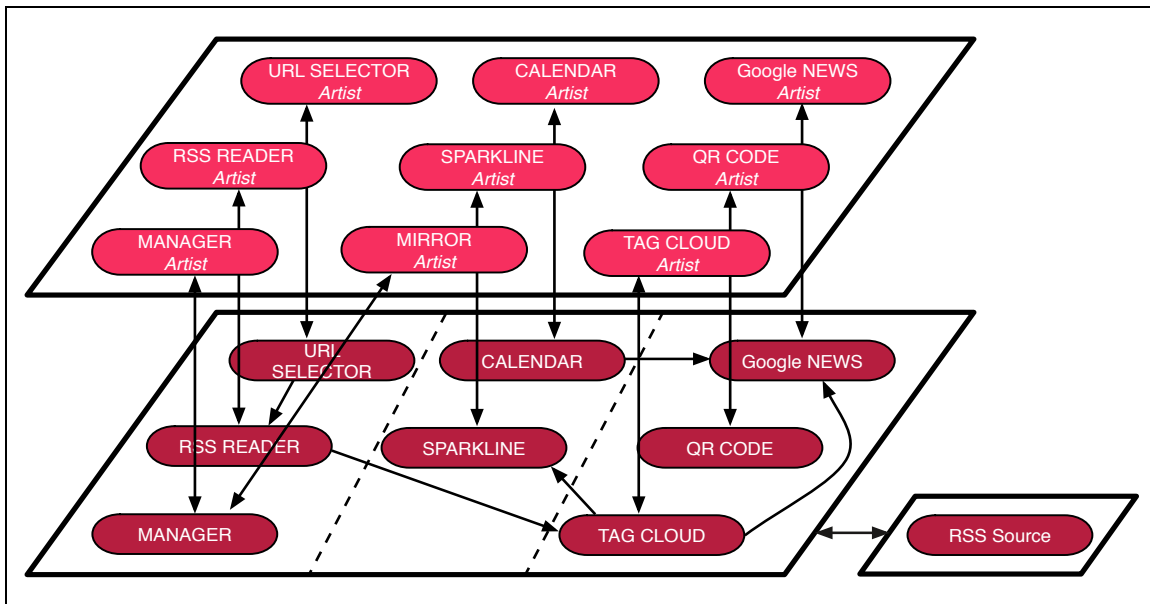
## 6.1. Example Application: Feed Reader

Our example application is a highly dynamic, re-configurable feed reader that consumes RSS or Atom feeds. For purposes of comparison, Google Reader or Bloglines may be considered as a starting reference point - however, our application has a much stronger computational and compositional aspect than either system offers today. In addition to merely displaying a feed, one such novel functionality present is that a user can dynamically link the feeds to a tag cloud to display the most popular words in the feed. A screenshot of the running application is presented in Figure 6.1 on page 160. A view of the computation models presented by our feed reader is seen in Figure 6.4 on page 162.

In our example, there are two separate classes of computations that are occurring: the *wid-  
get* computations running on the exemplary peers, and the *artist* computations running on

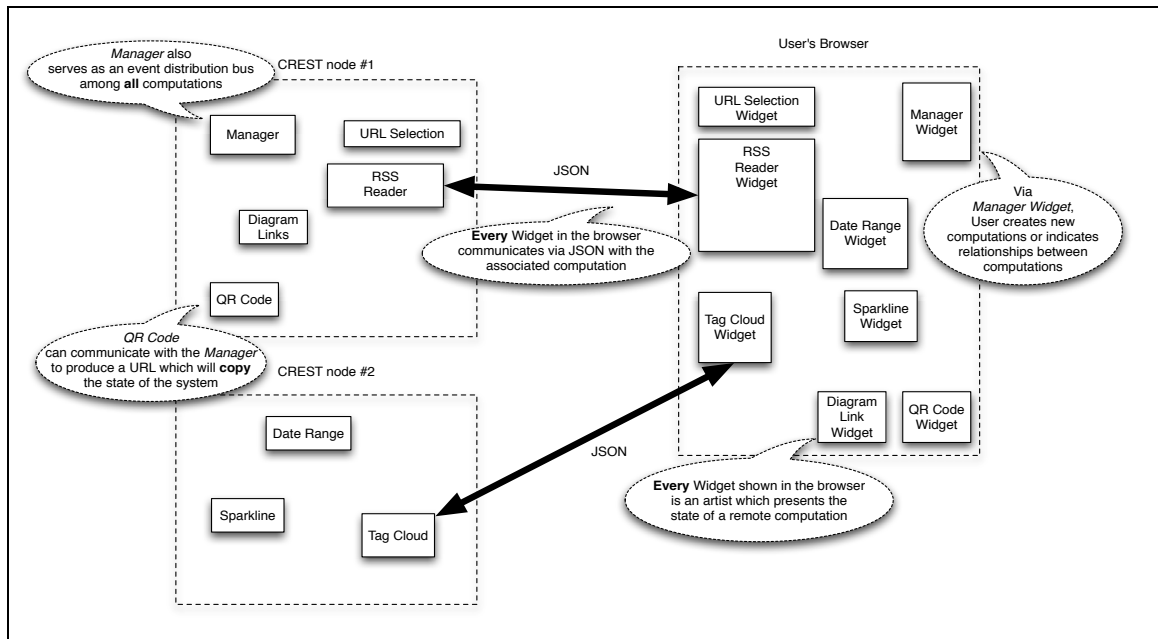


**Figure 6.2: Feed Reader Architecture**



**Figure 6.3: Feed Reader Computations (overview)**

the weak peers. An overview of the run-time architecture is provided in Figure 6.2 on page 161. There are eight different widgets: a manager (which allows a user, via a weak peer, to create and link widgets), an URL selector, an RSS reader, tag clouds, sparklines, a



**Figure 6.4: Feed Reader Computations (detail)**

calendar, a Google News reader, and a QR code (a 2D barcode format). Via a manager widget, these widgets can be linked together - such as the URL selector linking to the RSS reader, indicating that the reader should fetch its feed from a specific URL. Each one of these widget computations may have an associated artist computation which is responsible for visually rendering the state of the widget in a weak peer (such as a browser). In our example, the artists and widgets communicate via exchanging JSON over HTTP - more sophisticated computational exchanges (such as full closure and continuations) are commonly employed among exemplary peers. There does not have to be a one-to-one relationship between artist and widgets - in our example, a manager widget has two separate artists - one which lets the user add new widgets and another artist (the mirror) which visually depicts the entire state of the application using boxes, arrows, and color.

Using our CREST framework, all eight of our widgets total under 450 lines of Scheme code - the largest widget is the feed reader widget computation which is approximately

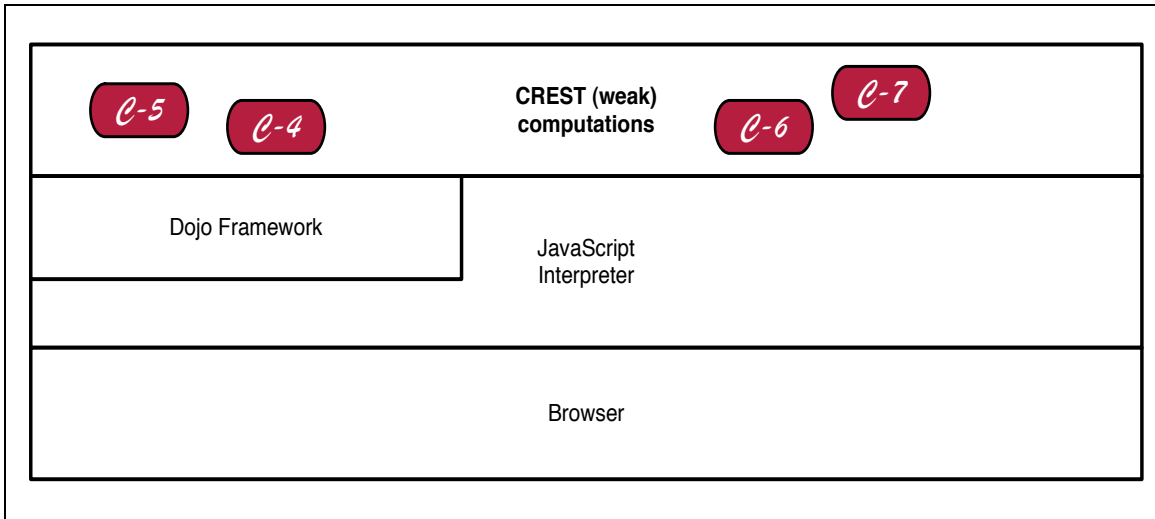


Figure 6.5: Weak CREST Peer

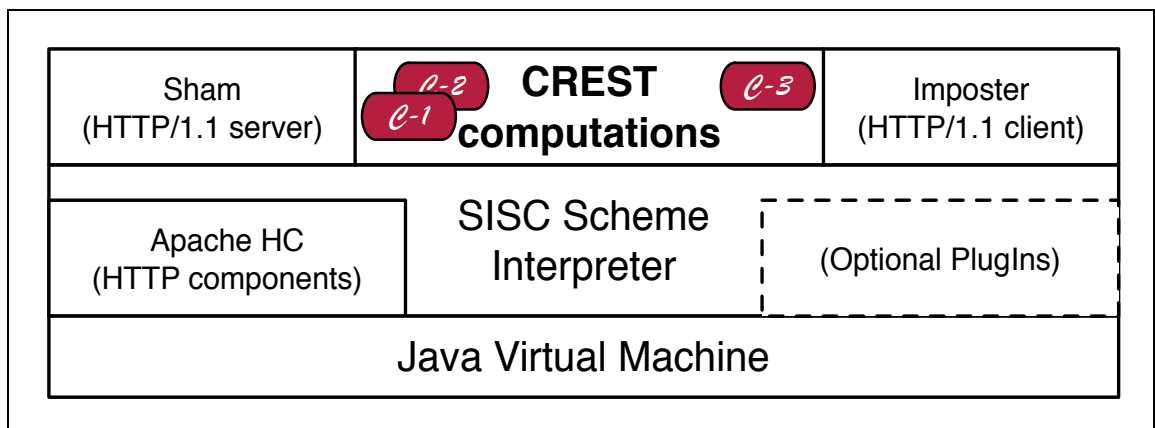


Figure 6.6: Exemplary CREST Peer

115 lines of code.<sup>1</sup> All of our artists are written on top of the Dojo JavaScript framework and comprise approximately 1,000 lines of JavaScript and HTML.

## 6.2. Design Considerations: Feed Reader

Using the considerations presented in Section 4.6 on page 141 as our guide, we now discuss their impact and how they manifest themselves in our demonstration example.

**Names.** Each instantiation of a widget computation has its own unique per-instance URL.

Through this URL, artist computations running in the confines of a weak peer or another

1. Appendix B contains the source code for two widgets: Manager and Tag Cloud.

widget computation on the same or different peer can deliver a message to a specific widget computation. These messages can either fetch the current state of the computation via a GET request (such as to retrieve the current tag cloud data) or update the state of the computation via a POST request (in order to deliver a new set of words to the tag cloud). Within the weak peer, artist computations have unique identifiers within the local DOM tree of the browser. However, these weak peers are restricted to only communicating with exemplary peers and can not expose any services or directly interact with other weak peers.

**Services.** Both the widget and artist computations in our example application are independently managed and run. Each widget computation locally defines what services it provides - such as the URL widget component permitting the storing and retrieval of the current value (in this case, a feed URL). In contrast, the widget manager computation offers the ability to spawn new widgets as well as maintaining information about already instantiated widgets.

It should also be noted that multiple artist computations may offer different perspectives on the same service. In our example, both the mirror and manager artist computations provide alternative views of the current state of the system (one as a list of existing widget computations, the other as a graphical representation of the existing computations).

**Time.** Over time, the computations offered by the example application will vary. The computations are not defined, created, or even initialized until they are explicitly activated by the manager widget. Additionally, there is a set of relationships between widget computations (links) that are created and destroyed over time. In our example, this set of rela-

tionship determines the data flow from one widget computation to another widget computation.

**State.** In our example application, the widget computations have the ability to maintain a local per-instance state. Our feed reader widget computation will retrieve and parse the designated feed on a periodic basis as defined by a clock-tick. The feed reader widget computation will then store the parsed feed as a local stateful value. By maintaining the state (essentially caching the parsed feed), the feed reader widget can easily scale since it must only return the cached representation rather than constantly retrieving and parsing the feed in response to a given request. In contrast to the widget computations, our artist computations are all stateless. The artist computations are configured to periodically poll their affiliated widget computation for its state and then renders that state accordingly.

**Computation.** The artist computation's responsibility is to (visually) render the state of its affiliated widget computation into a specific form. The widget computations executing on the exemplary peer have no restrictions on what they can compute modulo security implications as discussed in Section 6.3 on page 167. Through the use of links between widget computations, dynamic composibility of services is supported. For instance, the Google News widget can be dynamically linked to the calendar and tag cloud widget computations in order to search for a given keyword (from the tag cloud) at a given date in the past (from the calendar).

**Transparency.** The demonstration supports varying degrees of computational exchange, namely "shallow" copy versus "deep" copy. In a shallow copy, all weak peers share among themselves a single collection of widget computations (perhaps distributed among multiple exemplary peers) but have independent artist computations. In this case, all weak

peers see exactly the same state updates for the same widget computations. Under deep copy, a new weak peer creates a fresh, forked collection of widget computations whose computational states are a continuation, captured at the instant of the join, of the parent collection. This weak peer will now observe, from that point forward, an independent, evolving state. In this way, the deep copy creates a new version of the feed reader application at the time the continuation is created.

Interestingly, due to the accelerated timeframes of our example application (feeds were updated every three seconds), we had to introduce cache-busting parameters in our artist computations on the Google Android phones because of the browser's aggressive caching behavior.

**Migration and latency.** Since all relationships between computations are identified by a URL, these computations may either be remote or local. In this way, widget computations can be migrated with abandon as long as they are accessible via an URL. By adding multiple exemplary peers, dynamic widget migration and load sharing (computational exchange) allows the sample application to scale seamlessly.

With regards to latency, the division between artist computations (which draw the local display) and the widget computation (which maintains the state) allows for minimal transference of data between nodes. Upon creation of the widget computation, the relevant artist computation is transferred and instantiated on the weak peer. Therefore, by providing the artist computation as an output of the widget computation, the widget computation has complete control over what formats the artist computation require and can hence use any optimized format that it desires.

### 6.3. Additional Related Work

Web Services impose a service perspective on large-scale web systems, however, its flaws are numerous and fundamental including inappropriate service granularity [Stamos, 1990], an unsuitable invocation mechanism [Waldo, 1994], and intermingling of data and metadata causing high latency [Davis, 2002]. Google Wave [Lassen, 2009] employs a classic client/server architecture for which the medium of exchange is XML documents and state deltas encoded as operational transforms. CREST resembles a web-scale actor model [Baker, 1977] and like [Ghosh, 2009] exploits a functional language as the implementation medium for its actor-like semantics. The demonstration sketched above resembles Yahoo Pipes [Yahoo Inc., 2009] or Marmite [Wong, 2007] however, unlike these systems, all significant computation is distributed outside the browser in CREST peers and within the browser the only computation is for the sake of rendering.

Security is a significant issue for mobile code systems and several distinct mechanisms are relevant for CREST. Strong authentication is a vital starting point for trust, resource allocation, and session management and for which we will employ self-certifying URLs [Kaminsky, 1999]. Mechanisms for resource restriction, such as memory caps and processor and network throttling, are well-known, however, environment sculpting [Vyzovitis, 2002] may be used to restrict access to dangerous functions by visiting computations and is the functional analog of the capability restriction in Caja [Miller, 2008]. In addition, since an exemplary peer executes atop the Java Virtual Machine, all of the security and safety mechanisms of the JVM may be brought to bear. Finally, CREST exemplary peers will employ byte code verifiers for remote and spawn computations and various forms of



law-governed interaction [Minsky, 2000] to monitor and constrain the behavior of collections of computations executing internet-wide on multiple peers.

# CHAPTER 7

## Conclusion

### 7.1. Recap / Summary

**Research Question.** Throughout this dissertation, we can phrase our motivating question as: *What happens when dynamism is introduced into the Web?* In this characterization, we define dynamism as phenomena that must be explained as a manifestation of change, whether through interpretation or alteration of the interpreter. As we discover through our investigations of this question in this dissertation, we ultimately find that *the underlying architecture of the Web shifts, from a focus on the exchange of static content to the exchange of active computations.*

**CREST Architectural Style.** In order to support this shift, we construct a new architectural style called CREST (Computational REST). As discussed in much more detail in Chapter 4 starting on page 126, there are five core CREST axioms:

- CA1. A resource is a locus of computations, named by an URL.
- CA2. The representation of a computation is an expression plus metadata to describe the expression.
- CA3. All computations are context-free.
- CA4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.
- CA5. The presence of intermediaries is promoted.

**CREST Considerations.** We also encounter several recurring themes that must be addressed by a computation-centric Web and are related to the axioms above:

- Computations and their expressions are explicitly *named*. (CA1, CA2)
- *Services* may be exposed through a variety of URLs which offer perspectives on the same computation. (CA1); interfaces may offer complementary supervisory functionality such as debugging or management. (CA4)
- Functions may be added to or removed from the binding environment over *time* or their semantics may change. (CA4)
- Computational loci may be *stateful* (and thus permit indirect interactions between computations), but must also support *stateless* computations. (CA3)
- Potentially autonomous *computations* exchange and maintain state (CA2, CA3); A rich set of stateful relationships exist among a set of distinct URLs. (CA1)
- The computation is *transparent* and can be inspected, routed, and cached. (CA5)
- The *migration* of the computation to be physically closer to the data store is supported thereby reducing the impact of network *latency*. (CA2)

These themes were discussed in more detail in Chapter 4 starting on page 126.

**CREST Framework.** To both facilitate the adoption of CREST and to explore the implications and consequences of the style, we have constructed a CREST framework (discussed in detail in Chapter 6 starting on page 159) that allows us to build applications in this style. Utilizing this framework, we have constructed a feed reader application which offers novel computational and compositional aspects.

**Contributions.** In summary, this dissertation provides the following contributions:

- analysis of the essential architectural decisions of the World Wide Web, followed by generalization, opens up an entirely new space of decentralized, Internet-based applications

- recasting the web as a mechanism for computational exchange instead of content exchange
- a new architectural style to support this recasting (CREST)
- demonstrating how CREST better explains architectural dissonance
- a framework for building applications backed by CREST

## **7.2. Future Work**

The implementation framework, described in Chapter 6 starting on page 159, upon which the sample application (a feed reader) was built is generic, is fully backwards compatible with the existing Web infrastructure, and will be the basis for our next phase of investigation with CREST. We anticipate examining how this computational web can be used to solve problems in other areas, ranging from the smart energy grid to situational awareness to high-speed streaming video applications. In addition to these application-based studies, we anticipate investigations into numerous support areas, including development and testing techniques and tools, implementation of the security elements, and provision of services for computation search and composition. In the following section, we explore one particular area deserving future exploration: recombinant web services powered by CREST.

## **7.3. Future Work: Recombinant Web Services**

Based upon our prior observations of the current landscape of web services (either SOAP or REST-based), we are convinced that the future evolution of the Web will continue to embrace services as a necessary fundamental element of the web. However, the current form of web services (as described in Chapter 3 starting on page 108) is inadequate to

meet these challenges. We believe that CREST can play a vital role in guiding the creation of a new generation of web services. We will call these new services “recombinant” and define them by the following properties:

- Loose coupling with logical separation of the service vendor and the service consumer. The coupling between the two is limited to a remotely accessible, narrowly exposed interface.
- Composition, both within the boundary of a single vendor and among distinct vendors. Services may be offered that are themselves amalgams of lower-order services.
- Reuse, with a single service capable of use by many consumers for many purposes.
- Autonomy, the extent to which the functional behavior and performance of service is independent of other services, either local or remote, and the span of authority of the service vendor.
- Context-free exchanges, reducing to zero the degree to which the service is required to retain state or history of prior interactions.
- Monitoring, to observe, record, and analyze detailed service behavior across multiple service vendors simultaneously at any time from any location.

### **7.3.1. CREST properties for recombinant services**

In order to achieve these recombinant services, we can look towards CREST to provide some design guidance. We find four properties that CREST can deliver which are vital to the success of these recombinant services: *fluid locus of computation*, *local composition of computation*, *easy trading of computation*, and *free migration of computation*.

**Fluid locus of computation.** Since CREST supports the transfer of computation, there is no guarantee that an origin server to which a client issued a request (computation) will be

the eventual responder, since the server may choose to delegate the completion of the computation to a third-party. The locus of computation is therefore fluid; there is no fixed relationship between request and response since one request may generate zero, one, or many responses and, in the latter case, from many distinct servers.

**Local composition of computation.** In CREST, the available functional substrate offered by a specific node can be remarkably fine-grained with rich dynamic compositional capabilities. Composition of computations, say  $g(f(x))$ , can synthesize a final result from intermediate CREST computations—a sharp contrast to Web Services which offer no local compositional logic. With CREST, this composition is both dynamic and local and uses common programming language primitives. Similarly, a CREST client can stitch partial computations from origin servers into a comprehensive whole as no one origin server may be able to supply all of the functional capability that the client requires.

**Easy trading of computation.** Under CREST URLs name specific computations—in the near-literal sense that the entirety of the computation is embedded in the URL. Therefore, if an individual wants to share a computation, exchanging URLs is sufficient. Moreover, this trading of computations can be done programmatically and across agency borders.

**Free migration of computation.** CREST servers can be easily synthesized or decomposed as physical or agency constraints warrant. A logical origin server may be dynamically reconstituted as a physical cooperative of servers with portions of the computation divided and delegated among many CREST peers. For example, certain functions, such as highly complex mathematical operations, can be delegated to special-purpose CREST servers (which possess optimized libraries or dedicated co-processors) to produce intermediate computations. Alternatively, CREST servers can act as intelligent load-balancers

directing requests based on traffic analysis. There is no requirement that such delegation must remain within a single agency sphere. An operator of a CREST service could contract with a third-party to perform specific portions of the computation unbeknownst to any one that uses its resources.

## REFERENCES

1. Aas, G. libwww-perl. <<http://lwp.linpro.no/lwp/>>, HTML, June 25, 2004.
2. Abelson, H., Sussman, G.J., and Sussman, J. Structure and Interpretation of Computer Programs. Second ed. 683 pgs., MIT Press, 1996.
3. Amazon Web Services. Amazon S3. <<http://docs.amazonwebservices.com/AmazonS3/2006-03-01/>>, March 1, 2006.
4. 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.
5. Apple Computer Inc. Introduction to Web Kit Plug-in Programming Topics. <[http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit\\_PluginProgTopic/index.html](http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit_PluginProgTopic/index.html)>, HTML, August 11, 2005.
6. ---. About Web Browser Plug-ins. <[http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit\\_PluginProgTopic/Concepts/AboutPlug-ins.html](http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit_PluginProgTopic/Concepts/AboutPlug-ins.html)>, HTML, August 11, 2005.
7. ---. Introduction to Web Kit Objective-C Programming Guide. <<http://developer.apple.com/documentation/Cocoa/Conceptual/DisplayWebContent/index.html>>, HTML, April 29, 2005.
8. ---. URL Loading System Overview. <<http://developer.apple.com/documentation/Cocoa/Conceptual/URLLoadingSystem/Concepts/URLOverview.html>>, HTML, October 8, 2005.
9. ---. Creating Plug-ins with Cocoa and the Web Kit. <[http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit\\_PluginProgTopic/Concepts/AboutPlugins.html](http://developer.apple.com/documentation/InternetWeb/Conceptual/WebKit_PluginProgTopic/Concepts/AboutPlugins.html)>, HTML, August 11, 2005.
10. ---. Mac OS X - Dashboard. <<http://www.apple.com/macosx/features/dashboard/>>, HTML, 2005.
11. Baker, H. and Hewitt, C. Laws for Communicating Parallel Processes. MIT Artificial Intelligence Laboratory Working Papers, Report WP-134A, May 10, 1977. <<http://hdl.handle.net/1721.1/41962>>.
12. Baker, M. Browser Innovation, Gecko and the Mozilla Project. <<http://www.mozilla.org/browser-innovation.html>>, HTML, February 25, 2003.
13. Bandhauer, J. XPJS Components Proposal. <<http://www.mozilla.org/scriptable/xpjs-components.html>>, HTML, July 1, 1999.
14. Bass, L., Clements, P., and Kazman, R. Software Architecture in Practice. SEI Series in Software Engineering. Addison Wesley: Reading, MA, 1998.
15. Bellwood, T., Clément, L., Ehnebuske, D., Hatley, A., Hondo, M., Husband, Y.L., Januszewski, K., Lee, S., McKee, B., Munter, J., and von Riegen, C. UDDI Version 3.0. <[http://uddi.org/pubs/uddi\\_v3.htm](http://uddi.org/pubs/uddi_v3.htm)>, 2002.
16. Blizzard, C. Untitled. <<http://www.0xdeadbeef.com/html/2003/01/index.shtml#20030114>>, HTML, January 14, 2003.
17. BowmanSoft. Mastering Internet Explorer: The Web Browser Control. Visual Basic Web Magazine. 2001. <[http://www.vbwm.com/art\\_2001/IE05/](http://www.vbwm.com/art_2001/IE05/)>.



18. 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.
19. 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>>.
20. Cardoso, J. Complexity analysis of BPEL Web processes. Software Process: Improvement and Practice. 12(1), p. 35-49, 2007. <<http://dx.doi.org/10.1002/spip.302>>.
21. Caswell-Daniels, M. Googles Flight Sim. <<http://www.isoma.net/games/goggles.html>>, HTML, 2007.
22. 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](http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html)>.
23. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the OSDI'06: Seventh Symposium on Operating System Design and Implementation. Seattle, WA, November, 2006. <<http://labs.google.com/papers/bigtable-osdi06.pdf>>.
24. 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>>.
25. 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.
26. 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.
27. Colan, M. Service-Oriented Architecture expands the vision of Web services, Part 1. <<http://www.ibm.com/developerworks/library/ws-soaintro.html>>, 2004.
28. CollabNet. Eyebrowse. <<http://eyebrowse.tigris.org/>>, HTML, 2006.
29. ---. Subversion. <<http://subversion.tigris.org/>>, HTML, 2008.
30. Cook, M. Securing I.I.S. 5 and 6 Server. <[http://escarpment.net/training/Securing\\_Microsoft\\_IIS\\_5\\_and\\_6\(slides\).pdf](http://escarpment.net/training/Securing_Microsoft_IIS_5_and_6(slides).pdf)>, PDF, February 14, 2005.
31. Davis, A., Parikh, J., and Weihl, W.E. Edgecomputing: Extending Enterprise Applications to the Edge of the Internet. In Proceedings of the 13th International World Wide Web Conference. p. 180-187, New York, NY, USA, 2004. <<http://doi.acm.org/10.1145/1013367.1013397>>.
32. Davis, D. and Parashar, M. Latency Performance of SOAP Implementations. In Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. p. 407-12, Berlin, Germany, May 21-24, 2002.
33. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Voshall, P., and Vogels, W. Dynamo: Amazon's highly

- available key-value store. ACM SIGOPS Operating Systems Review. 41(6), p. 205-220, 2007. <<http://doi.acm.org/10.1145/1323293.1294281>>.
34. Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. Globally distributed content delivery. IEEE Internet Computing. 6(5), p. 50-58, September, 2002. <<http://dx.doi.org/10.1109/MIC.2002.1036038>>.
  35. Dubois, M., Scheurich, C., and Briggs, F. Memory access buffering in multiprocesses. In Proceedings of the 13th Annual International Symposium on Computer Architecture. p. 434-442, Tokyo, Japan, 1986. <<http://doi.acm.org/10.1145/17407.17406>>.
  36. E-Soft Inc. Cookie Survey - Usage of Cookies on the Internet. <[http://www.securityspace.com/s\\_survey/data/man.200706/cookieReport.html](http://www.securityspace.com/s_survey/data/man.200706/cookieReport.html)>, 2007.
  37. eBay, I. REST - eBay Developers Program. <<http://developer.ebay.com/developer-center/rest>>, 2007.
  38. Eich, B. and Clary, B. JavaScript Language Resources. <<http://www.mozilla.org/js/language/>>, HTML, January 24, 2003.
  39. Eich, B. New Roadmaps. <<http://weblogs.mozillazine.org/roadmap/archives/009218.html>>, HTML, November 3, 2005.
  40. ---. 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>>.
  41. Erenkrantz, J.R. Create branch for repository mirroring? <<http://svn.haxx.se/dev/archive-2002-12/0905.shtml>>, December 13, 2002.
  42. ---. Serf and Subversion. <<http://svn.haxx.se/dev/archive-2005-11/1373.shtml>>, Email, November 27, 2005.
  43. 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.
  44. Evans, E. Gecko Embedding Basics. 2002. <<http://www.mozilla.org/projects/embedding/embedoverview/EmbeddingBasics.html>>.
  45. Faure, D. Chapter 13: Creating and Using Components (KParts). In KDE 2.0 Development, Sweet, D. ed. Sams Publishing, 2000.
  46. Festa, P. Apple snub stings Mozilla. CNet News. January 14, 2003. <<http://news.com.com/2100-1023-980492.html>>.
  47. Fielding, R.T. libwww-perl: WWW Protocol Library for Perl. <<http://ftp.ics.uci.edu/pub/websoft/libwww-perl/>>, HTML, June 25, 1998.
  48. ---. libwww-ada95: WWW Protocol Library for Ada95. <<http://ftp.ics.uci.edu/pub/websoft/libwww-ada95/>>, HTML, May 13, 1998.
  49. ---. Onions Network Streams Library. <<http://ftp.ics.uci.edu/pub/websoft/libwww-ada95/current/Onions/README>>, Text, May 13, 1998.
  50. ---. Shared Leadership in the Apache Project. Communications of the ACM. 42(4), p. 42-43, 1999.
  51. ---. 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>>.

52. 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.
53. Fielding, R.T. REPORT method considered harmful. <<http://lists.w3.org/Archives/Public/www-tag/2005Dec/0131.html>>, HTML, December 28, 2005.
54. Foster, I., Kesselman, C., Nick, J.M., and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure WG, Global Grid Forum, 2002*. <<http://www.globus.org/alliance/publications/papers/ogsa.pdf>>.
55. Free Software Foundation. Autoconf. <<http://www.gnu.org/software/autoconf/>>, HTML, February 2, 2005.
56. Frystyk Nielsen, H. W3C Libwww Review. W3C, Report, June, 1999. <<http://www.w3.org/Talks/1999/06/libwww/>>.
57. Fu, X., Bultan, T., and Su, J. Analysis of interacting BPEL web services. In *13th International Conference on World Wide Web*. p. 621-630, ACM: New York, NY, USA, 2004. <<http://doi.acm.org/10.1145/988672.988756>>.
58. Fuchs, M. Dreme: for Life in the Net. PhD Thesis. New York University, 1995.
59. Fuggetta, A., Picco, G.P., and Vigna, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering*. 24(5), p. 342-361, May, 1998.
60. Gal, A. Efficient Bytecode Verification and Compilation in a Virtual Machine. Ph.D. Thesis. University of California, Irvine, 2006.
61. Garrett, J.J. Ajax: A New Approach to Web applications. <<http://www.adaptive-path.com/publications/essays/archives/000385.php>>, HTML, February 18, 2005.
62. Gaudet, D. Apache Performance Notes. <<http://httpd.apache.org/docs/1.3/misc/perf-tuning.html>>, HTML, September 30, 1997.
63. Ghosh, D. and Vinoski, S. Scala and Lift - Functional Recipes for the Web. *IEEE Internet Computing*. 13(3), p. 88-92, 2009. <<http://doi.ieeecomputersociety.org/10.1109/MIC.2009.68>>.
64. 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.
65. 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.
66. Google. Google Maps API. <<http://code.google.com/apis/maps/>>, 2007.
67. ---. Design Elements - V8 JavaScript Engine. <<http://code.google.com/apis/v8/design.html>>, 2008.
68. Gosling, J. and Yellin, F. Window Toolkit and Applets. *The Java(TM) Application Programming Interface*. 2, 406 pgs., Addison-Wesley Professional, 1996.
69. Granroth, K. Embedded Components Tutorial. <<http://www.konqueror.org/componentstutorial/>>, HTML, March 3, 2000.
70. Graunke, P.T. and Krishnamurthi, S. Advanced Control Flows for Flexible Graphical User Interfaces: or, Growing GUIs on Trees or, Bookmarking GUIs. In *Proceedings of the 24th International Conference on Software Engineering*. p. 277-287, ACM Press. New York, NY, USA, 2002.

71. Grenness, C. Varnish 2.0: Speed up your web site and cut costs. <[http://varnish-cache.com/en/news/varnish\\_2\\_0\\_speed\\_up\\_your\\_web\\_site\\_and\\_cut\\_costs](http://varnish-cache.com/en/news/varnish_2_0_speed_up_your_web_site_and_cut_costs)>, 2008.
72. 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](http://f-m-c.org/publications/download/groene_et_al_2002-architecture_recovery_of_apache.pdf)>.
73. 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](http://f-m-c.org/projects/apache/download/the_apache_modelling_project.pdf)>.
74. Grosskurth, A. and Echiabi, A. 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>>.
75. ---. Concrete Architecture of Mozilla. University of Waterloo, Report, October 24, 2004. <<http://www.cs.uwaterloo.ca/~agrossku/2004/cs746/mozilla-concrete.pdf>>.
76. 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.
77. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H.F. Simple Object Access Protocol (SOAP) 1.2: Adjuncts. <<http://www.w3.org/TR/soap12-part2/>>, W3C, HTML, 2003.
78. Ha, V., Inkpen, K.M., Al Shaar, F., and Hdeib, L. An Examination of User Perception and Misconception of Internet Cookies. In Proceedings of the CHI 2006. p. 833-838, Montréal, Québec, Canada, April 22-27, 2006. <<http://doi.acm.org/10.1145/1125451.1125615>>.
79. Halls, D.A. Applying Mobile Code to Distributed Systems. Ph.D. Thesis Thesis. University of Cambridge, 1997.
80. Harris, W. and Potts, R. Necko: A new netlib kernel architecture. <<http://www.mozilla.org/docs/netlib/necko.html>>, HTML, April 14, 1999.
81. 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>>.
82. Hood, E. MHonArc. <<http://www.mhonarc.org/>>, HTML, May 17, 2004.
83. Hyatt, D. iTunes and WWebKit. <[http://weblogs.mozillazine.org/hyatt/archives/2004\\_06.html#005666](http://weblogs.mozillazine.org/hyatt/archives/2004_06.html#005666)>, HTML, June 8, 2004.
84. Kahan, J. Libwww - the W3C Sample Code Library. <<http://www.w3.org/Library/>>, W3C, HTML, September, 2003.
85. Kaminsky, M. and Banks, E. SFS-HTTP: Securing the Web with Self-Certifying URLs. MIT Laboratory for Computer Science, 1999. <<http://pdos.csail.mit.edu/~kaminsky/sfs-http.ps>>.
86. Katsaros, D., Pallis, G., Stamos, K., Vakali, A., Sidiropoulos, A., and Manolopoulos, Y. CDNs: Content Outsourcing via Generalized Communities. IEEE Transactions on Knowledge and Data Engineering. 21(1), p. 137-151, 2009.
87. 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](http://dx.doi.org/10.1016/0169-7552(94)90129-5)>.

88. KDE e.V. Reaktivite Released. <<http://www.konqueror.org/announcements/reaktivate.php>>, HTML, July 9, 2001.
89. ---. Konqueror FAQ. <<http://konqueror.kde.org/faq/>>, HTML, 2005.
90. Khare, R. Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems. PhD Thesis. Information and Computer Science, University of California, Irvine, 2003.
91. 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>>.
92. Koch, P.-P. Browsers - Explorer 5 Mac. <<http://www.quirksmode.org/browsers/explorer5mac.html>>, HTML, November 23, 2004.
93. 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>>.
94. 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>>.
95. Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM. 21(7), p. 558-565, July, 1978. <<http://doi.acm.org/10.1145/359545.359563>>.
96. Larsson, A. The Life of An HTML HTTP Request. <[http://www.mozilla.org/docs/url\\_load.html](http://www.mozilla.org/docs/url_load.html)>, Mozilla Foundation, HTML, October 8, 1999.
97. Lassen, S. and Thorogood, S. Google Wave Federation Architecture. <<http://www.waveprotocol.org/whitepapers/google-wave-architecture>>, 2009.
98. Lawrence, E. Fiddler PowerToy - Part 1: HTTP Debugging. <[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebgen/html/IE\\_IntroFiddler.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebgen/html/IE_IntroFiddler.asp)>, Microsoft Corporation, HTML, January, 2005.
99. Leighton, T. Improving performance on the Internet. Communications of the ACM. 52(2), p. 44-51, February, 2009. <<http://doi.acm.org/10.1145/1461928.1461944>>.
100. 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>>.
101. MacVittle, L. F5 BIG-IP v10. <<http://www.f5.com/pdf/white-papers/big-ip-v10-wp.pdf>>, 2009.
102. 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/>>.
103. Matthews, J., Findler, R.B., Graunke, P., Krishnamurthi, S., and Felleisen, M. Automatically Restructuring Programs for the Web. Automated Software Engineering. 11(4), p. 337--364, 2004.
104. McFarlane, N. Rapid Application Development with Mozilla. 704 pgs., Prentice Hall, 2003. <<http://mb.eschew.org/>>.

105. 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.
106. Mesbah, A. and van Deursen, A. A Component- and Push-based Architectural Style for Ajax Applications. *Journal of Systems and Software*. 81(12), p. 2194-2209, December, 2008. <<http://dx.doi.org/10.1016/j.jss.2008.04.005>>.
107. ---. Invariant-Based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. p. 210-220, Vancouver, BC, Canada, 2009. <<http://dx.doi.org/10.1109/ICSE.2009.5070522>>.
108. Microsoft Corporation. IIS 6.0 Operations Guide (IIS 6.0). <<http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/f74c464e-6d5d-403c-97e7-747cd798dde2.msp>>, Microsoft Windows Server 2003 TechCenter, HTML.
109. ---. Adding Toolbar Buttons. <<http://msdn.microsoft.com/workshop/browser/ext/tutorials/button.asp>>, Internet Explorer - Browser Extensions, HTML.
110. ---. Implementing a Custom Download Manager. <<http://msdn.microsoft.com/workshop/browser/ext/overview/downloadmgr.asp>>, Internet Explorer - Browser Extensions, HTML.
111. ---. 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](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/shellcc/platform/Shell/programmersguide/shell_adv/bands.asp)>, Advanced Shell Techniques, HTML.
112. ---. Web Accessories. <<http://msdn.microsoft.com/workshop/browser/accessory/overview/overview.asp>>, Internet Explorer - Browser Extensions, HTML.
113. ---. HTTP Protocol Stack (IIS 6.0). <<http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/a2a45c42-38bc-464c-a097-d7a202092a54.msp>>, Microsoft Windows Server 2003 TechCenter, HTML.
114. ---. What's Changed (IIS 6.0). <<http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/7b037954-441d-4037-a111-94df7880c319.msp>>, Microsoft Windows Server 2003 TechCenter, HTML.
115. ---. Configuring Isolation Modes (IIS 6.0). <<http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/ed3c22ba-39fc-4332-bdb7-a0d9c76e4355.msp>>, Microsoft Windows Server 2003 TechCenter, HTML.
116. ---. Adding Menu Items. <<http://msdn.microsoft.com/workshop/browser/ext/tutorials/menu.asp>>, Internet Explorer - Browser Extensions, HTML.
117. ---. Application Isolation Modes (IIS 6.0). <<http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/7b037954-441d-4037-a111-94df7880c319.msp>>, Microsoft Windows Server 2003 TechCenter, HTML.
118. ---. Asynchronous Pluggable Protocols. <<http://msdn.microsoft.com/workshop/networking/pluggable/pluggable.asp>>, Internet Explorer - Asynchronous Pluggable Protocols, HTML.
119. ---. Reusing the WebBrowser Control. <<http://msdn.microsoft.com/workshop/browser/webbrowser/webbrowser.asp>>, Internet Explorer - WebBrowser, HTML.
120. ---. 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.

121. ---. 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.
122. ---. WebBrowser Object. <<http://msdn.microsoft.com/workshop/browser/web-browser/reference/objects/webbrowser.asp>>, Internet Explorer - WebBrowser, HTML.
123. ---. Designing Secure ActiveX Controls. <<http://msdn.microsoft.com/workshop/components/activex/security.asp>>, Internet Development, HTML.
124. ---. 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.
125. ---. 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.
126. ---. Introduction to ActiveX Controls. <<http://msdn.microsoft.com/workshop/components/activex/intro.asp>>, Internet Development, HTML.
127. ---. How To: Implementing Cookies in ISAPI. <<http://support.microsoft.com/kb/q168864/>>, HTML, July 1, 2004.
128. ---. 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.
129. ---. 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.
130. ---. 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.
131. ---. Internet Explorer 5 for Mac. <<http://www.microsoft.com/mac/products/internetexplorer/internetexplorer.aspx>>, HTML, December 19, 2005.
132. ---. ASP 200 Error Setting Cookie Expiration Past January 19, 2038. <<http://support.microsoft.com/kb/247348>>, HTML, November 21, 2006.
133. Miller, M.S., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized JavaScript. <<http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf>>, PDF, June 7, 2008.
134. Miller, S.G. SISC: A complete Scheme interpreter in Java. Technical Report, 2003. <<http://sisc.sourceforge.net/sisc.pdf>>.
135. Miniwatts Marketing Group. Internet Usage Statistics: The Internet Big Picture. <<http://www.internetworldstats.com/stats.htm>>, 2009.
136. Minsky, N.H. and Ungureanu, V. Law-governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. ACM Transactions on Software Engineering Methodology. 9(3), p. 273-305, July, 2000. <<http://doi.acm.org/10.1145/352591.352592>>.
137. Mitchell, K. A Matter of Style: Web Services Architectural Patterns. In Proceedings of the XML 2002. Baltimore, MD, December 8-13, 2002.

138. Mitra, N. Simple Object Access Protocol (SOAP) 1.2: Primer. <<http://www.w3.org/TR/soap12-part0/>>, W3C, HTML, June 24, 2003.
139. 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.
140. 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>>.
141. Mozilla Foundation. NSPR: Module Description. <<http://www.mozilla.org/projects/nspr/about-nspr.html>>, HTML, September 20, 2000.
142. ---. SeaMonkey Milestones. <<http://www.mozilla.org/projects/seamoney/milestones/index.html>>, HTML, February 9, 2001.
143. ---. Old Milestone Releases. <<http://www.mozilla.org/projects/seamoney/release-notes/>>, HTML, November 27, 2002.
144. ---. Mozilla Embedding FAQ. <<http://www.mozilla.org/projects/embedding/faq.html>>, HTML, December 8, 2004.
145. National Center for Supercomputing Applications. NCSA HTTPd Tutorial: CGI Configuration. <<http://hoohoo.ncsa.uiuc.edu/docs/tutorials/cgi.html>>, HTML, September 27, 1995.
146. ---. 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.
147. ---. CGI: Common Gateway Interface. <<http://hoohoo.ncsa.uiuc.edu/cgi/intro.html>>, HTML, December 6, 1995.
148. ---. 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.
149. ---. NCSA Mosaic Version History. <<http://www.ncsa.uiuc.edu/Divisions/PublicAffairs/MosaicHistory/history.html>>, HTML, November 25, 2002.
150. Netcraft. Netcraft Web Server Survey. <<http://www.netcraft.com/survey/>>, HTML, December 2, 2009.
151. Netscape. JavaScript Guide. <<http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/>>, HTML, 1996.
152. ---. Client Side State - HTTP Cookies. <[http://wp.netscape.com/newsref/std/cookie\\_spec.html](http://wp.netscape.com/newsref/std/cookie_spec.html)>, HTML, 1999.
153. 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/>>.
154. Nielsen, H.F. Threads and Event Loops. <<http://www.w3.org/Library/User/Architecture/Events.html>>, HTML, July 13, 1999.
155. Nottingham, M. Understanding Web Services Attachments. BEA Dev2Dev. May 24, 2004. <[http://dev2dev.bea.com/pub/a/2004/05/websvcs\\_nottingham.html](http://dev2dev.bea.com/pub/a/2004/05/websvcs_nottingham.html)>.



156. O'Reilly, T. REST vs. SOAP at Amazon. <<http://www.oreillynet.com/pub/wlg/3005>>, April 3, 2003.
157. Oakland Software Incorporated. Java HTTP Client - Comparison. <[http://www.oaklandsoftware.com/product\\_16compare.html](http://www.oaklandsoftware.com/product_16compare.html)>, HTML, April, 2005.
158. Oeschger, I. API Reference: Netscape Gecko Plugins. 2.0 ed. 190 pgs., Netscape Communications, 2002. <<http://www.ics.uci.edu/~jerenkra/netscape-plugin.pdf>>.
159. Orton, J. neon HTTP and WebDAV client library. <<http://www.webdav.org/neon/>>, HTML, 2005.
160. Palankar, M.R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. Amazon S3 for science grids: a viable solution? In Proceedings of the 2008 International Workshop on Data-aware Distributed Computing. p. 55-64, Boston, MA, 2008. <<http://doi.acm.org/10.1145/1383519.1383526>>.
161. Pallis, G. and Vakali, A. Insight and Perspectives for Content Delivery Networks. Communications of the ACM. 49(1), p. 101-106, 2006. <<http://doi.acm.org/10.1145/1107458.1107462>>.
162. Papazoglou, M.P. Service-Oriented Computing: Concepts, Characteristics and Directions. Fourth International Conference on Web Information Systems Engineering (WISE'03). p. 3, 2003.
163. Papazoglou, M.P. and Georgakopoulos, D. Service-Oriented Computing. Communications of the ACM. 46(10), p. 25-28, October, 2003. <<http://doi.acm.org/10.1145/944217.944233>>.
164. Parrish, R. XPCOM Part 1: An Introduction to XPCOM. developerWorks. February 1, 2001. <<http://www-128.ibm.com/developerworks/webservices/library/co-xpcom.html>>.
165. 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>>.
166. Petterson, Y. A context mechanism for controlling caching of HTTP responses. Internet Engineering Task Force, Report, November 3, 2008. <<http://tools.ietf.org/html/draft-pettersen-cache-context-03>>.
167. Prescod, P. Some thoughts about SOAP versus REST on Security. <<http://www.prescod.net/rest/security.html>>, 2002.
168. Quantcast. Methodology FAQ. <<http://www.quantcast.com/docs/display/info/Methodology+FAQ>>, 2009.
169. Queinnec, C. The Influence of Browser on Evaluators or, Continuations to Program Web Servers. In Proceedings of the International Conference on Functional Programming. Montreal, Canada, 2000.
170. Raggett, D. HTML+ (Hypertext markup format). <[http://www.w3.org/MarkUp/HTMLPlus/htmlplus\\_1.html](http://www.w3.org/MarkUp/HTMLPlus/htmlplus_1.html)>, HTML, November 8, 1993.
171. Richardson, L. and Ruby, S. RESTful Web Services. 419 pgs., O'Reilly Media, Inc., 2007.
172. Ritchie, D.M. A Stream Input-Output System. AT&T Bell Laboratories Technical Journal. 63(8 Part 2), p. 1897-1910, October, 1984.
173. Rosenberg, D. Adding a New Protocol to Mozilla. 2004. <<http://www.nexgenmedia.net/docs/protocol/>>.

174. Rousskov, A. and Soloviev, V. A performance study of the Squid proxy on HTTP/1.0. *World Wide Web*. 2(1-2), p. 47-67, June, 1999. <<http://dx.doi.org/10.1023/A:1019240520661>>.
175. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., and Saint-Paul, R. Supporting the dynamic evolution of Web service protocols in service-oriented architectures. *ACM Transactions on the Web*. 2(2), p. 13, 2008. <<http://dx.doi.org/10.1145/1346237.1346241>>.
176. 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/>>.
177. Saroiu, S., Gummadi, K.P., Dunn, R.J., Gribble, S.D., and Levy, H.M. An analysis of Internet content delivery systems. *ACM SIGOPS Operating Systems Review*. 36(21), p. 315-327, Winter, 2002. <<http://doi.acm.org/10.1145/844128.844158>>.
178. 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.
179. Schmidt, J. *ISAPI*. Microsoft Internet Developer. Spring, 1996. <<http://www.microsoft.com/mind/0396/ISAPI/ISAPI.asp>>.
180. Sergeant, M. Using Qpsmtpd. *O'Reilly SysAdmin*. September 15, 2005. <<http://www.oreillynet.com/pub/a/sysadmin/2005/09/15/qpsmtpd.html>>.
181. Shaver, M. back; popular demand. <<http://shaver.off.net/diary/2003/01/15/back-popular-demand/>>, HTML, January 15, 2003.
182. Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. 242 pgs., Prentice Hall, 1996.
183. 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.
184. Smith, J. Optimizing and Performance Tuning IIS 6.0. *Informit.com*. September 10, 2004. <<http://www.informit.com/articles/article.asp?p=335881>>.
185. Smith, J.M. Fighting Physics. *Communications of the ACM*. 52(7), p. 60-65, 2009. <<http://dx.doi.org/10.1145/1538788.1538806>>.
186. Spolsky, J. Things You Should Never Do, Part I. <<http://www.joelonsoftware.com/articles/fog0000000069.html>>, Joel On Software, HTML, April 6, 2000.
187. Stachowiak, M. WebKit achieves Acid3 100/100 in public build. <<http://webkit.org/blog/173/webkit-achieves-acid3-100100-in-public-build/>>, 2008.
188. Stamos, J.W. and Gifford, D.K. Remote Evaluation. *ACM Transactions on Programming Languages & Systems*. 12(4), p. 537-564, 1990. <<http://doi.acm.org/10.1145/88616.88631>>.
189. Stein, G. and Erenkrantz, J. Serf Design Guide. <<http://svn.webdav.org/repos/projects/serf/trunk/design-guide.txt>>, HTML, September 14, 2004.
190. Stein, L. and Stewart, J. *WWW Security FAQ: Client Side Security*. <<http://www.w3.org/Security/Faq/wwwsf2.html>>, 2003.
191. Stenberg, D. libcurl vs neon for WebDav? <<http://curl.haxx.se/mail/lib-2003-03/0208.html>>, Email, March 19, 2003.
192. ---. High Performance libcurl - hiper. <<http://curl.haxx.se/libcurl/hiper/>>, HTML, December 26, 2005.

193. ---. Programs Using libcurl. <<http://curl.haxx.se/libcurl/using/apps.html>>, HTML, January 5, 2006.
194. ---. cURL - Frequently Asked Questions. <<http://curl.haxx.se/docs/faq.html>>, HTML, January 5, 2006.
195. Suryanarayana, G., Diallo, M.H., Erenkrantz, J.R., and Taylor, R.N. Architectural Support for Trust Models in Decentralized Applications. In Proceedings of the 28th International Conference on Software Engineering (ICSE 2006). p. 52-61, Shanghai, China, May, 2006.
196. Tarau, P. and Dahl, V. High-level Networking with Mobile Code and First-Order AND-Continuations. Theory and Practice of Logic Programming. 1(3), p. 359--380, May, 2001.
197. Thau, R.S. Notes on the Shambhala API. <[http://mail-archives.apache.org/mod\\_mbox/httpd-dev/199507.mbox/%3c9507051409.AA08582@volterra%3e](http://mail-archives.apache.org/mod_mbox/httpd-dev/199507.mbox/%3c9507051409.AA08582@volterra%3e)>, Email, July 5, 1995.
198. ---. 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](http://dx.doi.org/10.1016/0169-7552(96)00048-7)>.
199. The Apache HTTP Server Project. Apache MPM event. <<http://httpd.apache.org/docs/2.2/mod/event.html>>, HTML, December 5, 2005.
200. The Apache Portable Runtime Project. Apache Portable Runtime Project. <<http://httpd.apache.org/dev/guidelines.html>>, HTML, November 20, 2004.
201. The Apache Software Foundation. Apache API notes. <<http://httpd.apache.org/docs/1.3/misc/API.html>>, HTML, October 13, 2003.
202. ---. 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.
203. ---. The Apache HTTP Server Project. <<http://httpd.apache.org/>>, HTML, 2004.
204. ---. HttpClientPowered. <<http://wiki.apache.org/jakarta-httpclient/HttpClientPowered>>, HTML, 2005.
205. ---. HttpClient. <<http://jakarta.apache.org/commons/httpclient/>>, HTML, 2005.
206. ---. mod\_mbox. <[http://httpd.apache.org/mod\\_mbox/](http://httpd.apache.org/mod_mbox/)>, HTML, 2006.
207. Trachtenberg, A. PHP Web Services Without SOAP. <[http://www.onlamp.com/pub/a/php/2003/10/30/amazon\\_rest.html](http://www.onlamp.com/pub/a/php/2003/10/30/amazon_rest.html)>, October 30, 2003.
208. Trudelle, P. XPToolkit: Straight-up Elevator Story. <<http://www.mozilla.org/xpfe/ElevatorStraightUp.html>>, HTML, February 21, 1999.
209. Turner, D. and Oeschger, I. Creating XPCOM Components. 2003. <<http://www.mozilla.org/projects/xpcom/book/cxc/>>.
210. Udell, J. The Event-Driven Internet. Byte.com. December 3, 2001. <[http://www.byte.com/documents/s=1816/byt20011128s0003/1203\\_udell.html](http://www.byte.com/documents/s=1816/byt20011128s0003/1203_udell.html)>.
211. Vamosi, R. Security Watch: In defense of Mozilla Firefox. CNET Reviews. September 23, 2005. <[http://reviews.cnet.com/4520-3513\\_7-6333507-1.html](http://reviews.cnet.com/4520-3513_7-6333507-1.html)>.
212. Vatton, I. Amaya Home Page. <<http://www.w3.org/Amaya/>>, HTML, May 3, 2004.

213. Vogels, W. Eventually consistent. *Communications of the ACM*. 52(1), p. 40-44, January, 2009. <<http://doi.acm.org/10.1145/1435417.1435432>>.
214. Vyzovitis, D. and Lippman, A. MAST: A Dynamic Language for Programmable Networks. MIT Media Laboratory, Report, May, 2002.
215. W3C. Web of Services for Enterprise Computing. <<http://www.w3.org/2007/01/wos-ec-program.html>>, February 27-28, 2007.
216. Waldo, J., Wyant, G., Wollrath, A., and Kendall, S.C. A Note on Distributed Computing. Sun Microsystems, Report TR-94-29, November, 1994. <<http://research.sun.com/techrep/1994/abstract-29.html>>.
217. 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](http://blogs.msdn.com/david.wang/archive/2005/07/07/HOWTO_Use_Kernel_Response_Cache_with_IIS_6.aspx)>, HTML, July 7, 2005.
218. Wang, J. A survey of web caching schemes for the Internet. *ACM SIGCOMM Computer Communication Review*. 29(5), p. 36-46, October, 1999. <<http://doi.acm.org/10.1145/505696.505701>>.
219. Web Host Industry Review. Microsoft to Rewrite IIS, Release Patches. *Web Host Industry Review*. September 26, 2001. <<http://www.thewhir.com/marketwatch/iis926.cfm>>.
220. Whitehead, E.J. and Wiggins, M. WEBDAV: IETF Standard for Collaborative Authoring on the Web. *IEEE Internet Computing*. p. 34-40, September/October, 1998.
221. Wilson, B. Browser History: Netscape. <<http://www.bloobery.com/indexdot/history/netscape.htm>>, HTML, September 30, 2003.
222. Winer, D. XML-RPC Specification. <<http://www.xml-rpc.com/spec>>, June 15, 1999.
223. Wong, J. and Hong, J.I. Making Mashups with Marmite: Towards End-user Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors In Computing Systems*. p. 1435-1444, San Jose, California, USA, April 28-May 3, 2007. <<http://doi.acm.org/10.1145/1240624.1240842>>.
224. 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>>.
225. Yahoo, I. Flickr Services. <<http://www.flickr.com/services/api/>>, 2007.
226. Yahoo Inc. Pipes. <<http://pipes.yahoo.com/pipes/>>, 2009.
227. Young, M. AP News + Google Maps. <<http://www.81nassau.com/apnews/>>, HTML, 2006.
228. Zawinski, J. Grendel Overview. <<http://www.mozilla.org/projects/grendel/announce.html>>, HTML, September 8, 1998.
229. ---. java sucks. <<http://www.jwz.org/doc/java.html>>, HTML, 2000.

Appendix A. Architectural Characteristics of RESTful Systems

Name	NCSA HTTP Server	Apache HTTP Server	Apache HTTP Server 2.x	Internet Information Services	Internet Information Services 6.0	Mosaic	Navigator	Netcape 6.0	Netcape Mozilla 1.0	Firefox	Internet Explorer	Internet Explorer 7.0	Konqueror	Safari	libWWW	cURL	HTTPClient	Neon	Servlet	
Vendor Release	NCSA 12/1993	ASF 12/1995	ASF 4/2002	Microsoft 2/1996	Microsoft 3/2003	NCSA 5/1993	Netcape 12/1994	Netcape 11/2000	Netcape 11/2000	Mozilla 11/2004	Microsoft 8/1995	Microsoft 10/2005 [1]	KDE 10/2000	Apple 1/2003	W3C 10/1992	cURL Proj. 9/1998	ASF 9/2001	None 5/2000	None 10/2001	
REST Classification	User Agent Framework																			
<b>Portability: Indirect limitations and constraints upon the overall system architecture with respect to its environment</b>																				
Implementation Language	C	C	C	C++	C++	C	C, C++	C, C++	C, C++	C, C++	C++	C++	C++	C, C++	C	C	Java	C	C	
Operating Systems	Unix	Unix, OS/2, Windows, Netware+	Unix, OS/2, Windows, Netware+	Windows	Windows	Unix	Unix, Windows, Macintosh	Unix, Windows, Macintosh	Unix, Windows, Macintosh	Unix, Windows, Macintosh	Windows	Windows	Unix, Windows	Mac OS X	Unix, Windows, Macintosh	Unix, OS/2, Windows, Netware+	Any OS with a JVM	Unix, Windows	Unix, OS/2, Windows, Netware+	
OS-specific versions				a	a	a	a				a	a		a						
Window Interfaces		N/A	N/A			Monif	Xlib, MFC	GTK, MFC	GTK, MFC	GTK, MFC	MFC	MFC	QT	Cocoa			N/A			
Platform Portability	*	**	****	*	*	*	*	**	**	***	*	*	**	*	*	**	****	**	****	****
<b>Run-time Architecture: Direct limitations and constraints that the architecture represents with respect to the problem domain</b>																				
Parallelization	Process	Process	Hybrid	Threads	Threads	None	None	Threads	Threads	Threads	Threads	Threads	Threads	Threads	Event Loop	None	Threads	None	Threads	
Asynchronous			Partial													Partial				
HTTP Pipelining		a		a	a	a				✓ (Optional)	a	a	a	a	a				a	
Non-HTTP Protocols	**	**	**	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	
Security Risks	**	**	**	****	*	**	**	***	***	**	****	**	**	**	***	*	*	*	*	
<b>Internal Extensibility: Modifications through explicit introduction of architectural-level components</b>																				
Name	None	Hooks	Filters	ISAPI	ISAPI	CCI	Plugins	XP/COM	XP/COM	Web Accessories	Web Accessories	WebKit	WebKit	WebKit	Modules	Options	Objects	Handlers	Buckets	
Languages	None	C	C	C/C++	C/C++	C	C	C++	C++	HTML, C/C++, VB	HTML, C/C++, VB	Objective C	Objective C	Objective C	C	Over 30	Java	C	C	
Add New Protocols	*	Partial	a	a	a	*	**	a	a	a	a	a	a	a	a	a	a	a	a	
Modifiability	****	****	****	****	****	*	**	***	***	***	***	***	***	***	****	**	**	*	*	
Security Risks	****	****	****	****	****	****	***	***	***	****	****	****	****	****	****	****	*	****	****	
<b>External Extensibility: Changes that can be effected without introducing architectural-level components (Systems within the same classification often share the same external extensibility mechanisms)</b>																				
Notable Addition	CGI	PHP/SP/Perl/Python	ASP	ASP	ASP	Helper Apps	JavaScript	CSS	CSS	ActiveX	ActiveX									
Costs	*	*	*	*	*	*	**	***	***	***	***	***	***	***	*	*	*	*	*	
Accessibility	*	**	**	*	*	*	**	*	*	*	*	*	*	*	*	*	*	*	*	
Security Risks	****	****	****	****	****	*	**	*	*	****	****	****	****	****	****	****	****	****	****	
<b>Integration: Ability of an architecture to participate as part of a larger architecture</b>																				
App. Importability	Partial	Partial	a					a	a	a	a	a	a	a	a	a	a	a	a	
Language Control	Perl, Python	Perl, Python	Perl, Python	****	****	C++	C++	*	*	*	COM objects	Objective C	C++	Objective C	C	Over 30	Java	C	C	
<b>REST Constraints: Degrees of conformance to REST-imposed constraints</b>																				
Representation Meta-data Control	*	****	****	**	**	*	*	**	**	***	*	*	**	***	**	*	****	****	****	
Extensible Methods	a	a	a	a	a	*	**	a	a	a	*	*	Partial	a	a	a	a	a	a	
Resource/representation separation	*	**	**	**	**	*	**	***	***	***	*	*	*	*	*	*	*	*	*	
Internal transformation support	*	**	****	****	****	*	***	****	****	****	***	***	****	****	****	***	***	***	***	
Proxy Support	Partial	Partial	a			a	a	a	a	a	a	a	a	a	a	a	a	a	a	
Statefulness	****	****	****	**	**	****	*	*	*	*	*	*	*	*	*	*	*	*	*	
Cacheability	Partial	Partial	a	a	a	Partial	a	a	a	a	a	a	a	a	a	a	a	a	a	

\* = Low; \*\* = Moderate; \*\*\* = High; \*\*\*\* = Extreme

# Appendix B: Selected Feed Reader Widget Source Code

## Source Code: Manager Widget

```
(define (thunk/manager)
  (define (widget-list key value) (hashtable->alist value))

  (define (widget-spawn title)
    (cond ((string-ci=? "RSS Reader" title) (peer/spawn (this-peer) thunk/rss-feed))
          ((string-ci=? "Tag Cloud" title) (peer/spawn (this-peer) thunk/tagcloud))
          ((string-ci=? "QR Code" title) (peer/spawn (this-peer) thunk/qrcode))
          ;((string-ci=? "Mirror" title) (peer/spawn thunk/mirror))
          ((string-ci=? "URL Selector" title) (peer/spawn (this-peer) thunk/urlsel))
          ((string-ci=? "Sparkline" title) (peer/spawn (this-peer) thunk/sparkline))
          ((string-ci=? "Calendar" title) (peer/spawn (this-peer) thunk/calendar))
          ((string-ci=? "Google News" title) (peer/spawn (this-peer) thunk/google-news))
          (else ""))
    )
  )
  (let ((widgets-data (make-hashtable string-ci=?))
        (widgets-mbox (make-hashtable string-ci=?))
        (linkid 0))
    (display "thunk/manager: ready to serve\n")
    (let loop ((m (? (this-mailbox))))
      ;(display (format "thunk/manager: path:~a body:~a\n" (:message/path m) (:message/body
m)))
      (match
        (message/path+body m)

        (#(/http/get #(.origin ,uri ,request ,response))
          (let* (
              (req-uri (uri/path uri))
              (cond ((string-suffix-ci? "/maps" req-uri)
                     (http/response/entity! response (json/string (list (cons 'items (list->vector(hashtable/map widget-list widgets-data))))))
                    (else (display (format "Unknown URI: ~s" req-uri))))
                (! (:message/reply m) response :no-metadata: #f (:message/echo m))))

          (#(/http/post #(.origin ,uri ,req-body))
            (display (format "thunk/manager: body:~a\n" req-body))
            (let* (
                (req-uri (uri/path uri))
                (req-val (json/translate req-body))
                (cond ((string-suffix-ci? "/create" req-uri)
                      (let* ((table (make-hashtable string-ci=?))
                             (for-each (lambda (v) (hashtable/put! table (car v) (cdr v))) req-val)
                             (let* ((wid (hashtable/get table "id"))
                                    (title (hashtable/get table "title"))
                                    (wid-uuid (widget-spawn title))
                                    (wid-mbox (peer/mailbox (this-peer) wid-uuid))
                                    (type-url (format "/mailbox/~a" wid-uuid)))
                               (hashtable/put! table "url" type-url)
                               (hashtable/put! widgets-mbox wid wid-mbox)
                               (hashtable/put! widgets-data wid table))
                              ))
                    ((string-suffix-ci? "/link" req-uri)
                     (let* ((table (make-hashtable string-ci=?))
                            (for-each (lambda (v) (hashtable/put! table (car v) (cdr v))) req-val)
                            (hashtable/put! widgets-data (string-join (list "link" (number->string
linkid))) table)
                            (set! linkid (+ linkid 1))
                            (let* ((from-wid (hashtable/get table "from"))
```

```

        (to-wid (hashtable/get table "to"))
        (from-mbox (hashtable/get widgets-mbox from-wid))
        (to-mbox (hashtable/get widgets-mbox to-wid)))
      (! (@ from-mbox 'link/create) to-mbox)
    )
  ))
  ((string-suffix-ci? "/move" req-uri)
   (let* ((table (make-hashtable string-ci=?))
          (for-each (lambda (v) (hashtable/put! table (car v) (cdr v))) req-val)
          (let* ((wid (hashtable/get table "id"))
                 (cur-val (hashtable/get widgets-data wid)))
                (define (widget-list-update k v)
                  (hashtable/put! cur-val k v)
                )
                (hashtable/for-each widget-list-update table)
          ))
    )
    (else (display (format "Unknown URI: ~s" req-url))))
  ))
  (, _ignore #f))
(loop (? (this-mailbox))))
)

```

## Source Code: Tagcloud Widget

```
(define (thunk/tagcloud)

  (define (alist-filter list filter)
    (let loop ((pairs list) (outcome '()))
      (cond
        ((null? pairs) outcome)
        (else
         (if (filter (car pairs))
             (loop (cdr pairs) (cons (car pairs) outcome))
             (loop (cdr pairs) outcome))))))

  (define (topwc s wordlen count)
    (alist-filter s (lambda (x)
                      (and (>= (string-length (car x)) wordlen) (>= (cdr x) count)))))

  (define (generate-wordcount s l c)
    (let* ((hp (htmlparser/parse s))
           (ht (htmlparser/get-text hp))
           (wc (wordcount/list ht))
           )
      (topwc wc l c)
    )
  )

  (define (nc-map e) (list (cons 'name (car e)) (cons 'count (cdr e))))
  (let ((current-json #f)
        (current-text #f)
        (links '()))
    (let loop ((m (? (this-mailbox))))
      (match
       (message/path+body m)

        (#(link/create ,mailbox)
         (set! links (cons mailbox links))
         (! (@ mailbox 'link/data) current-text "text"))

        (#(link/data ,s)
         (set! current-text (generate-wordcount s 4 3))
         (set! current-json (json/string (list (cons 'items (list->vector (map nc-map current-text))))))
         (for-each (lambda (link) (! (@ link 'link/data) current-text "text")) links)
         )

        (#(/http/get #(.origin ,uri ,request ,response))
         (cond
          (current-json
           (http/response/entity! response current-json))
          (else
           (http/response/status! response 404)
           (http/response/entity! response "Not Found")))
         (! (:message/reply m) response :no-metadata: #f (:message/echo m)))
      )

    (loop (? (this-mailbox))))
  )
)
```