UNIVERSITY OF CALIFORNIA,
IRVINE

# Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

Dissertation Committee:
Professor Richard N. Taylor, Chair
Professor Mark S. Ackerman
Professor David S. Rosenblum

2000

The dissertation of Roy Thomas Fielding is approved
and is acceptable in quality and form
for publication on microfilm:

_____

_____

_____

Committee Chair

University of California, Irvine
2000

# DEDICATION

To
my parents,

Pete and Kathleen Fielding,

who made all of this possible,
for their endless encouragement and patience.

And also to

Tim Berners-Lee,

for making the World Wide Web an open, collaborative project.

*What is life?*
*It is the flash of a firefly in the night.*
*It is the breath of a buffalo in the wintertime.*
*It is the little shadow which runs across the grass*
*and loses itself in the sunset.*

— Crowfoot's last words (1890), Blackfoot warrior and orator.

*Almost everybody feels at peace with nature: listening to the ocean waves against the shore, by a still lake, in a field of grass, on a windblown heath. One day, when we have learned the timeless way again, we shall feel the same about our towns, and we shall feel as much at peace in them, as we do today walking by the ocean, or stretched out in the long grass of a meadow.*

— Christopher Alexander, The Timeless Way of Building (1979)

# TABLE OF CONTENTS

Page

# LIST OF FIGURES

# LIST OF TABLES

# ACKNOWLEDGMENTS

It has been a great pleasure working with the faculty, staff, and students at the University of California, Irvine, during my tenure as a doctoral student. This work would never have been possible if it were not for the freedom I was given to pursue my own research interests, thanks in large part to the kindness and considerable mentoring provided by Dick Taylor, my long-time advisor and committee chair. Mark Ackerman also deserves a great deal of thanks, for it was his class on distributed information services in 1993 that introduced me to the Web developer community and led to all of the design work described in this dissertation. Likewise, it was David Rosenblum's work on Internet-scale software architectures that convinced me to think of my own research in terms of architecture, rather than simply hypermedia or application-layer protocol design.

The Web's architectural style was developed iteratively over a six year period, but primarily during the first six months of 1995. It has been influenced by countless discussions with researchers at UCI, staff at the World Wide Web Consortium (W3C), and engineers within the HTTP and URI working groups of the Internet Engineering Taskforce (IETF). I would particularly like to thank Tim Berners-Lee, Henrik Frystyk Nielsen, Dan Connolly, Dave Raggett, Rohit Khare, Jim Whitehead, Larry Masinter, and Dan LaLiberte for many thoughtful conversations regarding the nature and goals of the WWW architecture. I'd also like to thank Ken Anderson for his insight into the open hypertext community and for trailblazing the path for hypermedia research at UCI. Thanks also to my fellow architecture researchers at UCI, all of whom finished before me, including Peyman Oreizy, Neno Medvidovic, Jason Robbins, and David Hilbert.

The Web architecture is based on the collaborative work of dozens of volunteer software developers, many of whom rarely receive the credit they deserve for pioneering the Web before it became a commercial phenomenon. In addition to the W3C folks above, recognition should go to the server developers that enabled much of the Web's rapid growth in 1993-1994 (more so, I believe, than did the browsers). That includes Rob McCool (NCSA httpd), Ari Luotonen (CERN httpd/proxy), and Tony Sanders (Plexus). Thanks also to "Mr. Content", Kevin Hughes, for being the first to implement most of the interesting ways to show information on the Web beyond hypertext. The early client developers also deserve thanks: Nicola Pellow (line-mode), Pei Wei (Viola), Tony Johnson (Midas), Lou Montulli (Lynx), Bill Perry (W3), and Marc Andreessen and Eric Bina (Mosaic for X). Finally, my personal thanks go to my libwww-perl collaborators, Oscar Nierstrasz, Martijn Koster, and Gisle Aas. Cheers!

The modern Web architecture is still defined more by the work of individual volunteers than by any single company. Chief among them are the members of the Apache Software Foundation. Special thanks go to Robert S. Thau for the incredibly robust Shambhala design that led to Apache 1.0, as well as for many discussions on desirable (and undesirable) Web extensions, to Dean Gaudet for teaching me more about detailed system performance evaluation than I thought I needed to know, and to Alexei Kosut for being the first to implement most of HTTP/1.1 in Apache. Additional thanks to the rest of the Apache Group founders, including Brian Behlendorf, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, and Andrew Wilson, for building a community that we can all be proud of and changing the world one more time.

I'd also like to thank all of the people at eBuilt who have made it such a great place to work. Particular thanks go to the four technical founders — Joe Lindsay, Phil Lindsay, Jim Hayes, and Joe Manna — for creating (and defending) a culture that makes engineering fun. Thanks also to Mike Dewey, Jeff Lenardson, Charlie Bunten, and Ted Lavoie, for making it possible to earn money while having fun. And special thanks to Linda Dailing, for being the glue that holds us all together.

Thanks and good luck go out to the team at Endeavors Technology, including Greg Bolcer, Clay Cover, Art Hitomi, and Peter Kammer. Finally, I'd like to thank my three muses—Laura, Nikki, and Ling—for their inspiration while writing this dissertation.

In large part, my dissertation research has been sponsored by the Defense Advanced Research Projects Agency, and Airforce Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Airforce Research Laboratory or the U.S. Government.

# CURRICULUM VITAE

## Roy Thomas Fielding

## Education

**Doctor of Philosophy** (2000)
University of California, Irvine
Information and Computer Science
Institute of Software Research
Advisor: Dr. Richard N. Taylor
Dissertation: *Architectural Styles and the Design of Network-based Software Architectures*

**Master of Science** (1993)
University of California, Irvine
Information and Computer Science
Major Emphasis: Software

**Bachelor of Science** (1988)
University of California, Irvine
Information and Computer Science

## Professional Experience

| | |
|---|---|
| 12/99 - | Chief Scientist, eBuilt, Inc., Irvine, California |
| 3/99 - | Chairman, The Apache Software Foundation |
| 4/92 - 12/99 | Graduate Student Researcher, Institute for Software Research<br>University of California, Irvine |
| 6/95 - 9/95 | Visiting Scholar, World Wide Web Consortium (W3C)<br>MIT Laboratory of Computer Science, Cambridge, Massachusetts |
| 9/91 - 3/92 | Teaching Assistant<br>ICS 121 - Introduction to Software Engineering<br>ICS 125A - Project in Software Engineering<br>University of California, Irvine |
| 11/89 - 6/91 | Software Engineer<br>ADC Kentrox, Inc., Portland, Oregon |
| 7/88 - 8/89 | Professional Staff (Software Engineer)<br>PRC Public Management Services, Inc., San Francisco, California |
| 10/86 - 6/88 | Programmer/Analyst<br>Megadyne Information Systems, Inc., Santa Ana, California |
| 6/84 - 9/86 | Programmer/Analyst<br>TRANSMAX, Inc., Santa Ana, California |

## Publications

### Refereed Journal Articles

[1] R. T. Fielding, E. J. Whitehead, Jr., K. M. Anderson, G. A. Bolcer, P. Oreizy, and R. N. Taylor. Web-based Development of Complex Information Products. *Communications of the ACM*, 41(8), August 1998, pp. 84-92.

[2] R. T. Fielding. Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web. *Computer Networks and ISDN Systems*, 27(2), November 1994, pp. 193-204. (Revision of [7] after special selection by referees.)

### Refereed Conference Publications

[3] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 407-416.

[4] A. Mockus, R. T. Fielding, and J. Herbsleb. A Case Study of Open Source Software Development: The Apache Server. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 263-272.

[5] E. J. Whitehead, Jr., R. T. Fielding, and K. M. Anderson. Fusing WWW and Link Server Technology: One Approach. In *Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext'96*, Washington, DC, March, 1996, pp. 81-86.

[6] M. S. Ackerman and R. T. Fielding. Collection Maintenance in the Digital Library. In *Proceedings of Digital Libraries '95*, Austin, Texas, June 1995, pp. 39-48.

[7] R. T. Fielding. Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web. In *Proceedings of the First International World Wide Web Conference*, Geneva, Switzerland, May 1994, pp. 147-156.

### Industry Standards

[8] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. *Internet Draft Standard RFC 2616*, June 1999. [Obsoletes *RFC 2068*, January 1997.]

[9] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. *Internet Draft Standard RFC 2396*, August 1998.

[10] J. Mogul, R. T. Fielding, J. Gettys, and H. F. Frystyk. Use and Interpretation of HTTP Version Numbers. *Internet Informational RFC 2145*, May 1997.

[11] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol — HTTP/1.0. *Internet Informational RFC 1945*, May 1996.

[12] R. T. Fielding. Relative Uniform Resource Locators. *Internet Proposed Standard RFC 1808*, June 1995.

### Industry Articles

[13] R. T. Fielding. The Secrets to Apache's Success. *Linux Magazine*, 1(2), June 1999, pp. 29-71.

[14] R. T. Fielding. Shared Leadership in the Apache Project. *Communications of the ACM*, 42(4), April 1999, pp. 42-43.

[15] R. T. Fielding and G. E. Kaiser. The Apache HTTP Server Project. *IEEE Internet Computing*, 1(4), July-August 1997, pp. 88-90.

### Non-Refereed Publications

[16] R. T. Fielding. Architectural Styles for Network-based Applications. Phase II Survey Paper, Department of Information and Computer Science, University of California, Irvine, July 1999.

[17] J. Grudin and R. T. Fielding. Working Group on Design Methods and Processes. In *Proceedings of the ICSE'94 Workshop on SE-HCI: Joint Research Issues*, Sorrento, Italy, May 1994. Published in "Software Engineering and Human-Computer Interaction," Springer-Verlag LNCS, vol. 896, 1995, pp. 4-8.

[18] R. T. Fielding. Conditional GET Proposal for HTTP Caching. *Published on the WWW*, January 1994.

### Published Software Packages

[19] *Apache httpd*. The Apache HTTP server is the world's most popular Web server software, used by more than 65% of all public Internet sites as of July 2000.

[20] *libwww-perl*. A library of Perl4 packages that provides a simple and consistent programming interface to the World Wide Web.

[21] *Onions*. A library of Ada95 packages that provides an efficient stackable streams capability for network and file system I/O.

[22] *MOMspider*. MOMspider is a web robot for providing multi-owner maintenance of distributed hypertext infostructures.

[23] *wwwstat*. A set of utilities for searching and summarizing WWW httpd server access logs and assisting other webmaster tasks.

## Formal Presentations

[1] *State of Apache*. O'Reilly Open Source Software Convention, Monterey, CA, July 2000.

[2] *Principled Design of the Modern Web Architecture*. 2000 International Conference on Software Engineering, Limerick, Ireland, June 2000.

[3] *HTTP and Apache*. ApacheCon 2000, Orlando, FL, March 2000.

[4] *Human Communication and the Design of the Modern Web Architecture*. WebNet World Conference on the WWW and the Internet (WebNet 99), Honolulu, HI, October 1999.

[5] *The Apache Software Foundation*. Computer & Communications Industry Association, Autumn Members Meeting, Dallas, TX, September 1999.

[6] *Uniform Resource Identifiers*. The Workshop on Internet-scale Technology (TWIST 99), Irvine, CA, August 1999.

[7] *Apache: Past, Present, and Future*. Web Design World, Seattle, WA, July 1999.

[8] *Progress Report on Apache*. ZD Open Source Forum, Austin, TX, June 1999.

[9] *Open Source, Apache-style: Lessons Learned from Collaborative Software Development*. Second Open Source and Community Licensing Summit, San Jose, CA, March 1999.

[10] *The Apache HTTP Server Project: Lessons Learned from Collaborative Software.* AT&T Labs — Research, Folsom Park, NJ, October 1998.

[11] *Collaborative Software Development: Joining the Apache Project*. ApacheCon '98, San Francisco, CA, October 1998.

[12] *Representational State Transfer: An Architectural Style for Distributed Hypermedia Interaction*. Microsoft Research, Redmond, WA, May 1998.

[13] *The Apache Group: A Case Study of Internet Collaboration and Virtual Communities*. UC Irvine Social Sciences WWW Seminar, Irvine, CA, May 1997.

[14] *WebSoft: Building a Global Software Engineering Environment*. Workshop on Software Engineering (on) the World Wide Web, 1997 International Conference on Software Engineering (ICSE 97), Boston, MA, May 1997.

[15] *Evolution of the Hypertext Transfer Protocol*. ICS Research Symposium, Irvine, CA, January 1997.

[16] *World Wide Web Infrastructure and Evolution*. IRUS SETT Symposium on WIRED: World Wide Web and the Internet, Irvine, CA, May 1996.

[17] *HTTP Caching*. Fifth International World Wide Web Conference (WWW5), Paris, France, May 1996.

[18] *The Importance of World Wide Web Infrastructure*. California Software Symposium (CSS '96), Los Angeles, CA, April 1996.

[19] *World Wide Web Software: An Insider's View.* IRUS Bay Area Roundtable (BART), Palo Alto, CA, January 1996.

[20] *libwww-Perl4 and libwww-Ada95*. Fourth International World Wide Web Conference, Boston, MA, December 1995.

[21] *Hypertext Transfer Protocol — HTTP/1.x*. Fourth International World Wide Web Conference, Boston, MA, December 1995.

[22] *Hypertext Transfer Protocol — HTTP/1.x*. HTTP Working Group, 34th Internet Engineering Taskforce Meeting, Dallas, TX, December 1995.

[23] *Hypertext Transfer Protocol — HTTP/1.0 and HTTP/1.1*. HTTP Working Group, 32nd Internet Engineering Taskforce Meeting, Danvers, MA, April 1995.

[24] *WWW Developer Starter Kits for Perl*. WebWorld Conference, Orlando, FL, January 1995, and Santa Clara, CA, April 1995.

[25] *Relative Uniform Resource Locators*. URI Working Group, 31st Internet Engineering Taskforce Meeting, San Jose, CA, December 1994.

[26] *Hypertext Transfer Protocol — HTTP/1.0*. HTTP BOF, 31st Internet Engineering Taskforce Meeting, San Jose, CA, December 1994.

[27] *Behind the Curtains: How the Web was/is/will be created*. UC Irvine Social Sciences World Wide Web Seminar, Irvine, CA, October 1995.

[28] *Maintaining Distributed Hypertext Infostructures: Welcome to MOMspider's Web.* First International World Wide Web Conference, Geneva, Switzerland, May 1994.

## Professional Activities

- Webmaster, 1997 International Conference on Software Engineering (ICSE'97), Boston, May 1997.
- HTTP Session Chair, Fifth International World Wide Web Conference (WWW5), Paris, France, May 1996.
- Birds-of-a-Feather Chair and Session Chair, Fourth International World Wide Web Conference (WWW4), Boston, December 1995.
- Student Volunteer, 17th International Conference on Software Engineering (ICSE 17), Seattle, June 1995.
- Student Volunteer, Second International World Wide Web Conference (WWW2), Chicago, October 1994.
- Student Volunteer, 16th International Conference on Software Engineering (ICSE 16), Sorrento, Italy, April 1994.
- Co-founder and member, The Apache Group, 1995-present.
- Founder and chief architect, libwww-perl collaborative project, 1994-95.
- ICS Representative, Associated Graduate Students Council, 1994-95.

## Professional Associations

- The Apache Software Foundation
- Association for Computing Machinery (ACM)
- ACM Special Interest Groups on Software Engineering (SIGSOFT), Data Communications (SIGCOMM), and Groupware (SIGGROUP)

## Honors, Awards, Fellowships

| | |
|---|---|
| 2000 | Appaloosa Award for Vision, O'Reilly Open Source 2000 |
| 2000 | Outstanding Graduate Student, UCI Alumni Association |
| 1999 | ACM Software System Award |
| 1999 | TR100: Top 100 young innovators, MIT Technology Review |
| 1991 | Regent's Fellowship, University of California |
| 1988 | Golden Key National Honor Society |
| 1987 | Dean's Honor List |

# ABSTRACT OF THE DISSERTATION

Architectural Styles and the Design of Network-based Software Architectures

by

Roy Thomas Fielding

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2000

Professor Richard N. Taylor, Chair

The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia system. The Web has been iteratively developed over the past ten years through a series of modifications to the standards that define its architecture. In order to identify those aspects of the Web that needed improvement and avoid undesirable modifications, a model for the modern Web architecture was needed to guide its design, definition, and deployment.

Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations. My work is motivated by the desire to understand and evaluate the architectural design of network-based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture. An architectural style is a named, coordinated set of architectural constraints.

This dissertation defines a framework for understanding software architecture via architectural styles and demonstrates how styles can be used to guide the architectural design of network-based application software. A survey of architectural styles for network-based applications is used to classify styles according to the architectural properties they induce on an architecture for distributed hypermedia. I then introduce the Representational State Transfer (REST) architectural style and describe how REST has been used to guide the design and development of the architecture for the modern Web.

REST emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. Finally, I describe the lessons learned from applying REST to the design of the Hypertext Transfer Protocol and Uniform Resource Identifier standards, and from their subsequent deployment in Web client and server software.

# INTRODUCTION

*Excuse me ... did you say 'knives'?*
— City Gent #1 (Michael Palin), The Architects Sketch [111]

As predicted by Perry and Wolf [105], software architecture has been a focal point for software engineering research in the 1990s. The complexity of modern software systems have necessitated a greater emphasis on componentized systems, where the implementation is partitioned into independent components that communicate to perform a desired task. Software architecture research investigates methods for determining how best to partition a system, how components identify and communicate with each other, how information is communicated, how elements of a system can evolve independently, and how all of the above can be described using formal and informal notations.

A good architecture is not created in a vacuum. All design decisions at the architectural level should be made within the context of the functional, behavioral, and social requirements of the system being designed, which is a principle that applies equally to both software architecture and the traditional field of building architecture. The guideline that "form follows function" comes from hundreds of years of experience with failed building projects, but is often ignored by software practitioners. The funny bit within the Monty Python sketch, cited above, is the absurd notion that an architect, when faced with the goal of designing an urban block of flats (apartments), would present a building design with all the components of a modern slaughterhouse. It might very well be the best slaughterhouse design ever conceived, but that would be of little comfort to the prospective tenants as they are whisked along hallways containing rotating knives.

The hyperbole of *The Architects Sketch* may seem ridiculous, but consider how often we see software projects begin with adoption of the latest fad in architectural design, and only later discover whether or not the system requirements call for such an architecture. Design-by-buzzword is a common occurrence. At least some of this behavior within the software industry is due to a lack of understanding of *why* a given set of architectural constraints is useful. In other words, the reasoning behind good software architectures is not apparent to designers when those architectures are selected for reuse.

This dissertation explores a junction on the frontiers of two research disciplines in computer science: software and networking. Software research has long been concerned with the categorization of software designs and the development of design methodologies, but has rarely been able to objectively evaluate the impact of various design choices on system behavior. Networking research, in contrast, is focused on the details of generic communication behavior between systems and improving the performance of particular communication techniques, often ignoring the fact that changing the interaction style of an application can have more impact on performance than the communication protocols used for that interaction. My work is motivated by the desire to understand and evaluate the architectural design of network-based application software through principled use of architectural constraints, thereby obtaining the functional, performance, and social properties desired of an architecture. When given a name, a coordinated set of architectural constraints becomes an architectural style.

The first three chapters of this dissertation define a framework for understanding software architecture via architectural styles, revealing how styles can be used to guide the architectural design of network-based application software. Common architectural styles

are surveyed and classified according to the architectural properties they induce when applied to an architecture for network-based hypermedia. This classification is used to identify a set of architectural constraints that could be used to improve the architecture of the early World Wide Web.

Architecting the Web requires an understanding of its requirements, as we shall discuss in Chapter 4. The Web is intended to be an *Internet-scale* distributed hypermedia system, which means considerably more than just geographical dispersion. The Internet is about interconnecting information networks across organizational boundaries. Suppliers of information services must be able to cope with the demands of anarchic scalability and the independent deployment of software components. Distributed hypermedia provides a uniform means of accessing services through the embedding of action controls within the presentation of information retrieved from remote sites. An architecture for the Web must therefore be designed with the context of communicating large-grain data objects across high-latency networks and multiple trust boundaries.

Chapter 5 introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles.

Over the past six years, the REST architectural style has been used to guide the design and development of the architecture for the modern Web, as presented in Chapter 6. This work was done in conjunction with my authoring of the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI), the two specifications that define the generic interface used by all component interactions on the Web.

Like most real-world systems, not all components of the deployed Web architecture obey every constraint present in its architectural design. REST has been used both as a means to define architectural improvements and to identify architectural mismatches. Mismatches occur when, due to ignorance or oversight, a software implementation is deployed that violates the architectural constraints. While mismatches cannot be avoided in general, it is possible to identify them before they become standardized. Several mismatches within the modern Web architecture are summarized in Chapter 6, along with analyses of why they arose and how they deviate from REST.

In summary, this dissertation makes the following contributions to software research within the field of Information and Computer Science:

- a framework for understanding software architecture through architectural styles, including a consistent set of terminology for describing software architecture;
- a classification of architectural styles for network-based application software by the architectural properties they would induce when applied to the architecture for a distributed hypermedia system;
- REST, a novel architectural style for distributed hypermedia systems; and,
- application and evaluation of the REST architectural style in the design and deployment of the architecture for the modern World Wide Web.

# CHAPTER 1
# Software Architecture

In spite of the interest in software architecture as a field of research, there is little agreement among researchers as to what exactly should be included in the definition of architecture. In many cases, this has led to important aspects of architectural design being overlooked by past research. This chapter defines a self-consistent terminology for software architecture based on an examination of existing definitions within the literature and my own insight with respect to network-based application architectures. Each definition, highlighted within a box for ease of reference, is followed by a discussion of how it is derived from, or compares to, related research.

## 1.1 Run-time Abstraction

> A **software architecture** is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.

At the heart of software architecture is the principle of abstraction: hiding some of the details of a system through encapsulation in order to better identify and sustain its properties [117]. A complex system will contain many levels of abstraction, each with its own architecture. An architecture represents an abstraction of system behavior at that level, such that architectural elements are delineated by the abstract interfaces they provide to other elements at that level [9]. Within each element may be found another architecture, defining the system of sub-elements that implement the behavior represented

by the parent element's abstract interface. This recursion of architectures continues down to the most basic system elements: those that cannot be decomposed into less abstract elements.

In addition to levels of architecture, a software system will often have multiple operational phases, such as start-up, initialization, normal processing, re-initialization, and shutdown. Each operational phase has its own architecture. For example, a configuration file will be treated as a data element during the start-up phase, but won't be considered an architectural element during normal processing, since at that point the information it contained will have already been distributed throughout the system. It may, in fact, have defined the normal processing architecture. An overall description of a system architecture must be capable of describing not only the operational behavior of the system's architecture during each phase, but also the architecture of transitions between phases.

Perry and Wolf [105] define processing elements as "transformers of data," while Shaw et al. [118] describe components as "the locus of computation and state." This is further clarified in Shaw and Clements [122]: "A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters." This raises an important distinction between software architecture and what is typically referred to as software structure: the former is an abstraction of the run-time behavior of a software system, whereas the latter is a property of the static software source code. Although there are advantages to having the modular structure of the source code match the decomposition of behavior within a running system, there are also advantages to having independent software components be implemented using parts of the same code (e.g., shared libraries). We separate the view of software architecture from that of the

source code in order to focus on the software's run-time characteristics independent of a given component's implementation. Therefore, architectural design and source code structural design, though closely related, are separate design activities. Unfortunately, some descriptions of software architecture fail to make this distinction (e.g., [9]).

## 1.2 Elements

> A **software architecture** is defined by a configuration of architectural elements—components, connectors, and data—constrained in their relationships in order to achieve a desired set of architectural properties.

A comprehensive examination of the scope and intellectual basis for software architecture can be found in Perry and Wolf [105]. They present a model that defines a software architecture as a set of architectural *elements* that have a particular *form*, explicated by a set of *rationale*. Architectural elements include processing, data, and connecting elements. Form is defined by the properties of the elements and the relationships among the elements — that is, the constraints on the elements. The rationale provides the underlying basis for the architecture by capturing the motivation for the choice of architectural style, the choice of elements, and the form.

My definitions for software architecture are an elaborated version of those within the Perry and Wolf [105] model, except that I exclude rationale. Although rationale is an important aspect of software architecture research and of architectural description in particular, including it within the definition of software architecture would imply that design documentation is part of the run-time system. The presence or absence of rationale can influence the evolution of an architecture, but, once constituted, the architecture is independent of its reasons for being. Reflective systems [80] can use the characteristics of

past performance to change future behavior, but in doing so they are replacing one lower-level architecture with another lower-level architecture, rather than encompassing rationale within those architectures.

As an illustration, consider what happens to a building if its blueprints and design plans are burned. Does the building immediately collapse? No, since the properties by which the walls sustain the weight of the roof remain intact. An architecture has, by design, a set of properties that allow it to meet or exceed the system requirements. Ignorance of those properties may lead to later changes which violate the architecture, just as the replacement of a load-bearing wall with a large window frame may violate the structural stability of a building. Thus, instead of rationale, our definition of software architecture includes architectural properties. Rationale explicates those properties, and lack of rationale may result in gradual decay or degradation of the architecture over time, but the rationale itself is not part of the architecture.

A key feature of the model in Perry and Wolf [105] is the distinction of the various element types. *Processing elements* are those that perform transformations on data, *data elements* are those that contain the information that is used and transformed, and *connecting elements* are the glue that holds the different pieces of the architecture together. I use the more prevalent terms of *components* and *connectors* to refer to processing and connecting elements, respectively.

Garlan and Shaw [53] describe an architecture of a system as a collection of computational components together with a description of the interactions between these components—the connectors. This model is expanded upon in Shaw et al. [118]: The architecture of a software system defines that system in terms of components and of

interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. Further elaboration of this definition can be found in Shaw and Garlan [121].

What is surprising about the Shaw et al. [118] model is that, rather than defining the software's architecture as existing within the software, it is defining a description of the software's architecture as if that were the architecture. In the process, software architecture as a whole is reduced to what is commonly found in most informal architecture diagrams: boxes (components) and lines (connectors). Data elements, along with many of the dynamic aspects of real software architectures, are ignored. Such a model is incapable of adequately describing network-based software architectures, since the nature, location, and movement of data elements within the system is often the single most significant determinant of system behavior.

### 1.2.1 Components

> A **component** is an abstract unit of software instructions and internal state that provides a transformation of data via its interface.

Components are the most easily recognized aspect of software architecture. Perry and Wolf's [105] processing elements are defined as those components that supply the transformation on the data elements. Garlan and Shaw [53] describe components simply as the elements that perform computation. Our definition attempts to be more precise in making the distinction between components and the software within connectors.

A component is an abstract unit of software instructions and internal state that provides a transformation of data via its interface. Example transformations include

loading into memory from secondary storage, performing some calculation, translating to a different format, encapsulation with other data, etc. The behavior of each component is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another component [9]. In other words, a component is defined by its interface and the services it provides to other components, rather than by its implementation behind the interface. Parnas [101] would define this as the set of assumptions that other architectural elements can make about the component.

### 1.2.2 Connectors

> A **connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components.

Perry and Wolf [105] describe connecting elements vaguely as the glue that holds the various pieces of the architecture together. A more precise definition is provided by Shaw and Clements [122]: A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components. Examples include shared representations, remote procedure calls, message-passing protocols, and data streams.

Perhaps the best way to think about connectors is to contrast them with components. Connectors enable communication between components by transferring data elements from one interface to another without changing the data. Internally, a connector may consist of a subsystem of components that transform the data for transfer, perform the transfer, and then reverse the transformation for delivery. However, the external behavioral abstraction captured by the architecture ignores those details. In contrast, a component may, but not always will, transform data from the external perspective.

### 1.2.3 Data

> A **datum** is an element of information that is transferred from a component, or received by a component, via a connector.

As noted above, the presence of data elements is the most significant distinction between the model of software architecture defined by Perry and Wolf [105] and the model used by much of the research labelled software architecture [1, 5, 9, 53, 56, 117-122, 128]. Boasson [24] criticizes current software architecture research for its emphasis on component structures and architecture development tools, suggesting that more focus should be placed on data-centric architectural modeling. Similar comments are made by Jackson [67].

A datum is an element of information that is transferred from a component, or received by a component, via a connector. Examples include byte-sequences, messages, marshalled parameters, and serialized objects, but do not include information that is permanently resident or hidden within a component. From the architectural perspective, a "file" is a transformation that a file system component might make from a "file name" datum received on its interface to a sequence of bytes recorded within an internally hidden storage system. Components can also generate data, as in the case of a software encapsulation of a clock or sensor.

The nature of the data elements within a network-based application architecture will often determine whether or not a given architectural style is appropriate. This is particularly evident in the comparison of mobile code design paradigms [50], where the choice must be made between interacting with a component directly or transforming the component into a data element, transferring it across a network, and then transforming it

back to a component that can be interacted with locally. It is impossible to evaluate such an architecture without considering data elements at the architectural level.

## 1.3 Configurations

> A **configuration** is the structure of architectural relationships among components, connectors, and data during a period of system run-time.

Abowd et al. [1] define architectural description as supporting the description of systems in terms of three basic syntactic classes: components, which are the locus of computation; connectors, which define the interactions between components; and configurations, which are collections of interacting components and connectors. Various style-specific concrete notations may be used to represent these visually, facilitate the description of legal computations and interactions, and constrain the set of desirable systems.

Strictly speaking, one might think of a configuration as being equivalent to a set of specific constraints on component interaction. For example, Perry and Wolf [105] include topology in their definition of architectural form relationships. However, separating the active topology from more general constraints allows an architect to more easily distinguish the active configuration from the potential domain of all legitimate configurations. Additional rationale for distinguishing configurations within architectural description languages is presented in Medvidovic and Taylor [86].

## 1.4 Properties

The set of architectural properties of a software architecture includes all properties that derive from the selection and arrangement of components, connectors, and data within the system. Examples include both the functional properties achieved by the system and non-

functional properties, such as relative ease of evolution, reusability of components, efficiency, and dynamic extensibility, often referred to as quality attributes [9].

Properties are induced by the set of constraints within an architecture. Constraints are often motivated by the application of a software engineering principle [58] to an aspect of the architectural elements. For example, the *uniform pipe-and-filter* style obtains the qualities of reusability of components and configurability of the application by applying generality to its component interfaces — constraining the components to a single interface type. Hence, the architectural constraint is "uniform component interface," motivated by the generality principle, in order to obtain two desirable qualities that will become the architectural properties of reusable and configurable components when that style is instantiated within an architecture.

The goal of architectural design is to create an architecture with a set of architectural properties that form a superset of the system requirements. The relative importance of the various architectural properties depends on the nature of the intended system. Section 2.3 examines the properties that are of particular interest to network-based application architectures.

## 1.5 Styles

> An **architectural style** is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.

Since an architecture embodies both functional and non-functional properties, it can be difficult to directly compare architectures for different types of systems, or for even the

same type of system set in different environments. Styles are a mechanism for categorizing architectures and for defining their common characteristics [38]. Each style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction by ignoring the incidental details of the rest of the architecture [117].

Perry and Wolf [105] define architectural style as an abstraction of element types and formal aspects from various specific architectures, perhaps concentrating on only certain aspects of an architecture. An architectural style encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships. This definition allows for styles that focus only on the connectors of an architecture, or on specific aspects of the component interfaces.

In contrast, Garlan and Shaw [53], Garlan et al. [56], and Shaw and Clements [122] all define style in terms of a pattern of interactions among typed components. Specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [53]. This restricted view of architectural styles is a direct result of their definition of software architecture — thinking of architecture as a formal description, rather than as a running system, leads to abstractions based only in the shared patterns of box and line diagrams. Abowd et al. [1] go further and define this explicitly as viewing the collection of conventions that are used to interpret a class of architectural descriptions as defining an architectural style.

New architectures can be defined as instances of specific styles [38]. Since architectural styles may address different aspects of software architecture, a given

architecture may be composed of multiple styles. Likewise, a hybrid style can be formed by combining multiple basic styles into a single coordinated style.

Some architectural styles are often portrayed as "silver bullet" solutions for all forms of software. However, a good designer should select a style that matches the needs of the particular problem being solved [119]. Choosing the right architectural style for a network-based application requires an understanding of the problem domain [67] and thereby the communication needs of the application, an awareness of the variety of architectural styles and the particular concerns they address, and the ability to anticipate the sensitivity of each interaction style to the characteristics of network-based communication [133].

Unfortunately, using the term style to refer to a coordinated set of constraints often leads to confusion. This usage differs substantially from the etymology of *style*, which would emphasize personalization of the design process. Loerke [76] devotes a chapter to denigrating the notion that personal stylistic concerns have any place in the work of a professional architect. Instead, he describes styles as the critics' view of past architecture, where the available choice of materials, the community culture, or the ego of the local ruler were responsible for the architectural style, not the designer. In other words, Loerke views the real source of style in traditional building architecture to be the set of constraints applied to the design, and attaining or copying a specific style should be the least of the designer's goals. Since referring to a named set of constraints as a style makes it easier to communicate the characteristics of common constraints, we use architectural styles as a method of abstraction, rather than as an indicator of personalized design.

## 1.6 Patterns and Pattern Languages

In parallel with the software engineering research in architectural styles, the object-oriented programming community has been exploring the use of design patterns and pattern languages to describe recurring abstractions in object-based software development. A design pattern is defined as an important and recurring system construct. A pattern language is a system of patterns organized in a structure that guides the patterns' application [70]. Both concepts are based on the writings of Alexander et al. [3, 4] with regard to building architecture.

The design space of patterns includes implementation concerns specific to the techniques of object-oriented programming, such as class inheritance and interface composition, as well as the higher-level design issues addressed by architectural styles [51]. In some cases, architectural style descriptions have been recast as architectural patterns [120]. However, a primary benefit of patterns is that they can describe relatively complex protocols of interactions between objects as a single abstraction [91], thus including both constraints on behavior and specifics of the implementation. In general, a pattern, or pattern language in the case of multiple integrated patterns, can be thought of as a recipe for implementing a desired set of interactions among objects. In other words, a pattern defines a process for solving a problem by following a path of design and implementation choices [34].

Like software architectural styles, the software patterns research has deviated somewhat from its origin in building architecture. Indeed, Alexander's notion of patterns centers not on recurring arrangements of architectural elements, but rather on the recurring pattern of events—human activity and emotion—that take place within a space, with the

understanding that a pattern of events cannot be separated from the space where it occurs [3]. Alexander's design philosophy is to identify patterns of life that are common to the target culture and determine what architectural constraints are needed to differentiate a given space such that it enables the desired patterns to occur naturally. Such patterns exist at multiple levels of abstraction and at all scales.

> *As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.*
>
> *As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.*
>
> *The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing. [3]*

In many ways, Alexander's patterns have more in common with software architectural styles than the design patterns of OOPL research. An architectural style, as a coordinated set of constraints, is applied to a design space in order to induce the architectural properties that are desired of the system. By applying a style, an architect is differentiating the software design space in the hope that the result will better match the forces inherent in the application, thus leading to system behavior that enhances the natural pattern rather than conflicting with it.

## 1.7 Views

> *An architectural viewpoint is often application-specific and varies widely based on the application domain. ... we have seen architectural viewpoints that address a variety of issues, including: temporal issues, state and control approaches, data representation, transaction life cycle, security safeguards, and peak demand and graceful degradation. No doubt there are many more possible viewpoints. [70]*

In addition to the many architectures within a system, and the many architectural styles from which the architectures are composed, it is also possible to view an architecture from many different perspectives. Perry and Wolf [105] describe three important views in software architecture: processing, data, and connection views. A process view emphasizes the data flow through the components and some aspects of the connections among the components with respect to the data. A data view emphasizes the processing flow, with less emphasis on the connectors. A connection view emphasizes the relationship between components and the state of communication.

Multiple architectural views are common within case studies of specific architectures [9]. One architectural design methodology, the 4+1 View Model [74], organizes the description of a software architecture using five concurrent views, each of which addresses a specific set of concerns.

## 1.8 Related Work

I include here only those areas of research that define software architecture or describe software architectural styles. Other areas for software architecture research include architectural analysis techniques, architecture recovery and re-engineering, tools and environments for architectural design, architecture refinement from specification to implementation, and case studies of deployed software architectures [55]. Related work in the areas of style classification, distributed process paradigms, and middleware are discussed in Chapter 3.

### 1.8.1 Design Methodologies

Most early research on software architecture was concentrated on design methodologies. For example, object-oriented design [25] advocates a way to structure problems that leads naturally to an object-based architecture (or, more accurately, does not lead naturally to any other form of architecture). One of the first design methodologies to emphasize design at the architectural level is Jackson System Development [30]. JSD intentionally structures the analysis of a problem so that it leads to a style of architecture that combines pipe-and-filter (data flow) and process control constraints. These design methodologies tend to produce only one style of architecture.

There has been some initial work investigating methodologies for the analysis and development of architectures. Kazman et al. have described design methods for eliciting the architectural aspects of a design through scenario-based analysis with SAAM [68] and architectural trade-off analysis via ATAM [69]. Shaw [119] compares a variety of box-and-arrow designs for an automobile cruise control system, each done using a different design methodology and encompassing several architectural styles.

### 1.8.2 Handbooks for Design, Design Patterns, and Pattern Languages

Shaw [117] advocates the development of architectural handbooks along the same lines as traditional engineering disciplines. The object-oriented programming community has taken the lead in producing catalogs of design patterns, as exemplified by the "Gang of Four" book [51] and the essays edited by Coplien and Schmidt [33].

Software design patterns tend to be more problem-oriented than architectural styles. Shaw [120] presents eight example architectural patterns based on the architectural styles

described in [53], including information on the kinds of problems best suited to each architecture. Buschmann et al. [28] provide a comprehensive examination of the architectural patterns common to object-based development. Both references are purely descriptive and make no attempt to compare or illustrate the differences among architectural patterns.

Tepfenhart and Cusick [129] use a two dimensional map to differentiate among domain taxonomies, domain models, architectural styles, frameworks, kits, design patterns, and applications. In the topology, design patterns are predefined design structures used as building blocks for a software architecture, whereas architectural styles are sets of operational characteristics that identify an architectural family independent of application domain. However, they fail to define architecture itself.

### 1.8.3 Reference Models and Domain-specific Software Architectures (DSSA)

Reference models are developed to provide conceptual frameworks for describing architectures and showing how components are related to each other [117]. The Object Management Architecture (OMA), developed by the OMG [96] as a reference model for brokered distributed object architectures, specifies how objects are defined and created, how client applications invoke objects, and how objects can be shared and reused. The emphasis is on management of distributed objects, rather than efficient application interaction.

Hayes-Roth et al. [62] define domain-specific software architecture (DSSA) as comprising: a) a reference architecture, which describes a general computational framework for a significant domain of applications, b) a component library, which

contains reusable chunks of domain expertise, and c) an application configuration method for selecting and configuring components within the architecture to meet particular application requirements. Tracz [130] provides a general overview of DSSA.

DSSA projects have been successful at transferring architectural decisions to running systems by restricting the software development space to a specific architectural style that matches the domain requirements [88]. Examples include ADAGE [10] for avionics, AIS [62] for adaptive intelligent systems, and MetaH [132] for missile guidance, navigation, and control systems. DSSA emphasize reuse of components within a common architectural domain, rather than selecting an architectural style that is specific to each system.

### 1.8.4 Architecture Description Languages (ADL)

Most of the recent published work regarding software architectures is in the area of architecture description languages (ADL). An ADL is, according to Medvidovic and Taylor [86], a language that provides features for the explicit specification and modeling of a software system's conceptual architecture, including at a minimum: components, component interfaces, connectors, and architectural configurations.

Darwin is a declarative language which is intended to be a general purpose notation for specifying the structure of systems composed of diverse components using diverse interaction mechanisms [81]. Darwin's interesting qualities are that it allows the specification of distributed architectures and dynamically composed architectures [82].

UniCon [118] is a language and associated toolset for composing an architecture from a restricted set of component and connector examples. Wright [5] provides a formal basis

for specifying the interactions between architectural components by specifying connector types by their interaction protocols.

Like design methodologies, ADLs often introduce specific architectural assumptions that may impact their ability to describe some architectural styles, and may conflict with the assumptions in existing middleware [38]. In some cases, an ADL is designed specifically for a single architectural style, thus improving its capacity for specialized description and analysis at the cost of generality. For example, C2SADEL [88] is an ADL designed specifically to describe architectures developed in the C2 style [128]. In contrast, ACME [57] is an ADL that attempts to be as generic as possible, but with the trade-off being that it doesn't support style-specific analysis and the building of actual applications; rather, its focus is on the interchange among analysis tools.

### 1.8.5 Formal Architectural Models

Abowd et al. [1] claim that architectural styles can be described formally in terms of a small set of mappings from the syntactic domain of architectural descriptions (box-and-line diagrams) to the semantic domain of architectural meaning. However, this assumes that the architecture is the description, rather than an abstraction of a running system.

Inverardi and Wolf [65] use the Chemical Abstract Machine (CHAM) formalism to model software architecture elements as chemicals whose reactions are controlled by explicitly stated rules. It specifies the behavior of components according to how they transform available data elements and uses composition rules to propagate the individual transformations into an overall system result. While this is an interesting model, it is

unclear as to how CHAM could be used to describe any form of architecture whose purpose goes beyond transforming a data stream.

Rapide [78] is a concurrent, event-based simulation language specifically designed for defining and simulating system architectures. The simulator produces a partially-ordered set of events that can be analyzed for conformance to the architectural constraints on interconnection. Le Métayer [75] presents a formalism for the definition of architectures in terms of graphs and graph grammars.

## 1.9 Summary

This chapter examined the background for this dissertation. Introducing and formalizing a consistent set of terminology for software architecture concepts is necessary to avoid the confusion between architecture and architecture description that is common in the literature, particularly since much of the prior research on architecture excludes data as an important architectural element. I concluded with a survey of other research related to software architecture and architectural styles.

The next two chapters continue our discussion of background material by focusing on network-based application architectures and describing how styles can be used to guide their architectural design, followed by a survey of common architectural styles using a classification methodology that highlights the architectural properties induced when the styles are applied to an architecture for network-based hypermedia.

# CHAPTER 2
# Network-based Application Architectures

This chapter continues our discussion of background material by focusing on network-based application architectures and describing how styles can be used to guide their architectural design.

## 2.1 Scope

Architecture is found at multiple levels within software systems. This dissertation examines the highest level of abstraction in software architecture, where the interactions among components are capable of being realized in network communication. We limit our discussion to styles for network-based application architectures in order to reduce the dimensions of variance among the styles studied.

### 2.1.1 Network-based vs. Distributed

The primary distinction between network-based architectures and software architectures in general is that communication between components is restricted to message passing [6], or the equivalent of message passing if a more efficient mechanism can be selected at run-time based on the location of components [128].

Tanenbaum and van Renesse [127] make a distinction between distributed systems and network-based systems: a distributed system is one that looks to its users like an ordinary centralized system, but runs on multiple, independent CPUs. In contrast, network-based systems are those capable of operation across a network, but not

necessarily in a fashion that is transparent to the user. In some cases it is desirable for the user to be aware of the difference between an action that requires a network request and one that is satisfiable on their local system, particularly when network usage implies an extra transaction cost [133]. This dissertation covers network-based systems by not limiting the candidate styles to those that preserve transparency for the user.

### 2.1.2 Application Software vs. Networking Software

Another restriction on the scope of this dissertation is that we limit our discussion to application architectures, excluding the operating system, networking software, and some architectural styles that would only use a network for system support (e.g., process control styles [53]). Applications represent the "business-aware" functionality of a system [131].

Application software architecture is an abstraction level of an overall system, in which the goals of a user action are representable as functional architectural properties. For example, a hypermedia application must be concerned with the location of information pages, performing requests, and rendering data streams. This is in contrast to a networking abstraction, where the goal is to move bits from one location to another without regard to why those bits are being moved. It is only at the application level that we can evaluate design trade-offs based on the number of interactions per user action, the location of application state, the effective throughput of all data streams (as opposed to the potential throughput of a single data stream), the extent of communication being performed per user action, etc.

## 2.2 Evaluating the Design of Application Architectures

One of the goals of this dissertation is to provide design guidance for the task of selecting or creating the most appropriate architecture for a given application domain, keeping in mind that an architecture is the realization of an architectural design and not the design itself. An architecture can be evaluated by its run-time characteristics, but we would obviously prefer an evaluation mechanism that could be applied to the candidate architectural designs before having to implement all of them. Unfortunately, architectural designs are notoriously hard to evaluate and compare in an objective manner. Like most artifacts of creative design, architectures are normally presented as a completed work, as if the design simply sprung fully-formed from the architect's mind. In order to evaluate an architectural design, we need to examine the design rationale behind the constraints it places on a system, and compare the properties derived from those constraints to the target application's objectives.

The first level of evaluation is set by the application's functional requirements. For example, it makes no sense to evaluate the design of a process control architecture against the requirements of a distributed hypermedia system, since the comparison is moot if the architecture would not function. Although this will eliminate some candidates, in most cases there will remain many other architectural designs that are capable of meeting the application's functional needs. The remainder differ by their relative emphasis on the non-functional requirements—the degree to which each architecture would support the various non-functional architectural properties that have been identified as necessary for the system. Since properties are created by the application of architectural constraints, it is possible to evaluate and compare different architectural designs by identifying the

constraints within each architecture, evaluating the set of properties induced by each constraint, and comparing the cumulative properties of the design to those properties required of the application.

As described in the previous chapter, an architectural style is a coordinated set of architectural constraints that has been given a name for ease of reference. Each architectural design decision can be seen as an application of a style. Since the addition of a constraint may derive a new style, we can think of the space of all possible architectural styles as a derivation tree, with its root being the null style (empty set of constraints). When their constraints do not conflict, styles can be combined to form hybrid styles, eventually culminating in a hybrid style that represents a complete abstraction of the architectural design. An architectural design can therefore be analyzed by breaking-down its set of constraints into a derivation tree and evaluating the cumulative effect of the constraints represented by that tree. If we understand the properties induced by each basic style, then traversing the derivation tree gives us an understanding of the overall design's architectural properties. The specific needs of an application can then be matched against the properties of the design. Comparison becomes a relatively simple matter of identifying which architectural design satisfies the most desired properties for that application.

Care must be taken to recognize when the effects of one constraint may counteract the benefits of some other constraint. Nevertheless, it is possible for an experienced software architect to build such a derivation tree of architectural constraints for a given application domain, and then use the tree to evaluate many different architectural designs for applications within that domain. Thus, building a derivation tree provides a mechanism for architectural design guidance.

The evaluation of architectural properties within a tree of styles is specific to the needs of a particular application domain because the impact of a given constraint is often dependent on the application characteristics. For example, the pipe-and-filter style enables several positive architectural properties when used within a system that requires data transformations between components, whereas it would add nothing but overhead to a system that consists only of control messages. Since it is rarely useful to compare architectural designs across different application domains, the simplest means of ensuring consistency is to make the tree domain-specific.

Design evaluation is frequently a question of choosing between trade-offs. Perry and Wolf [105] describe a method of recognizing trade-offs explicitly by placing a numeric weight against each property to indicate its relative importance to the architecture, thus providing a normalized metric for comparing candidate designs. However, in order to be a meaningful metric, each weight would have to be carefully chosen using an objective scale that is consistent across all properties. In practice, no such scale exists. Rather than having the architect fiddle with weight values until the result matches their intuition, I prefer to present all of the information to the architect in a readily viewable form, and let the architect's intuition be guided by the visual pattern. This will be demonstrated in the next chapter.

## 2.3 Architectural Properties of Key Interest

This section describes the architectural properties used to differentiate and classify architectural styles in this dissertation. It is not intended to be a comprehensive list. I have included only those properties that are clearly influenced by the restricted set of styles

surveyed. Additional properties, sometimes referred to as software qualities, are covered by most textbooks on software engineering (e.g., [58]). Bass et al. [9] examine qualities in regards to software architecture.

### 2.3.1 Performance

One of the main reasons to focus on styles for network-based applications is because component interactions can be the dominant factor in determining user-perceived performance and network efficiency. Since the architectural style influences the nature of those interactions, selection of an appropriate architectural style can make the difference between success and failure in the deployment of a network-based application.

The performance of a network-based application is bound first by the application requirements, then by the chosen interaction style, followed by the realized architecture, and finally by the implementation of each component. In other words, software cannot avoid the basic cost of achieving the application needs; e.g., if the application requires that data be located on system A and processed on system B, then the software cannot avoid moving that data from A to B. Likewise, an architecture cannot be any more efficient than its interaction style allows; e.g., the cost of multiple interactions to move the data from A to B cannot be any less than that of a single interaction from A to B. Finally, regardless of the quality of an architecture, no interaction can take place faster than a component implementation can produce data and its recipient can consume data.

#### *2.3.1.1 Network Performance*

Network performance measures are used to describe some attributes of communication. *Throughput* is the rate at which information, including both application data and

communication overhead, is transferred between components. *Overhead* can be separated into initial setup overhead and per-interaction overhead, a distinction which is useful for identifying connectors that can share setup overhead across multiple interactions (*amortization*). *Bandwidth* is a measure of the maximum available throughput over a given network link. *Usable bandwidth* refers to that portion of bandwidth which is actually available to the application.

Styles impact network performance by their influence on the number of interactions per user action and the granularity of data elements. A style that encourages small, strongly typed interactions will be efficient in an application involving small data transfers among known components, but will cause excessive overhead within applications that involve large data transfers or negotiated interfaces. Likewise, a style that involves the coordination of multiple components arranged to filter a large data stream will be out of place in an application that primarily requires small control messages.

#### *2.3.1.2 User-perceived Performance*

User-perceived performance differs from network performance in that the performance of an action is measured in terms of its impact on the user in front of an application rather than the rate at which the network moves information. The primary measures for user-perceived performance are latency and completion time.

*Latency* is the time period between initial stimulus and the first indication of a response. Latency occurs at several points in the processing of a network-based application action: 1) the time needed for the application to recognize the event that initiated the action; 2) the time required to setup the interactions between components; 3) the time required to transmit each interaction to the components; 4) the time required to

process each interaction on those components; and, 5) the time required to complete sufficient transfer and processing of the result of the interactions before the application is able to begin rendering a usable result. It is important to note that, although only (3) and (5) represent actual network communication, all five points can be impacted by the architectural style. Furthermore, multiple component interactions per user action are additive to latency unless they take place in parallel.

*Completion* is the amount of time taken to complete an application action. Completion time is dependent upon all of the aforementioned measures. The difference between an action's completion time and its latency represents the degree to which the application is incrementally processing the data being received. For example, a Web browser that can render a large image while it is being received provides significantly better user-perceived performance than one that waits until the entire image is completely received prior to rendering, even though both experience the same network performance.

It is important to note that design considerations for optimizing latency will often have the side-effect of degrading completion time, and vice versa. For example, compression of a data stream can produce a more efficient encoding if the algorithm samples a significant portion of the data before producing the encoded transformation, resulting in a shorter completion time to transfer the encoded data across the network. However, if this compression is being performed on-the-fly in response to a user action, then buffering a large sample before transfer may produce an unacceptable latency. Balancing these trade-offs can be difficult, particularly when it is unknown whether the recipient cares more about latency (e.g., Web browsers) or completion (e.g., Web spiders).

#### *2.3.1.3 Network Efficiency*

An interesting observation about network-based applications is that the best application performance is obtained by not using the network. This essentially means that the most efficient architectural styles for a network-based application are those that can effectively minimize use of the network when it is possible to do so, through reuse of prior interactions (caching), reduction of the frequency of network interactions in relation to user actions (replicated data and disconnected operation), or by removing the need for some interactions by moving the processing of data closer to the source of the data (mobile code).

The impact of the various performance issues is often related to the scope of distribution for the application. The benefits of a style under local conditions may become drawbacks when faced with global conditions. Thus, the properties of a style must be framed in relation to the interaction distance: within a single process, across processes on a single host, inside a local-area network (LAN), or spread across a wide-area network (WAN). Additional concerns become evident when interactions across a WAN, where a single organization is involved, are compared to interactions across the Internet, involving multiple trust boundaries.

### 2.3.2 Scalability

Scalability refers to the ability of the architecture to support large numbers of components, or interactions among components, within an active configuration. Scalability can be improved by simplifying components, by distributing services across many components (decentralizing the interactions), and by controlling interactions and configurations as a

result of monitoring. Styles influence these factors by determining the location of application state, the extent of distribution, and the coupling between components.

Scalability is also impacted by the frequency of interactions, whether the load on a component is distributed evenly over time or occurs in peaks, whether an interaction requires guaranteed delivery or a best-effort, whether a request involves synchronous or asynchronous handling, and whether the environment is controlled or anarchic (i.e., can you trust the other components?).

### 2.3.3 Simplicity

The primary means by which architectural styles induce simplicity is by applying the principle of separation of concerns to the allocation of functionality within components. If functionality can be allocated such that the individual components are substantially less complex, then they will be easier to understand and implement. Likewise, such separation eases the task of reasoning about the overall architecture. I have chosen to lump the qualities of complexity, understandability, and verifiability under the general property of simplicity, since they go hand-in-hand for a network-based system.

Applying the principle of generality to architectural elements also improves simplicity, since it decreases variation within an architecture. Generality of connectors leads to middleware [22].

### 2.3.4 Modifiability

Modifiability is about the ease with which a change can be made to an application architecture. Modifiability can be further broken down into evolvability, extensibility, customizability, configurability, and reusability, as described below. A particular concern

of network-based systems is dynamic modifiability [98], where the modification is made to a deployed application without stopping and restarting the entire system.

Even if it were possible to build a software system that perfectly matches the requirements of its users, those requirements will change over time just as society changes over time. Because the components participating in a network-based application may be distributed across multiple organizational boundaries, the system must be prepared for gradual and fragmented change, where old and new implementations coexist, without preventing the new implementations from making use of their extended capabilities.

#### *2.3.4.1 Evolvability*

Evolvability represents the degree to which a component implementation can be changed without negatively impacting other components. Static evolution of components generally depends on how well the architectural abstraction is enforced by the implementation, and thus is not something unique to any particular architectural style. Dynamic evolution, however, can be influenced by the style if it includes constraints on the maintenance and location of application state. The same techniques used to recover from partial failure conditions in a distributed system [133] can be used to support dynamic evolution.

#### *2.3.4.2 Extensibility*

Extensibility is defined as the ability to add functionality to a system [108]. Dynamic extensibility implies that functionality can be added to a deployed system without impacting the rest of the system. Extensibility is induced within an architectural style by reducing the coupling between components, as exemplified by event-based integration.

#### *2.3.4.3 Customizability*

Customizability refers to the ability to temporarily specialize the behavior of an architectural element, such that it can then perform an unusual service. A component is customizable if it can be extended by one client of that component's services without adversely impacting other clients of that component [50]. Styles that support customization may also improve simplicity and scalability, since service components can be reduced in size and complexity by directly implementing only the most frequent services and allowing infrequent services to be defined by the client. Customizability is a property induced by the remote evaluation and code-on-demand styles.

#### *2.3.4.4 Configurability*

Configurability is related to both extensibility and reusability in that it refers to post-deployment modification of components, or configurations of components, such that they are capable of using a new service or data element type. The pipe-and-filter and code-on-demand styles are two examples that induce configurability of configurations and components, respectively.

#### *2.3.4.5 Reusability*

Reusability is a property of an application architecture if its components, connectors, or data elements can be reused, without modification, in other applications. The primary mechanisms for inducing reusability within architectural styles is reduction of coupling (knowledge of identity) between components and constraining the generality of component interfaces. The uniform pipe-and-filter style exemplifies these types of constraints.

### 2.3.5 Visibility

Styles can also influence the visibility of interactions within a network-based application by restricting interfaces via generality or providing access to monitoring. Visibility in this case refers to the ability of a component to monitor or mediate the interaction between two other components. Visibility can enable improved performance via shared caching of interactions, scalability through layered services, reliability through reflective monitoring, and security by allowing the interactions to be inspected by mediators (e.g., network firewalls). The mobile agent style is an example where the lack of visibility may lead to security concerns.

This usage of the term visibility differs from that in Ghezzi et al. [58], where they are referring to visibility into the development process rather than the product.

### 2.3.6 Portability

Software is portable if it can run in different environments [58]. Styles that induce portability include those that move code along with the data to be processed, such as the virtual machine and mobile agent styles, and those that constrain the data elements to a set of standardized formats.

### 2.3.7 Reliability

Reliability, within the perspective of application architectures, can be viewed as the degree to which an architecture is susceptible to failure at the system level in the presence of partial failures within components, connectors, or data. Styles can improve reliability by avoiding single points of failure, enabling redundancy, allowing monitoring, or reducing the scope of failure to a recoverable action.

## 2.4 Summary

This chapter examined the scope of the dissertation by focusing on network-based application architectures and describing how styles can be used to guide their architectural design. It also defined the set of architectural properties that will be used throughout the rest of the dissertation for the comparison and evaluation of architectural styles.

The next chapter presents a survey of common architectural styles for network-based application software within a classification framework that evaluates each style according to the architectural properties it would induce if applied to an architecture for a prototypical network-based hypermedia system.

# CHAPTER 3
# Network-based Architectural Styles

This chapter presents a survey of common architectural styles for network-based application software within a classification framework that evaluates each style according to the architectural properties it would induce if applied to an architecture for a prototypical network-based hypermedia system.

## 3.1 Classification Methodology

The purpose of building software is not to create a specific topology of interactions or use a particular component type — it is to create a system that meets or exceeds the application needs. The architectural styles chosen for a system's design must conform to those needs, not the other way around. Therefore, in order to provide useful design guidance, a classification of architectural styles should be based on the architectural properties induced by those styles.

### 3.1.1 Selection of Architectural Styles for Classification

The set of architectural styles included in the classification is by no means comprehensive of all possible network-based application styles. Indeed, a new style can be formed merely by adding an architectural constraint to any one of the styles surveyed. My goal is to describe a representative sample of styles, particularly those already identified within the software architecture literature, and provide a framework by which other styles can be added to the classification as they are developed.

I have intentionally excluded styles that do not enhance the communication or interaction properties when combined with one of the surveyed styles to form a network-based application. For example, the blackboard architectural style [95] consists of a central repository and a set of components (knowledge sources) that operate opportunistically upon the repository. A blackboard architecture can be extended to a network-based system by distributing the components, but the properties of such an extension are entirely based on the interaction style chosen to support the distribution — notifications via event-based integration, polling *a la* client-server, or replication of the repository. Thus, there would be no added value from including it in the classification, even though the hybrid style is network-capable.

### 3.1.2 Style-induced Architectural Properties

My classification uses relative changes in the architectural properties induced by each style as a means of illustrating the effect of each architectural style when applied to a system for distributed hypermedia. Note that the evaluation of a style for a given property depends on the type of system interaction being studied, as described in Section 2.2. The architectural properties are relative in the sense that adding an architectural constraint may improve or reduce a given property, or simultaneously improve one aspect of the property and reduce some other aspect of the property. Likewise, improving one property may lead to the reduction of another.

Although our discussion of architectural styles will include those applicable to a wide range of network-based systems, our evaluation of each style will be based on its impact upon an architecture for a single type of software: network-based hypermedia systems.

Focusing on a particular type of software allows us to identify the advantages of one style over another in the same way that a designer of a system would evaluate those advantages. Since we do not intend to declare any single style as being universally desirable for all types of software, restricting the focus of our evaluation simply reduces the dimensions over which we need to evaluate. Evaluating the same styles for other types of application software is an open area for future research.

### 3.1.3 Visualization

I use a table of style versus architectural properties as the primary visualization for this classification. The table values indicate the relative influence that the style for a given row has on a column's property. Minus (–) symbols accumulate for negative influences and plus (+) symbols for positive, with plus-minus (±) indicating that it depends on some aspect of the problem domain. Although this is a gross simplification of the details presented in each section, it does indicate the degree to which a style has addressed (or ignored) an architectural property.

An alternative visualization would be a property-based derivation graph for classifying architectural styles. The styles would be classified according to how they are derived from other styles, with the arcs between styles illustrated by architectural properties gained or lost. The starting point of the graph would be the null style (no constraints). It is possible to derive such a graph directly from the descriptions.

## 3.2 Data-flow Styles

| Style | Derivation | Net Perform. | UP Perform. | Efficiency | Scalability | Simplicity | Evolvability | Extensibility | Customiz. | Configur. | Reusability | Visibility | Portability | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PF | | | ± | | | + | + | + | | + | + | | | |
| UPF | PF | – | ± | | | ++ | + | + | | ++ | ++ | + | | |

Table 3-1. Evaluation of Data-flow Styles for Network-based Hypermedia

### 3.2.1 Pipe and Filter (PF)

In a pipe and filter style, each component (filter) reads streams of data on its inputs and produces streams of data on its outputs, usually while applying a transformation to the input streams and processing them incrementally so that output begins before the input is completely consumed [53]. This style is also referred to as a one-way data flow network [6]. The constraint is that a filter must be completely independent of other filters (zero coupling): it must not share state, control thread, or identity with the other filters on its upstream and downstream interfaces [53].

Abowd et al. [1] provide an extensive formal description of the pipe and filter style using the Z language. The *Khoros* software development environment for image processing [112] provides a good example of using the pipe and filter style to build a range of applications.

Garlan and Shaw [53] describe the advantageous properties of the pipe and filter style as follows. First, PF allows the designer to understand the overall input/output of the system as a simple composition of the behaviors of the individual filters (simplicity). Second, PF supports reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them (reusability). Third, PF systems can be easily maintained and enhanced: new filters can be added to existing systems

(extensibility) and old filters can be replaced by improved ones (evolvability). Fourth, they permit certain kinds of specialized analysis (verifiability), such as throughput and deadlock analysis. Finally, they naturally support concurrent execution (user-perceived performance).

Disadvantages of the PF style include: propagation delay is added through long pipelines, batch sequential processing occurs if a filter cannot incrementally process its inputs, and no interactivity is allowed. A filter cannot interact with its environment because it cannot know that any particular output stream shares a controller with any particular input stream. These properties decrease user-perceived performance if the problem being addressed does not fit the pattern of a data flow stream.

One aspect of PF styles that is rarely mentioned is that there is an implied “invisible hand” that arranges the configuration of filters in order to establish the overall application. A network of filters is typically arranged just prior to each activation, allowing the application to specify the configuration of filter components based on the task at hand and the nature of the data streams (configurability). This controller function is considered a separate operational phase of the system, and hence a separate architecture, even though one cannot exist without the other.

### 3.2.2 Uniform Pipe and Filter (UPF)

The uniform pipe and filter style adds the constraint that all filters must have the same interface. The primary example of this style is found in the Unix operating system, where filter processes have an interface consisting of one input data stream of characters (stdin) and two output data streams of characters (stdout and stderr). Restricting the interface

allows independently developed filters to be arranged at will to form new applications. It also simplifies the task of understanding how a given filter works.

A disadvantage of the uniform interface is that it may reduce network performance if the data needs to be converted to or from its natural format.

## 3.3 Replication Styles

| Style | Derivation | Net Perform. | UP Perform. | Efficiency | Scalability | Simplicity | Evolvability | Extensibility | Customiz. | Configur. | Reusability | Visibility | Portability | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RR | | | ++ | | + | | | | | | | | | + |
| $ | RR | | + | + | + | + | | | | | | | | |

Table 3-2. Evaluation of Replication Styles for Network-based Hypermedia

### 3.3.1 Replicated Repository (RR)

Systems based on the replicated repository style [6] improve the accessibility of data and scalability of services by having more than one process provide the same service. These decentralized servers interact to provide clients the illusion that there is just one, centralized service. Distributed filesystems, such as XMS [49], and remote versioning systems, like CVS [www.cyclic.com], are the primary examples.

Improved user-perceived performance is the primary advantage, both by reducing the latency of normal requests and enabling disconnected operation in the face of primary server failure or intentional roaming off the network. Simplicity remains neutral, since the complexity of replication is offset by the savings of allowing network-unaware components to operate transparently on locally replicated data. Maintaining consistency is the primary concern.

### 3.3.2 Cache ($)

A variant of replicated repository is found in the cache style: replication of the result of an individual request such that it may be reused by later requests. This form of replication is most often found in cases where the potential data set far exceeds the capacity of any one client, as in the WWW [20], or where complete access to the repository is unnecessary. Lazy replication occurs when data is replicated upon a not-yet-cached response for a request, relying on locality of reference and commonality of interest to propagate useful items into the cache for later reuse. Active replication can be performed by pre-fetching cacheable entries based on anticipated requests.

Caching provides slightly less improvement than the replicated repository style in terms of user-perceived performance, since more requests will miss the cache and only recently accessed data will be available for disconnected operation. On the other hand, caching is much easier to implement, doesn't require as much processing and storage, and is more efficient because data is transmitted only when it is requested. The cache style becomes network-based when it is combined with a client-stateless-server style.

## 3.4 Hierarchical Styles

| Style | Derivation | Net Perform. | UP Perform. | Efficiency | Scalability | Simplicity | Evolvability | Extensibility | Customiz. | Configur. | Reusability | Visibility | Portability | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CS | | | | | + | + | + | | | | | | | |
| LS | | | − | | + | | + | | | | + | | + | |
| LCS | CS+LS | | − | | ++ | + | ++ | | | | + | | + | |
| CSS | CS | − | | | ++ | + | + | | | | | + | | + |
| C$SS | CSS+$ | − | + | + | ++ | + | + | | | | | + | | + |
| LC$SS | LCS+C$SS | − | ± | + | +++ | ++ | ++ | | | | + | + | + | + |
| RS | CS | | | + | − | + | + | | | | | − | | |
| RDA | CS | | | + | − | − | | | | | | + | | − |

Table 3-3. Evaluation of Hierarchical Styles for Network-based Hypermedia

### 3.4.1 Client-Server (CS)

The client-server style is the most frequently encountered of the architectural styles for network-based applications. A server component, offering a set of services, listens for requests upon those services. A client component, desiring that a service be performed, sends a request to the server via a connector. The server either rejects or performs the request and sends a response back to the client. A variety of client-server systems are surveyed by Sinha [123] and Umar [131].

Andrews [6] describes client-server components as follows: A client is a triggering process; a server is a reactive process. Clients make requests that trigger reactions from servers. Thus, a client initiates activity at times of its choosing; it often then delays until its request has been serviced. On the other hand, a server waits for requests to be made and then reacts to them. A server is usually a non-terminating process and often provides service to more than one client.

Separation of concerns is the principle behind the client-server constraints. A proper separation of functionality should simplify the server component in order to improve scalability. This simplification usually takes the form of moving all of the user interface functionality into the client component. The separation also allows the two types of components to evolve independently, provided that the interface doesn't change.

The basic form of client-server does not constrain how application state is partitioned between client and server components. It is often referred to by the mechanisms used for the connector implementation, such as remote procedure call [23] or message-oriented middleware [131].

### 3.4.2 Layered System (LS) and Layered-Client-Server (LCS)

A layered system is organized hierarchically, each layer providing services to the layer above it and using services of the layer below it [53]. Although layered system is considered a "pure" style, its use within network-based systems is limited to its combination with the client-server style to provide layered-client-server.

Layered systems reduce coupling across multiple layers by hiding the inner layers from all except the adjacent outer layer, thus improving evolvability and reusability. Examples include the processing of layered communication protocols, such as the TCP/IP and OSI protocol stacks [138], and hardware interface libraries. The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance [32].

Layered-client-server adds proxy and gateway components to the client-server style. A proxy [116] acts as a shared server for one or more client components, taking requests and

forwarding them, with possible translation, to server components. A gateway component appears to be a normal server to clients or proxies that request its services, but is in fact forwarding those requests, with possible translation, to its "inner-layer" servers. These additional mediator components can be added in multiple layers to add features like load balancing and security checking to the system.

Architectures based on layered-client-server are referred to as two-tiered, three-tiered, or multi-tiered architectures in the information systems literature [131].

LCS is also a solution to managing identity in a large scale distributed system, where complete knowledge of all servers would be prohibitively expensive. Instead, servers are organized in layers such that rarely used services are handled by intermediaries rather than directly by each client [6].

### 3.4.3 Client-Stateless-Server (CSS)

The client-stateless-server style derives from client-server with the additional constraint that no *session state* is allowed on the server component. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is kept entirely on the client.

These constraints improve the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures [133]. Scalability is improved

because not having to store state between requests allows the server component to quickly free resources and further simplifies implementation.

The disadvantage of client-stateless-server is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests, since that data cannot be left on the server in a shared context.

### 3.4.4 Client-Cache-Stateless-Server (C$SS)

The client-cache-stateless-server style derives from the client-stateless-server and cache styles via the addition of cache components. A cache acts as a mediator between client and server in which the responses to prior requests can, if they are considered cacheable, be reused in response to later requests that are equivalent and likely to result in a response identical to that in the cache if the request were to be forwarded to the server. An example system that makes effective use of this style is Sun Microsystems' NFS [115].

The advantage of adding cache components is that they have the potential to partially or completely eliminate some interactions, improving efficiency and user-perceived performance.

### 3.4.5 Layered-Client-Cache-Stateless-Server (LC$SS)

The layered-client-cache-stateless-server style derives from both layered-client-server and client-cache-stateless-server through the addition of proxy and/or gateway components. An example system that uses an LC$SS style is the Internet domain name system (DNS).

The advantages and disadvantages of LC$SS are simply a combination of those for LCS and C$SS. However, note that we don't count the contributions of the CS style twice, since the benefits are not additive if they come from the same ancestral derivation.

### 3.4.6 Remote Session (RS)

The remote session style is a variant of client-server that attempts to minimize the complexity, or maximize the reuse, of the client components rather than the server component. Each client initiates a session on the server and then invokes a series of services on the server, finally exiting the session. Application state is kept entirely on the server. This style is typically used when it is desired to access a remote service using a generic client (e.g., TELNET [106]) or via an interface that mimics a generic client (e.g., FTP [107]).

The advantages of the remote session style are that it is easier to centrally maintain the interface at the server, reducing concerns about inconsistencies in deployed clients when functionality is extended, and improves efficiency if the interactions make use of extended session context on the server. The disadvantages are that it reduces scalability of the server, due to the stored application state, and reduces visibility of interactions, since a monitor would have to know the complete state of the server.

### 3.4.7 Remote Data Access (RDA)

The remote data access style [131] is a variant of client-server that spreads the application state across both client and server. A client sends a database query in a standard format, such as SQL, to a remote server. The server allocates a workspace and performs the query, which may result in a very large data set. The client can then make further operations upon the result set (such as table joins) or retrieve the result one piece at a time. The client must know about the data structure of the service to build structure-dependent queries.

The advantages of remote data access are that a large data set can be iteratively reduced on the server side without transmitting it across the network, improving efficiency, and visibility is improved by using a standard query language. The disadvantages are that the client needs to understand the same database manipulation concepts as the server implementation (lacking simplicity) and storing application context on the server decreases scalability. Reliability also suffers, since partial failure can leave the workspace in an unknown state. Transaction mechanisms (e.g., two-phase commit) can be used to fix the reliability problem, though at a cost of added complexity and interaction overhead.

## 3.5 Mobile Code Styles

Mobile code styles use mobility in order to dynamically change the distance between the processing and source of data or destination of results. These styles are comprehensively examined in Fuggetta et al. [50]. A site abstraction is introduced at the architectural level, as part of the active configuration, in order to take into account the location of the different components. Introducing the concept of location makes it possible to model the cost of an interaction between components at the design level. In particular, an interaction between components that share the same location is considered to have negligible cost when compared to an interaction involving communication through the network. By changing its location, a component may improve the proximity and quality of its interaction, reducing interaction costs and thereby improving efficiency and user-perceived performance.

In all of the mobile code styles, a data element is dynamically transformed into a component. Fuggetta et al. [50] use an analysis that compares the code's size as a data element to the savings in normal data transfer in order to determine whether mobility is desirable for a given action. This would be impossible to model from an architectural standpoint if the definition of software architecture excludes data elements.

| Style | Derivation | Net Perform. | UP Perform. | Efficiency | Scalability | Simplicity | Evolvability | Extensibility | Customiz. | Configur. | Reusability | Visibility | Portability | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| VM | | | | | | ± | | + | | | | − | + | |
| REV | CS+VM | | | + | − | ± | | + | + | | | − | + | − |
| COD | CS+VM | | + | + | + | ± | | + | | + | | − | | |
| LCODC$SS | LC$SS+COD | − | ++ | ++ | +4+ | +±+ | ++ | + | | + | + | ± | + | + |
| MA | REV+COD | | + | ++ | | ± | | ++ | + | + | | − | + | |

Table 3-4. Evaluation of Mobile Code Styles for Network-based Hypermedia

### 3.5.1 Virtual Machine (VM)

Underlying all of the mobile code styles is the notion of a virtual machine, or interpreter, style [53]. The code must be executed in some fashion, preferably within a controlled environment to satisfy security and reliability concerns, which is exactly what the virtual machine style provides. It is not, in itself, a network-based style, but it is commonly used as such when combined with a component in the client-server style (REV and COD styles).

Virtual machines are commonly used as the engine for scripting languages, including general purpose languages like Perl [134] and task-specific languages like PostScript [2]. The primary benefits are the separation between instruction and implementation on a particular platform (portability) and ease of extensibility. Visibility is reduced because it is hard to know what an executable will do simply by looking at the code. Simplicity is

reduced due to the need to manage the evaluation environment, but that may be compensated in some cases as a result of simplifying the static functionality.

### 3.5.2 Remote Evaluation (REV)

In the remote evaluation style [50], derived from the client-server and virtual machine styles, a client component has the know-how necessary to perform a service, but lacks the resources (CPU cycles, data source, etc.) required, which happen to be located at a remote site. Consequently, the client sends the know-how to a server component at the remote site, which in turn executes the code using the resources available there. The results of that execution are then sent back to the client. The remote evaluation style assumes that the provided code will be executed in a sheltered environment, such that it won't impact other clients of the same server aside from the resources being used.

The advantages of remote evaluation include the ability to customize the server component's services, which provides for improved extensibility and customizability, and better efficiency when the code can adapt its actions to the environment inside the server (as opposed to the client making a series of interactions to do the same). Simplicity is reduced due to the need to manage the evaluation environment, but that may be compensated in some cases as a result of simplifying the static server functionality. Scalability is reduced; this can be improved with the server's management of the execution environment (killing long-running or resource-intensive code when resources are tight), but the management function itself leads to difficulties regarding partial failure and reliability. The most significant limitation, however, is the lack of visibility due to the

client sending code instead of standardized queries. Lack of visibility leads to obvious deployment problems if the server cannot trust the clients.

### 3.5.3 Code on Demand (COD)

In the code-on-demand style [50], a client component has access to a set of resources, but not the know-how on how to process them. It sends a request to a remote server for the code representing that know-how, receives that code, and executes it locally.

The advantages of code-on-demand include the ability to add features to a deployed client, which provides for improved extensibility and configurability, and better user-perceived performance and efficiency when the code can adapt its actions to the client's environment and interact with the user locally rather than through remote interactions. Simplicity is reduced due to the need to manage the evaluation environment, but that may be compensated in some cases as a result of simplifying the client's static functionality. Scalability of the server is improved, since it can off-load work to the client that would otherwise have consumed its resources. Like remote evaluation, the most significant limitation is the lack of visibility due to the server sending code instead of simple data. Lack of visibility leads to obvious deployment problems if the client cannot trust the servers.

### 3.5.4 Layered-Code-on-Demand-Client-Cache-Stateless-Server (LCODC$SS)

As an example of how some architectures are complementary, consider the addition of code-on-demand to the layered-client-cache-stateless-server style discussed above. Since the code can be treated as just another data element, this does not interfere with the

advantages of the LC$SS style. An example is the HotJava Web browser [java.sun.com], which allows applets and protocol extensions to be downloaded as typed media.

The advantages and disadvantages of LCODC$SS are just a combination of those for COD and LC$SS. We could go further and discuss the combination of COD with other CS styles, but this survey is not intended to be exhaustive (nor exhausting).

### 3.5.5 Mobile Agent (MA)

In the mobile agent style [50], an entire computational component is moved to a remote site, along with its state, the code it needs, and possibly some data required to perform the task. This can be considered a derivation of the remote evaluation and code-on-demand styles, since the mobility works both ways.

The primary advantage of the mobile agent style, beyond those already described for REV and COD, is that there is greater dynamism in the selection of when to move the code. An application can be in the midst of processing information at one location when it decides to move to another location, presumably in order to reduce the distance between it and the next set of data it wishes to process. In addition, the reliability problem of partial failure is reduced because the application state is in one location at a time [50].

## 3.6 Peer-to-Peer Styles

| Style | Derivation | Net Perform. | UP Perform. | Efficiency | Scalability | Simplicity | Evolvability | Extensibility | Customiz. | Configur. | Reusability | Visibility | Portability | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EBI | | | | + | −− | ± | + | + | | + | + | − | | − |
| C2 | EBI+LCS | | − | + | | + | ++ | + | | + | ++ | ± | + | ± |
| DO | CS+CS | − | | + | | | + | + | | + | + | − | | − |
| BDO | DO+LCS | − | − | | | | ++ | + | | + | ++ | − | + | |

Table 3-5. Evaluation of Peer-to-Peer Styles for Network-based Hypermedia

### 3.6.1 Event-based Integration (EBI)

The event-based integration style, also known as the implicit invocation or event system style, reduces coupling between components by removing the need for identity on the connector interface. Instead of invoking another component directly, a component can announce (or broadcast) one or more events. Other components in a system can register interest in that type of event and, when the event is announced, the system itself invokes all of the registered components [53]. Examples include the Model-View-Controller paradigm in Smalltalk-80 [72] and the integration mechanisms of many software engineering environments, including Field [113], SoftBench [29], and Polylith [110].

The event-based integration style provides strong support for extensibility through the ease of adding new components that listen for events, for reuse by encouraging a general event interface and integration mechanism, and for evolution by allowing components to be replaced without affecting the interfaces of other components [53]. Like pipe-and-filter systems, a higher-level configuring architecture is needed for the “invisible hand” that places components on the event interface. Most EBI systems also include explicit invocation as a complementary form of interaction [53]. For applications that are

dominated by data monitoring, rather than data retrieval, EBI can improve efficiency by removing the need for polling interactions.

The basic form of EBI system consists of one event bus to which all components listen for events of interest to them. Of course, this immediately leads to scalability issues with regard to the number of notifications, event storms as other components broadcast as a result of events caused by that notification, and a single point of failure in the notification delivery system. This can be ameliorated though the use of layered systems and filtering of events, at the cost of simplicity.

Other disadvantages of EBI systems are that it can be hard to anticipate what will happen in response to an action (poor understandability) and event notifications are not suitable for exchanging large-grain data [53]. Also, there is no support for recovery from partial failure.

### 3.6.2 C2

The C2 architectural style [128] is directed at supporting large-grain reuse and flexible composition of system components by enforcing substrate independence. It does so by combining event-based integration with layered-client-server. Asynchronous notification messages going down, and asynchronous request messages going up, are the sole means of intercomponent communication. This enforces loose coupling of dependency on higher layers (service requests may be ignored) and zero coupling with lower levels (no knowledge of notification usage), improving control over the system without losing most of the advantages of EBI.

Notifications are announcements of a state change within a component. C2 does not constrain what should be included with a notification: a flag, a delta of state change, or a complete state representation are all possibilities. A connector's primary responsibility is the routing and broadcasting of messages; its secondary responsibility is message filtering. The introduction of layered filtering of messages solves the EBI problems with scalability, while improving evolvability and reusability as well. Heavyweight connectors that include monitoring capabilities can be used to improve visibility and reduce the reliability problems of partial failure.

### 3.6.3 Distributed Objects

The distributed objects style organizes a system as a set of components interacting as peers. An object is an entity that encapsulates some private state information or data, a set of associated operations or procedures that manipulate the data, and possibly a thread of control, so that collectively they can be considered a single unit [31]. In general, an object's state is completely hidden and protected from all other objects. The only way it can be examined or modified is by making a request or invocation on one of the object's publicly accessible operations. This creates a well-defined interface for each object, enabling the specification of an object's operations to be made public while at the same time keeping the implementation of its operations and the representation of its state information private, thus improving evolvability.

An operation may invoke other operations, possibly on other objects. These operations may in turn make invocations on others, and so on. A chain of related invocations is referred to as an action [31]. State is distributed among the objects. This can be

advantageous in terms of keeping the state where it is most likely to be up-to-date, but has the disadvantage in that it is difficult to obtain an overall view of system activity (poor visibility).

In order for one object to interact with another, it must know the identity of that other object. When the identity of an object changes, it is necessary to modify all other objects that explicitly invoke it [53]. There must be some controller object that is responsible for maintaining the system state in order to complete the application requirements. Central issues for distributed object systems include: object management, object interaction management, and resource management [31].

Object systems are designed to isolate the data being processed. As a consequence, data streaming is not supported in general. However, this does provide better support for object mobility when combined with the mobile agent style.

### 3.6.4 Brokered Distributed Objects

In order to reduce the impact of identity, modern distributed object systems typically use one or more intermediary styles to facilitate communication. This includes event-based integration and brokered client/server [28]. The brokered distributed object style introduces name resolver components whose purpose is to answer client object requests for general service names with the specific name of an object that will satisfy the request. Although improving reusability and evolvability, the extra level of indirection requires additional network interactions, reducing efficiency and user-perceived performance.

Brokered distributed object systems are currently dominated by the industrial standards development of CORBA within the OMG [97] and the international standards development of Open Distributed Processing (ODP) within ISO/IEC [66].

In spite of all the interest associated with distributed objects, they fare poorly when compared to most other network-based architectural styles. They are best used for applications that involve the remote invocation of encapsulated services, such as hardware devices, where the efficiency and frequency of network interactions is less a concern.

## 3.7 Limitations

Each architectural style promotes a certain type of interaction among components. When components are distributed across a wide-area network, use or misuse of the network drives application usability. By characterizing styles by their influence on architectural properties, and particularly on the network-based application performance of a distributed hypermedia system, we gain the ability to better choose a software design that is appropriate for the application. There are, however, a couple limitations with the chosen classification.

The first limitation is that the evaluation is specific to the needs of distributed hypermedia. For example, many of the good qualities of the pipe-and-filter style disappear if the communication is fine-grained control messages, and are not applicable at all if the communication requires user interactivity. Likewise, layered caching only adds to latency, without any benefit, if none of the responses to client requests are cacheable. This type of distinction does not appear in the classification, and is only addressed informally in the discussion of each style. I believe this limitation can be overcome by creating separate

classification tables for each type of communication problem. Example problem areas would include, among others, large grain data retrieval, remote information monitoring, search, remote control systems, and distributed processing.

A second limitation is with the grouping of architectural properties. In some cases, it is better to identify the specific aspects of, for example, understandability and verifiability induced by an architectural style, rather than lumping them together under the rubric of simplicity. This is particularly the case for styles which might improve verifiability at the expense of understandability. However, the more abstract notion of a property also has value as a single metric, since we do not want to make the classification so specific that no two styles impact the same category. One solution would be a classification that presented both the specific properties and a summary property.

Regardless, this initial survey and classification is a necessary prerequisite to any further classifications that might address its limitations.

## 3.8 Related Work

### 3.8.1 Classification of Architectural Styles and Patterns

The area of research most directly related to this chapter is the identification and classification of architectural styles and architecture-level patterns.

Shaw [117] describes a few architectural styles, later expanded in Garlan and Shaw [53]. A preliminary classification of these styles is presented in Shaw and Clements [122] and repeated in Bass et al. [9], in which a two-dimensional, tabular classification strategy is used with control and data issues as the primary axes, organized by the following categories of features: which kinds of components and connectors are used in the style;

how control is shared, allocated, and transferred among the components; how data is communicated through the system; how data and control interact; and, what type of reasoning is compatible with the style. The primary purpose of the taxonomy is to identify style characteristics, rather than to assist in their comparison. It concludes with a small set of "rules of thumb" as a form of design guidance

Unlike this chapter, the Shaw and Clements [122] classification does not assist in evaluating designs in a way that is useful to an application designer. The problem is that the purpose of building software is not to build a specific shape, topology or component type, so organizing the classification in that fashion does not help a designer find a style that corresponds to their needs. It also mixes the essential differences among styles with other issues which have only incidental significance, and obscures the derivation relationships among styles. Furthermore, it does not focus on any particular type of architecture, such as network-based applications. Finally, it does not describe how styles can be combined, nor the effect of their combination.

Buschmann and Meunier [27] describe a classification scheme that organizes patterns according to granularity of abstraction, functionality, and structural principles. The granularity of abstraction separates patterns into three categories: architectural frameworks (templates for architectures), design patterns, and idioms. Their classification addresses some of the same issues as this dissertation, such as separation of concerns and structural principles that lead to architectural properties, but only covers two of the architectural styles described here. Their classification is considerably expanded in Buschmann et al. [28] with more extensive discussion of architectural patterns and their relation to software architecture.

Zimmer [137] organizes design patterns using a graph based on their relationships, making it easier to understand the overall structure of the patterns in the Gamma et al. [51] catalog. However, the patterns classified are not architectural patterns, and the classification is based exclusively on derivation or uses relationships rather than on architectural properties.

### 3.8.2 Distributed Systems and Programming Paradigms

Andrews [6] surveys how processes in a distributed program interact via message passing. He defines concurrent programs, distributed programs, kinds of processes in a distributed program (filters, clients, servers, peers), interaction paradigms, and communication channels. Interaction paradigms represent the communication aspects of software architectural styles. He describes paradigms for one-way data flow through networks of filters (pipe-and-filter), client-server, heartbeat, probe/echo, broadcast, token passing, replicated servers, and replicated workers with bag of tasks. However, the presentation is from the perspective of multiple processes cooperating on a single task, rather than general network-based architectural styles.

Sullivan and Notkin [126] provide a survey of implicit invocation research and describe its application to improving the evolution quality of software tool suites. Barrett et al. [8] present a survey of event-based integration mechanisms by building a framework for comparison and then seeing how some systems fit within that framework. Rosenblum and Wolf [114] investigate a design framework for Internet-scale event notification. All are concerned with the scope and requirements of an EBI style, rather than providing solutions for network-based systems.

Fuggetta et al. [50] provide a thorough examination and classification of mobile code paradigms. This chapter builds upon their work to the extent that I compare the mobile code styles with other network-capable styles, and place them within a single framework and set of architectural definitions.

### 3.8.3 Middleware

Bernstein [22] defines middleware as a distributed system service that includes standard programming interfaces and protocols. These services are called middleware because they act as a layer above the OS and networking software and below industry-specific applications. Umar [131] presents an extensive treatment of the subject.

Architecture research regarding middleware focuses on the problems and effects of integrating components with off-the-shelf middleware. Di Nitto and Rosenblum [38] describe how the usage of middleware and predefined components can influence the architecture of a system being developed and, conversely, how specific architectural choices can constrain the selection of middleware. Dashofy et al. [35] discuss the use of middleware with the C2 style.

Garlan et al. [56] point out some of the architectural assumptions within off-the-shelf components, examining the authors' problems with reusing subsystems in creating the Aesop tool for architectural design [54]. They classify the problems into four main categories of assumptions that can contribute to architectural mismatch: nature of components, nature of connectors, global architectural structure, and construction process.

## 3.9 Summary

This chapter has presented a survey of common architectural styles for network-based application software within a classification framework that evaluates each style according to the architectural properties it would induce if applied to an architecture for a prototypical network-based hypermedia system. The overall classification is summarized below in Table 3-6.

The next chapter uses the insight garnered from this survey and classification to hypothesize methods for developing and evaluating an architectural style to guide the design of improvements for the modern World Wide Web architecture.

| Style | Derivation | Net Perform. | UP Perform. | Efficiency | Scalability | Simplicity | Evolvability | Extensibility | Customiz. | Configur. | Reusability | Visibility | Portability | Reliability |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PF | | | ± | | | + | + | + | | + | + | | | |
| UPF | PF | − | ± | | | ++ | + | + | | ++ | ++ | + | | |
| RR | | | ++ | | + | | | | | | | | | + |
| $ | RR | | + | + | + | + | | | | | | | | |
| CS | | | | | + | + | + | | | | | | | |
| LS | | | − | | + | | + | | | | + | | + | |
| LCS | CS+LS | | − | | ++ | + | ++ | | | | + | | + | |
| CSS | CS | − | | | ++ | + | + | | | | | + | | + |
| C$SS | CSS+$ | − | + | + | ++ | + | + | | | | | + | | + |
| LC$SS | LCS+C$SS | − | ± | + | +++ | ++ | ++ | | | | + | + | + | + |
| RS | CS | | | + | − | + | + | | | | | − | | |
| RDA | CS | | | + | − | − | | | | | | + | | − |
| VM | | | | | | ± | | + | | | | − | + | |
| REV | CS+VM | | | + | − | ± | | + | + | | | − | + | − |
| COD | CS+VM | | + | + | + | ± | | + | | + | | − | | |
| LCODC$SS | LC$SS+COD | − | ++ | ++ | +4+ | +±+ | ++ | + | | + | + | ± | + | + |
| MA | REV+COD | | + | ++ | | ± | | ++ | + | + | | − | + | |
| EBI | | | | + | −− | ± | + | + | | + | + | − | | − |
| C2 | EBI+LCS | | − | + | | + | ++ | + | | + | ++ | ± | + | ± |
| DO | CS+CS | − | | + | | | + | + | | + | + | − | | − |
| BDO | DO+LCS | − | − | | | | ++ | + | | + | ++ | − | + | |

Table 3-6. Evaluation Summary

# CHAPTER 4
# Designing the Web Architecture: Problems and Insights

This chapter presents the requirements of the World Wide Web architecture and the problems faced in designing and evaluating proposed improvements to its key communication protocols. I use the insights garnered from the survey and classification of architectural styles for network-based hypermedia systems to hypothesize methods for developing an architectural style that would be used to guide the design of improvements for the modern Web architecture.

## 4.1 WWW Application Domain Requirements

Berners-Lee [20] writes that the "Web's major goal was to be a shared information space through which people and machines could communicate." What was needed was a way for people to store and structure their own information, whether permanent or ephemeral in nature, such that it could be usable by themselves and others, and to be able to reference and structure the information stored by others so that it would not be necessary for everyone to keep and maintain local copies.

The intended end-users of this system were located around the world, at various university and government high-energy physics research labs connected via the Internet. Their machines were a heterogeneous collection of terminals, workstations, servers and supercomputers, requiring a hodge podge of operating system software and file formats. The information ranged from personal research notes to organizational phone listings. The challenge was to build a system that would provide a universally consistent interface to

this structured information, available on as many platforms as possible, and incrementally deployable as new people and organizations joined the project.

### 4.1.1 Low Entry-barrier

Since participation in the creation and structuring of information was voluntary, a low entry-barrier was necessary to enable sufficient adoption. This applied to all users of the Web architecture: readers, authors, and application developers.

Hypermedia was chosen as the user interface because of its simplicity and generality: the same interface can be used regardless of the information source, the flexibility of hypermedia relationships (links) allows for unlimited structuring, and the direct manipulation of links allows the complex relationships within the information to guide the reader through an application. Since information within large databases is often much easier to access via a search interface rather than browsing, the Web also incorporated the ability to perform simple queries by providing user-entered data to a service and rendering the result as hypermedia.

For authors, the primary requirement was that partial availability of the overall system must not prevent the authoring of content. The hypertext authoring language needed to be simple and capable of being created using existing editing tools. Authors were expected to keep such things as personal research notes in this format, whether directly connected to the Internet or not, so the fact that some referenced information was unavailable, either temporarily or permanently, could not be allowed to prevent the reading and authoring of information that was available. For similar reasons, it was necessary to be able to create references to information before the target of that reference was available. Since authors

were encouraged to collaborate in the development of information sources, references needed to be easy to communicate, whether in the form of e-mail directions or written on the back of a napkin at a conference.

Simplicity was also a goal for the sake of application developers. Since all of the protocols were defined as text, communication could be viewed and interactively tested using existing network tools. This enabled early adoption of the protocols to take place in spite of the lack of standards.

### 4.1.2 Extensibility

While simplicity makes it possible to deploy an initial implementation of a distributed system, extensibility allows us to avoid getting stuck forever with the limitations of what was deployed. Even if it were possible to build a software system that perfectly matches the requirements of its users, those requirements will change over time just as society changes over time. A system intending to be as long-lived as the Web must be prepared for change.

### 4.1.3 Distributed Hypermedia

Hypermedia is defined by the presence of application control information embedded within, or as a layer above, the presentation of information. Distributed hypermedia allows the presentation and control information to be stored at remote locations. By its nature, user actions within a distributed hypermedia system require the transfer of large amounts of data from where the data is stored to where it is used. Thus, the Web architecture must be designed for large-grain data transfer.

The usability of hypermedia interaction is highly sensitive to user-perceived latency: the time between selecting a link and the rendering of a usable result. Since the Web's information sources are distributed across the global Internet, the architecture needs to minimize network interactions (round-trips within the data transfer protocols).

### 4.1.4 Internet-scale

The Web is intended to be an *Internet-scale* distributed hypermedia system, which means considerably more than just geographical dispersion. The Internet is about interconnecting information networks across multiple organizational boundaries. Suppliers of information services must be able to cope with the demands of anarchic scalability and the independent deployment of software components.

#### *4.1.4.1 Anarchic Scalability*

Most software systems are created with the implicit assumption that the entire system is under the control of one entity, or at least that all entities participating within a system are acting towards a common goal and not at cross-purposes. Such an assumption cannot be safely made when the system runs openly on the Internet. Anarchic scalability refers to the need for architectural elements to continue operating when they are subjected to an unanticipated load, or when given malformed or maliciously constructed data, since they may be communicating with elements outside their organizational control. The architecture must be amenable to mechanisms that enhance visibility and scalability.

The anarchic scalability requirement applies to all architectural elements. Clients cannot be expected to maintain knowledge of all servers. Servers cannot be expected to retain knowledge of state across requests. Hypermedia data elements cannot retain "back-

pointers," an identifier for each data element that references them, since the number of references to a resource is proportional to the number of people interested in that information. Particularly newsworthy information can also lead to "flash crowds": sudden spikes in access attempts as news of its availability spreads across the world.

Security of the architectural elements, and the platforms on which they operate, also becomes a significant concern. Multiple organizational boundaries implies that multiple trust boundaries could be present in any communication. Intermediary applications, such as firewalls, should be able to inspect the application interactions and prevent those outside the security policy of the organization from being acted upon. The participants in an application interaction should either assume that any information received is untrusted, or require some additional authentication before trust can be given. This requires that the architecture be capable of communicating authentication data and authorization controls. However, since authentication degrades scalability, the architecture's default operation should be limited to actions that do not need trusted data: a safe set of operations with well-defined semantics.

#### *4.1.4.2 Independent Deployment*

Multiple organizational boundaries also means that the system must be prepared for gradual and fragmented change, where old and new implementations co-exist without preventing the new implementations from making use of their extended capabilities. Existing architectural elements need to be designed with the expectation that later architectural features will be added. Likewise, older implementations need to be easily identified so that legacy behavior can be encapsulated without adversely impacting newer architectural elements. The architecture as a whole must be designed to ease the

deployment of architectural elements in a partial, iterative fashion, since it is not possible to force deployment in an orderly manner.

## 4.2 Problem

In late 1993, it became clear that more than just researchers would be interested in the Web. Adoption had occurred first in small research groups, spread to on-campus dorms, clubs, and personal home pages, and later to the institutional departments for campus information. When individuals began publishing their personal collections of information, on whatever topics they might feel fanatic about, the social network-effect launched an exponential growth of websites that continues today. Commercial interest in the Web was just beginning, but it was clear by then that the ability to publish on an international scale would be irresistible to businesses.

Although elated by its success, the Internet developer community became concerned that the rapid growth in the Web's usage, along with some poor network characteristics of early HTTP, would quickly outpace the capacity of the Internet infrastructure and lead to a general collapse. This was worsened by the changing nature of application interactions on the Web. Whereas the initial protocols were designed for single request-response pairs, new sites used an increasing number of in-line images as part of the content of Web pages, resulting in a different interaction profile for browsing. The deployed architecture had significant limitations in its support for extensibility, shared caching, and intermediaries, which made it difficult to develop ad-hoc solutions to the growing problems. At the same time, commercial competition within the software market led to an influx of new and occasionally contradictory feature proposals for the Web's protocols.

Working groups within the Internet Engineering Taskforce were formed to work on the Web's three primary standards: URI, HTTP, and HTML. The charter of these groups was to define the subset of existing architectural communication that was commonly and consistently implemented in the early Web architecture, identify problems within that architecture, and then specify a set of standards to solve those problems. This presented us with a challenge: how do we introduce a new set of functionality to an architecture that is already widely deployed, and how do we ensure that its introduction does not adversely impact, or even destroy, the architectural properties that have enabled the Web to succeed?

## 4.3 Approach

The early Web architecture was based on solid principles—separation of concerns, simplicity, and generality—but lacked an architectural description and rationale. The design was based on a set of informal hypertext notes [14], two early papers oriented towards the user community [12, 13], and archived discussions on the Web developer community mailing list (www-talk@info.cern.ch). In reality, however, the only true description of the early Web architecture was found within the implementations of libwww (the CERN protocol library for clients and servers), Mosaic (the NCSA browser client), and an assortment of other implementations that interoperated with them.

An architectural style can be used to define the principles behind the Web architecture such that they are visible to future architects. As discussed in Chapter 1, a style is a named set of constraints on architectural elements that induces the set of properties desired of the

architecture. The first step in my approach, therefore, is to identify the constraints placed within the early Web architecture that are responsible for its desirable properties.

> **Hypothesis I**: The design rationale behind the WWW architecture can be described by an architectural style consisting of the set of constraints applied to the elements within the Web architecture.

Additional constraints can be applied to an architectural style in order to extend the set of properties induced on instantiated architectures. The next step in my approach is to identify the properties desirable in an Internet-scale distributed hypermedia system, select additional architectural styles that induce those properties, and combine them with the early Web constraints to form a new, hybrid architectural style for the modern Web architecture.

> **Hypothesis II**: Constraints can be added to the WWW architectural style to derive a new hybrid style that better reflects the desired properties of a modern Web architecture.

Using the new architectural style as a guide, we can compare proposed extensions and modifications to the Web architecture against the constraints within the style. Conflicts indicate that the proposal would violate one or more of the design principles behind the Web. In some cases, the conflict could be removed by requiring the use of a specific indicator whenever the new feature is used, as is often done for HTTP extensions that impact the default cacheability of a response. For severe conflicts, such as a change in the interaction style, the same functionality would either be replaced with a design more conducive to the Web's style, or the proposer would be told to implement the functionality as a separate architecture running in parallel to the Web.

**Hypothesis III**: Proposals to modify the Web architecture can be compared to the updated WWW architectural style and analyzed for conflicts prior to deployment.

Finally, the updated Web architecture, as defined by the revised protocol standards that have been written according to the guidelines of the new architectural style, is deployed through participation in the development of the infrastructure and middleware software that make up the majority of Web applications. This included my direct participation in software development for the Apache HTTP server project and the libwww-perl client library, as well as indirect participation in other projects by advising the developers of the W3C libwww and jigsaw projects, the Netscape Navigator, Lynx, and MSIE browsers, and dozens of other implementations, as part of the IETF discourse.

Although I have described this approach as a single sequence, it is actually applied in a non-sequential, iterative fashion. That is, over the past six years I have been constructing models, adding constraints to the architectural style, and testing their affect on the Web's protocol standards via experimental extensions to client and server software. Likewise, others have suggested the addition of features to the architecture that were outside the scope of my then-current model style, but not in conflict with it, which resulted in going back and revising the architectural constraints to better reflect the improved architecture. The goal has always been to maintain a consistent and correct model of how I intend the Web architecture to behave, so that it could be used to guide the protocol standards that define appropriate behavior, rather than to create an artificial model that would be limited to the constraints originally imagined when the work began.

## 4.4 Summary

This chapter presented the requirements of the World Wide Web architecture and the problems faced in designing and evaluating proposed improvements to its key communication protocols. The challenge is to develop a method for designing improvements to an architecture such that the improvements can be evaluated prior to their deployment. My approach is to use an architectural style to define and improve the design rationale behind the Web's architecture, to use that style as the acid test for proving proposed extensions prior to their deployment, and to deploy the revised architecture via direct involvement in the software development projects that have created the Web's infrastructure.

The next chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, as it has been developed to represent the model for how the modern Web should work. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

# CHAPTER 5
# Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

## 5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.


Figure 5-1. Null Style

### 5.1.1 Starting with the Null Style

There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing—a blank slate, whiteboard, or drawing board—and builds-up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole, without constraints, and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. Where the first emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context. REST has been developed using the latter process. Figures 5-1 through 5-8 depict this graphically in terms of how the applied constraints would differentiate the process view of an architecture as the incremental set of constraints is applied.

The Null style (Figure 5-1) is simply an empty set of constraints. From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components. It is the starting point for our description of REST.

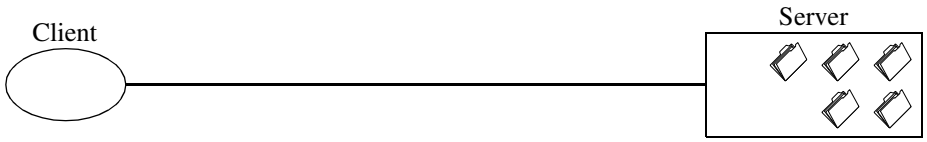


Figure 5-2. Client-Server

### 5.1.2 Client-Server

The first constraints added to our hybrid style are those of the client-server architectural style (Figure 5-2), described in Section 3.4.1. Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

### 5.1.3 Stateless

We next add a constraint to the client-server interaction: communication must be stateless in nature, as in the client-stateless-server (CSS) style of Section 3.4.3 (Figure 5-3), such that each request from client to server must contain all of the information necessary to

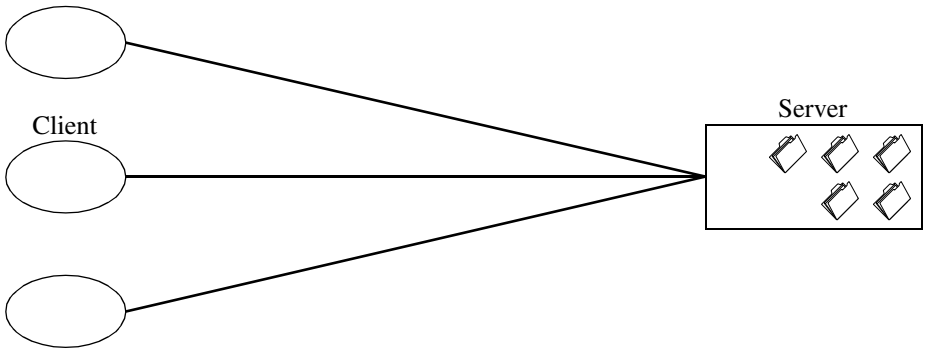


Figure 5-3. Client-Stateless-Server

understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

This constraint induces the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures [133]. Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests.

Like most architectural choices, the stateless constraint reflects a design trade-off. The disadvantage is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests, since that data cannot be left on the server in a shared context. In addition, placing the application state on the client-side reduces the server's control over consistent application behavior, since the application becomes dependent on the correct implementation of semantics across multiple client versions.

### 5.1.4 Cache

In order to improve network efficiency, we add cache constraints to form the client-cache-stateless-server style of Section 3.4.4 (Figure 5-4). Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

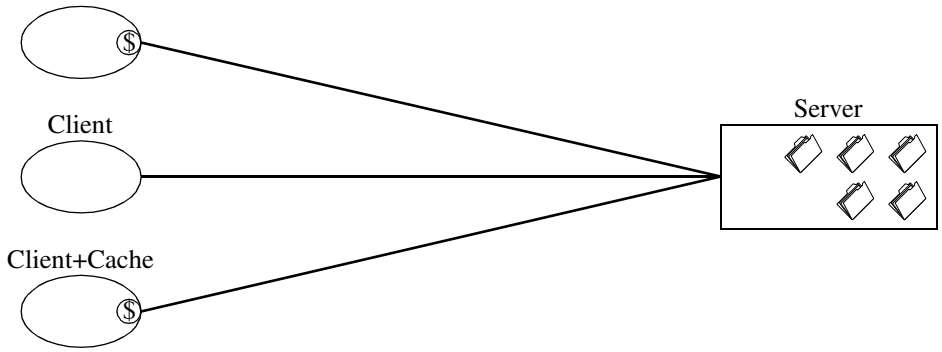


Figure 5-4. Client-Cache-Stateless-Server

The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

The early Web architecture, as portrayed by the diagram in Figure 5-5 [11], was defined by the client-cache-stateless-server set of constraints. That is, the design rationale presented for the Web architecture prior to 1994 focused on stateless client-server interaction for the exchange of static documents over the Internet. The protocols for communicating interactions had rudimentary support for non-shared caches, but did not constrain the interface to a consistent set of semantics for all resources. Instead, the Web relied on the use of a common client-server implementation library (CERN libwww) to maintain consistency across Web applications.

Developers of Web implementations had already exceeded the early design. In addition to static documents, requests could identify services that dynamically generated responses, such as image-maps [Kevin Hughes] and server-side scripts [Rob McCool].

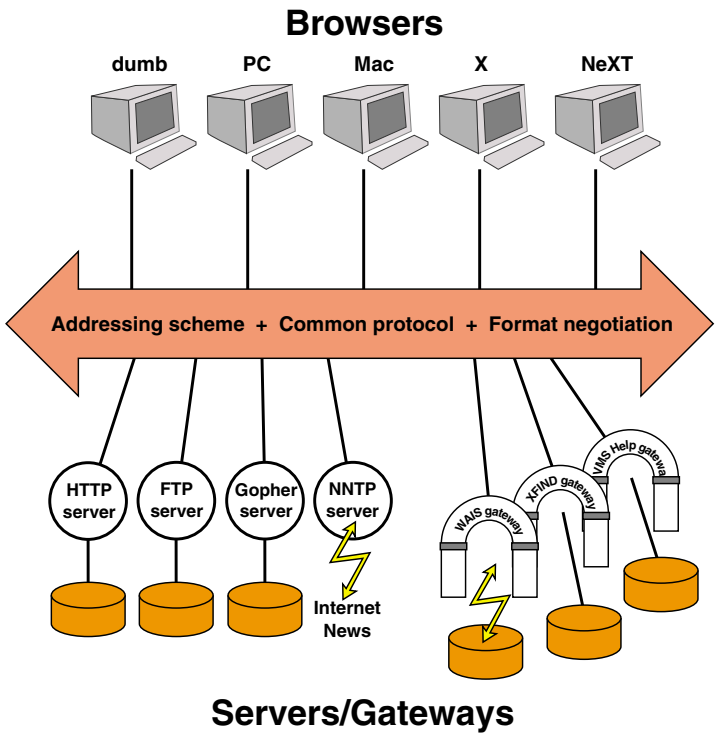




Figure 5-5. Early WWW Architecture Diagram

Work had also begun on intermediary components, in the form of proxies [79] and shared caches [59], but extensions to the protocols were needed in order for them to communicate reliably. The following sections describe the constraints added to the Web's architectural style in order to guide the extensions that form the modern Web architecture.

### 5.1.5 Uniform Interface

The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components (Figure 5-6). By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages

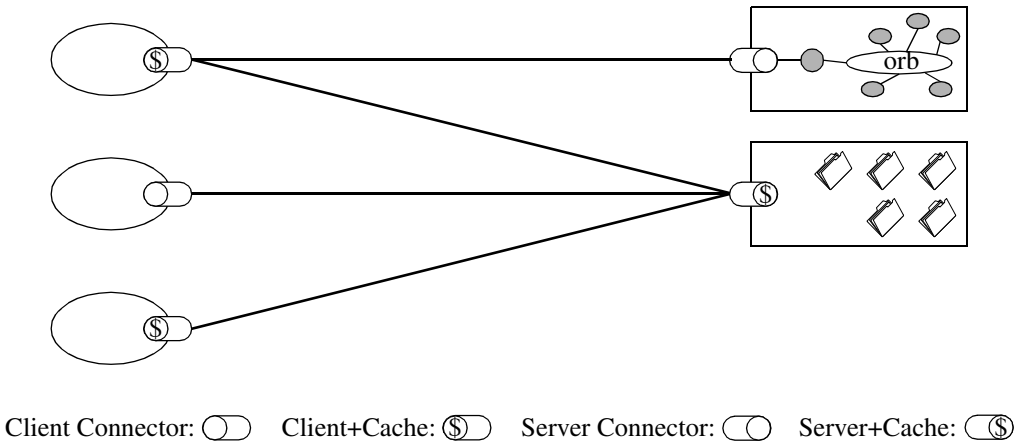


Figure 5-6. Uniform-Client-Cache-Stateless-Server

independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state. These constraints will be discussed in Section 5.2.

### 5.1.6 Layered System

In order to further improve behavior for Internet-scale requirements, we add layered system constraints (Figure 5-7). As described in Section 3.4.2, the layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which

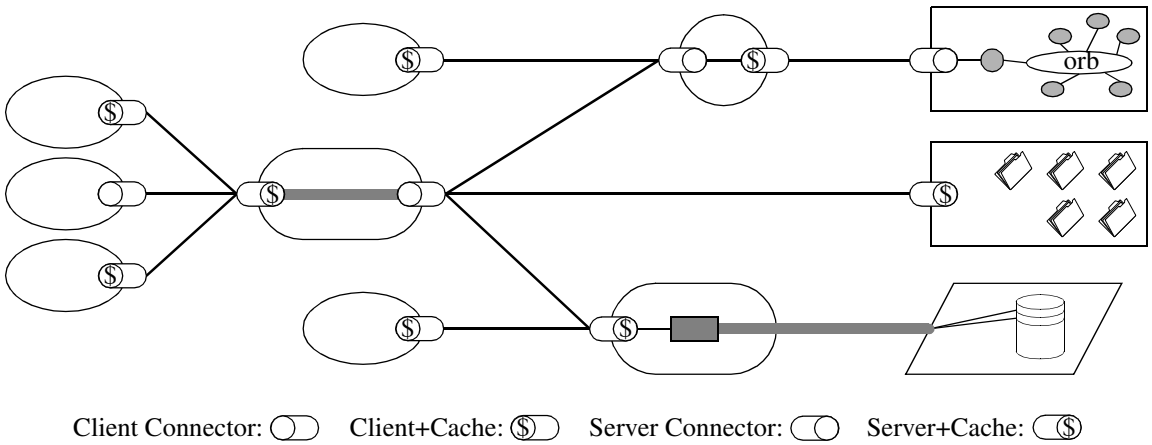


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.

The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance [32]. For a network-based system that supports cache constraints, this can be offset by the benefits of shared caching at intermediaries. Placing shared caches at the boundaries of an organizational domain can result in significant performance benefits [136]. Such layers also allow security policies to be enforced on data crossing the organizational boundary, as is required by firewalls [79].

The combination of layered system and uniform interface constraints induces architectural properties similar to those of the uniform pipe-and-filter style (Section 3.2.2). Although REST interaction is two-way, the large-grain data flows of

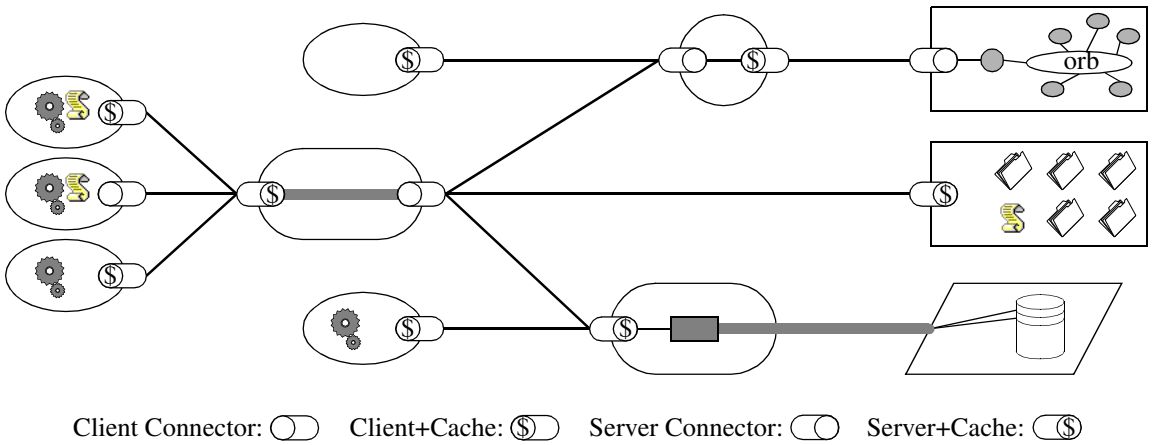


Figure 5-8. REST

hypermedia interaction can each be processed like a data-flow network, with filter components selectively applied to the data stream in order to transform the content as it passes [26]. Within REST, intermediary components can actively transform the content of messages because the messages are self-descriptive and their semantics are visible to intermediaries.

### 5.1.7 Code-On-Demand

The final addition to our constraint set for REST comes from the code-on-demand style of Section 3.5.3 (Figure 5-8). REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.

The notion of an optional constraint may seem like an oxymoron. However, it does have a purpose in the architectural design of a system that encompasses multiple organizational boundaries. It means that the architecture only gains the benefit (and suffers

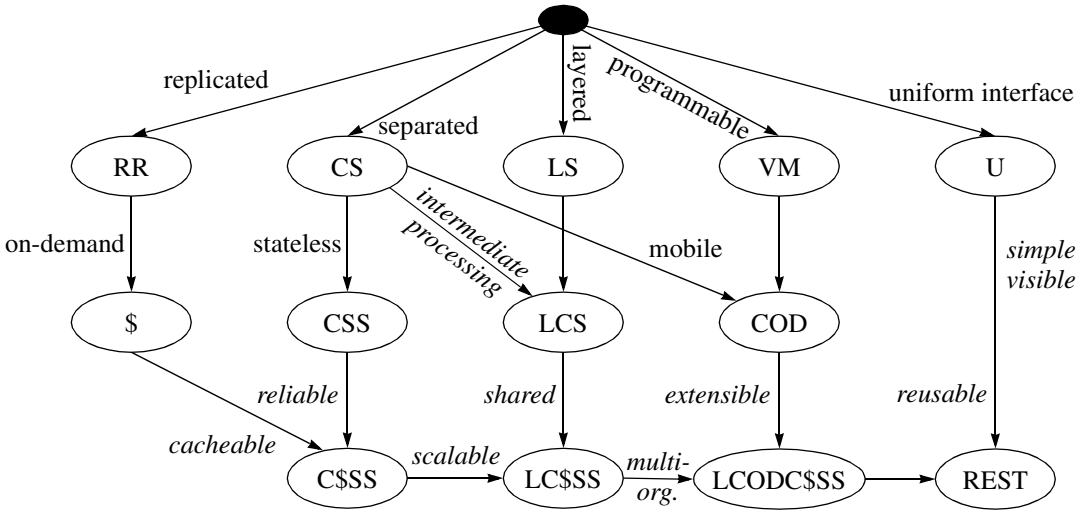


Figure 5-9. REST Derivation by Style Constraints

the disadvantages) of the optional constraints when they are known to be in effect for some realm of the overall system. For example, if all of the client software within an organization is known to support Java applets [45], then services within that organization can be constructed such that they gain the benefit of enhanced functionality via downloadable Java classes. At the same time, however, the organization's firewall may prevent the transfer of Java applets from external sources, and thus to the rest of the Web it will appear as if those clients do not support code-on-demand. An optional constraint allows us to design an architecture that supports the desired behavior in the general case, but with the understanding that it may be disabled within some contexts.

### 5.1.8 Style Derivation Summary

REST consists of a set of architectural constraints chosen for the properties they induce on candidate architectures. Although each of these constraints can be considered in isolation, describing them in terms of their derivation from common architectural styles makes it

easier to understand the rationale behind their selection. Figure 5-9 depicts the derivation of REST's constraints graphically in terms of the network-based architectural styles examined in Chapter 3.

## 5.2 REST Architectural Elements

The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.

### 5.2.1 Data Elements

Unlike the distributed object style [31], where all data is encapsulated within and hidden by the processing components, the nature and state of an architecture's data elements is a key aspect of REST. The rationale for this design can be seen in the nature of distributed hypermedia. When a link is selected, information needs to be moved from the location where it is stored to the location where it will be used by, in most cases, a human reader. This is unlike many other distributed processing paradigms [6, 50], where it is possible, and usually more efficient, to move the "processing agent" (e.g., mobile code, stored procedure, search expression, etc.) to the data rather than move the data to the processor.

A distributed hypermedia architect has only three fundamental options: 1) render the data where it is located and send a fixed-format image to the recipient; 2) encapsulate the

data with a rendering engine and send both to the recipient; or, 3) send the raw data to the recipient along with metadata that describes the data type, so that the recipient can choose their own rendering engine.

Each option has its advantages and disadvantages. Option 1, the traditional client-server style [31], allows all information about the true nature of the data to remain hidden within the sender, preventing assumptions from being made about the data structure and making client implementation easier. However, it also severely restricts the functionality of the recipient and places most of the processing load on the sender, leading to scalability problems. Option 2, the mobile object style [50], provides information hiding while enabling specialized processing of the data via its unique rendering engine, but limits the functionality of the recipient to what is anticipated within that engine and may vastly increase the amount of data transferred. Option 3 allows the sender to remain simple and scalable while minimizing the bytes transferred, but loses the advantages of information hiding and requires that both sender and recipient understand the same data types.

REST provides a hybrid of all three options by focusing on a shared understanding of data types with metadata, but limiting the scope of what is revealed to a standardized interface. REST components communicate by transferring a representation of a resource in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface. The benefits of the mobile object style are approximated by sending a representation that consists of instructions in the standard data format of an encapsulated rendering engine (e.g., Java [45]). REST therefore gains the

separation of concerns of the client-server style without the server scalability problem, allows information hiding through a generic interface to enable encapsulation and evolution of services, and provides for a diverse set of functionality through downloadable feature-engines.

REST's data elements are summarized in Table 5-1.

Table 5-1. REST Data Elements

| **Data Element** | **Modern Web Examples** |
|---|---|
| resource | the intended conceptual target of a hypertext reference |
| resource identifier | URL, URN |
| representation | HTML document, JPEG image |
| representation metadata | media type, last-modified time |
| resource metadata | source link, alternates, vary |
| control data | if-modified-since, cache-control |

#### *5.2.1.1 Resources and Resource Identifiers*

The key abstraction of information in REST is a *resource*. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

More precisely, a resource $R$ is a temporally varying membership function $M_R(t)$, which for time $t$ maps to a set of entities, or values, which are equivalent. The values in the set may be *resource representations* and/or *resource identifiers*. A resource can map to the

empty set, which allows references to be made to a concept before any realization of that concept exists — a notion that was foreign to most hypertext systems prior to the Web [61]. Some resources are static in the sense that, when examined at any time after their creation, they always correspond to the same value set. Others have a high degree of variance in their value over time. The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another.

For example, the "authors' preferred version" of an academic paper is a mapping whose value changes over time, whereas a mapping to "the paper published in the proceedings of conference X" is static. These are two distinct resources, even if they both map to the same value at some point in time. The distinction is necessary so that both resources can be identified and referenced independently. A similar example from software engineering is the separate identification of a version-controlled source code file when referring to the "latest revision", "revision number 1.2.7", or "revision included with the Orange release."

This abstract definition of a resource enables key features of the Web architecture. First, it provides generality by encompassing many sources of information without artificially distinguishing them by type or implementation. Second, it allows late binding of the reference to a representation, enabling content negotiation to take place based on characteristics of the request. Finally, it allows an author to reference the concept rather than some singular representation of that concept, thus removing the need to change all existing links whenever the representation changes (assuming the author used the right identifier).

REST uses a *resource identifier* to identify the particular resource involved in an interaction between components. REST connectors provide a generic interface for accessing and manipulating the value set of a resource, regardless of how the membership function is defined or the type of software that is handling the request. The naming authority that assigned the resource identifier, making it possible to reference the resource, is responsible for maintaining the semantic validity of the mapping over time (i.e., ensuring that the membership function does not change).

Traditional hypertext systems [61], which typically operate in a closed or local environment, use unique node or document identifiers that change every time the information changes, relying on link servers to maintain references separately from the content [135]. Since centralized link servers are an anathema to the immense scale and multi-organizational domain requirements of the Web, REST relies instead on the author choosing a resource identifier that best fits the nature of the concept being identified. Naturally, the quality of an identifier is often proportional to the amount of money spent to retain its validity, which leads to broken links as ephemeral (or poorly supported) information moves or disappears over time.

#### *5.2.1.2 Representations*

REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A *representation* is a sequence of bytes, plus *representation metadata* to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.

A representation consists of data, metadata describing the data, and, on occasion, metadata to describe the metadata (usually for the purpose of verifying message integrity). Metadata is in the form of name-value pairs, where the name corresponds to a standard that defines the value's structure and semantics. Response messages may include both representation metadata and *resource metadata*: information about the resource that is not specific to the supplied representation.

*Control data* defines the purpose of a message between components, such as the action being requested or the meaning of a response. It is also used to parameterize requests and override the default behavior of some connecting elements. For example, cache behavior can be modified by control data included in the request or response message.

Depending on the message control data, a given representation may indicate the current state of the requested resource, the desired state for the requested resource, or the value of some other resource, such as a representation of the input data within a client's query form, or a representation of some error condition for a response. For example, remote authoring of a resource requires that the author send a representation to the server, thus establishing a value for that resource that can be retrieved by later requests. If the value set of a resource at a given time consists of multiple representations, content negotiation may be used to select the best representation for inclusion in a given message.

The data format of a representation is known as a *media type* [48]. A representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type. Some media types are intended for automated processing, some are intended to be rendered for viewing by a user, and a few

are capable of both. Composite media types can be used to enclose multiple representations in a single message.

The design of a media type can directly impact the user-perceived performance of a distributed hypermedia system. Any data that must be received before the recipient can begin rendering the representation adds to the latency of an interaction. A data format that places the most important rendering information up front, such that the initial information can be incrementally rendered while the rest of the information is being received, results in much better user-perceived performance than a data format that must be entirely received before rendering can begin.

For example, a Web browser that can incrementally render a large HTML document while it is being received provides significantly better user-perceived performance than one that waits until the entire document is completely received prior to rendering, even though the network performance is the same. Note that the rendering ability of a representation can also be impacted by the choice of content. If the dimensions of dynamically-sized tables and embedded objects must be determined before they can be rendered, their occurrence within the viewing area of a hypermedia page will increase its latency.

### 5.2.2 Connectors

REST uses various connector types, summarized in Table 5-2, to encapsulate the activities of accessing resources and transferring resource representations. The connectors present an abstract interface for component communication, enhancing simplicity by providing a clean separation of concerns and hiding the underlying implementation of resources and

Table 5-2. REST Connectors

| Connector | Modern Web Examples |
|---|---|
| client | libwww, libwww-perl |
| server | libwww, Apache API, NSAPI |
| cache | browser cache, Akamai cache network |
| resolver | bind (DNS lookup library) |
| tunnel | SOCKS, SSL after HTTP CONNECT |

communication mechanisms. The generality of the interface also enables substitutability: if the users' only access to the system is via an abstract interface, the implementation can be replaced without impacting the users. Since a connector manages network communication for a component, information can be shared across multiple interactions in order to improve efficiency and responsiveness.

All REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it. This restriction accomplishes four functions: 1) it removes any need for the connectors to retain application state between requests, thus reducing consumption of physical resources and improving scalability; 2) it allows interactions to be processed in parallel without requiring that the processing mechanism understand the interaction semantics; 3) it allows an intermediary to view and understand a request in isolation, which may be necessary when services are dynamically rearranged; and, 4) it forces all of the information that might factor into the reusability of a cached response to be present in each request.

The connector interface is similar to procedural invocation, but with important differences in the passing of parameters and results. The in-parameters consist of request

control data, a resource identifier indicating the target of the request, and an optional representation. The out-parameters consist of response control data, optional resource metadata, and an optional representation. From an abstract viewpoint the invocation is synchronous, but both in and out-parameters can be passed as data streams. In other words, processing can be invoked before the value of the parameters is completely known, thus avoiding the latency of batch processing large data transfers.

The primary connector types are client and server. The essential difference between the two is that a *client* initiates communication by making a request, whereas a *server* listens for connections and responds to requests in order to supply access to its services. A component may include both client and server connectors.

A third connector type, the *cache* connector, can be located on the interface to a client or server connector in order to save cacheable responses to current interactions so that they can be reused for later requested interactions. A cache may be used by a client to avoid repetition of network communication, or by a server to avoid repeating the process of generating a response, with both cases serving to reduce interaction latency. A cache is typically implemented within the address space of the connector that uses it.

Some cache connectors are shared, meaning that its cached responses may be used in answer to a client other than the one for which the response was originally obtained. Shared caching can be effective at reducing the impact of “flash crowds” on the load of a popular server, particularly when the caching is arranged hierarchically to cover large groups of users, such as those within a company’s intranet, the customers of an Internet service provider, or Universities sharing a national network backbone. However, shared caching can also lead to errors if the cached response does not match what would have

been obtained by a new request. REST attempts to balance the desire for transparency in cache behavior with the desire for efficient use of the network, rather than assuming that absolute transparency is always required.

A cache is able to determine the cacheability of a response because the interface is generic rather than specific to each resource. By default, the response to a retrieval request is cacheable and the responses to other requests are non-cacheable. If some form of user authentication is part of the request, or if the response indicates that it should not be shared, then the response is only cacheable by a non-shared cache. A component can override these defaults by including control data that marks the interaction as cacheable, non-cacheable or cacheable for only a limited time.

A *resolver* translates partial or complete resource identifiers into the network address information needed to establish an inter-component connection. For example, most URI include a DNS hostname as the mechanism for identifying the naming authority for the resource. In order to initiate a request, a Web browser will extract the hostname from the URI and make use of a DNS resolver to obtain the Internet Protocol address for that authority. Another example is that some identification schemes (e.g., URN [124]) require an intermediary to translate a permanent identifier to a more transient address in order to access the identified resource. Use of one or more intermediate resolvers can improve the longevity of resource references through indirection, though doing so adds to the request latency.

The final form of connector type is a *tunnel*, which simply relays communication across a connection boundary, such as a firewall or lower-level network gateway. The only reason it is modeled as part of REST and not abstracted away as part of the network

infrastructure is that some REST components may dynamically switch from active component behavior to that of a tunnel. The primary example is an HTTP proxy that switches to a tunnel in response to a CONNECT method request [71], thus allowing its client to directly communicate with a remote server using a different protocol, such as TLS, that doesn't allow proxies. The tunnel disappears when both ends terminate their communication.

### 5.2.3 Components

REST components, summarized in Table 5-3, are typed by their roles in an overall application action.

Table 5-3. REST Components

| Component | Modern Web Examples |
|---|---|
| origin server | Apache httpd, Microsoft IIS |
| gateway | Squid, CGI, Reverse Proxy |
| proxy | CERN Proxy, Netscape Proxy, Gauntlet |
| user agent | Netscape Navigator, Lynx, MOMspider |

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.

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.

Intermediary components act as both a client and a server in order to forward, with possible translation, requests and responses. A *proxy* component is an intermediary selected by a client to provide interface encapsulation of other services, data translation, performance enhancement, or security protection. A *gateway* (a.k.a., *reverse proxy*) component is an intermediary imposed by the network or origin server to provide an interface encapsulation of other services, for data translation, performance enhancement, or security enforcement. Note that the difference between a proxy and a gateway is that a client determines when it will use a proxy.

## 5.3 REST Architectural Views

Now that we have an understanding of the REST architectural elements in isolation, we can use architectural views [105] to describe how the elements work together to form an architecture. Three types of view—process, connector, and data—are useful for illuminating the design principles of REST.

### 5.3.1 Process View

A process view of an architecture is primarily effective at eliciting the interaction relationships among components by revealing the path of data as it flows through the system. Unfortunately, the interaction of a real system usually involves an extensive number of components, resulting in an overall view that is obscured by the details.

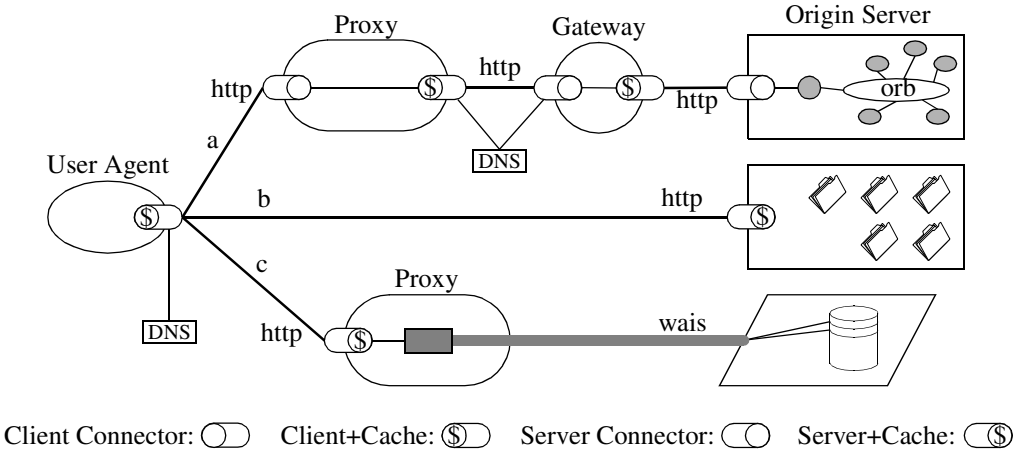


Figure 5-10. Process View of a REST-based Architecture

A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according to the properties of each resource identifier and the configuration of the client connector. Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture. Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache. Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface. Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.

Figure 5-10 provides a sample of the process view from a REST-based architecture at a particular instance during the processing of three parallel requests.

REST's client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—proxies, gateways, and firewalls—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests,

standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.

Since the components are connected dynamically, their arrangement and function for a particular application action has characteristics similar to a pipe-and-filter style. Although REST components communicate via bidirectional streams, the processing of each direction is independent and therefore susceptible to stream transducers (filters). The generic connector interface allows components to be placed on the stream based on the properties of each request or response.

Services may be implemented using a complex hierarchy of intermediaries and multiple distributed origin servers. The stateless nature of REST allows each interaction to be independent of the others, removing the need for an awareness of the overall component topology, an impossible task for an Internet-scale architecture, and allowing components to act as either destinations or intermediaries, determined dynamically by the target of each request. Connectors need only be aware of each other's existence during the scope of their communication, though they may cache the existence and capabilities of other components for performance reasons.

### 5.3.2 Connector View

A connector view of an architecture concentrates on the mechanics of the communication between components. For a REST-based architecture, we are particularly interested in the constraints that define the generic resource interface.

Client connectors examine the resource identifier in order to select an appropriate communication mechanism for each request. For example, a client may be configured to

connect to a specific proxy component, perhaps one acting as an annotation filter, when the identifier indicates that it is a local resource. Likewise, a client can be configured to reject requests for some subset of identifiers.

REST does not restrict communication to a particular protocol, but it does constrain the interface between components, and hence the scope of interaction and implementation assumptions that might otherwise be made between components. For example, the Web's primary transfer protocol is HTTP, but the architecture also includes seamless access to resources that originate on pre-existing network servers, including FTP [107], Gopher [7], and WAIS [36]. Interaction with those services is restricted to the semantics of a REST connector. This constraint sacrifices some of the advantages of other architectures, such as the stateful interaction of a relevance feedback protocol like WAIS, in order to retain the advantages of a single, generic interface for connector semantics. In return, the generic interface makes it possible to access a multitude of services through a single proxy. If an application needs the additional capabilities of another architecture, it can implement and invoke those capabilities as a separate system running in parallel, similar to how the Web architecture interfaces with "telnet" and "mailto" resources.

### 5.3.3 Data View

A data view of an architecture reveals the application state as information flows through the components. Since REST is specifically targeted at distributed information systems, it views an application as a cohesive structure of information and control alternatives through which a user can perform a desired task. For example, looking-up a word in an on-line dictionary is one application, as is touring through a virtual museum, or reviewing

a set of class notes to study for an exam. Each application defines goals for the underlying system, against which the system's performance can be measured.

Component interactions occur in the form of dynamically sized messages. Small or medium-grain messages are used for control semantics, but the bulk of application work is accomplished via large-grain messages containing a complete resource representation. The most frequent form of request semantics is that of retrieving a representation of a resource (e.g., the "GET" method in HTTP), which can often be cached for later reuse.

REST concentrates all of the control state into the representations received in response to interactions. The goal is to improve server scalability by eliminating any need for the server to maintain an awareness of the client state beyond the current request. An application's state is therefore defined by its pending requests, the topology of connected components (some of which may be filtering buffered data), the active requests on those connectors, the data flow of representations in response to those requests, and the processing of those representations as they are received by the user agent.

An application reaches a steady-state whenever it has no outstanding requests; i.e., it has no pending requests and all of the responses to its current set of requests have been completely received or received to the point where they can be treated as a representation data stream. For a browser application, this state corresponds to a "web page," including the primary representation and ancillary representations, such as in-line images, embedded applets, and style sheets. The significance of application steady-states is seen in their impact on both user-perceived performance and the burstiness of network request traffic.

The user-perceived performance of a browser application is determined by the latency between steady-states: the period of time between the selection of a hypermedia link on one web page and the point when usable information has been rendered for the next web page. The optimization of browser performance is therefore centered around reducing this communication latency.

Since REST-based architectures communicate primarily through the transfer of representations of resources, latency can be impacted by both the design of the communication protocols and the design of the representation data formats. The ability to incrementally render the response data as it is received is determined by the design of the media type and the availability of layout information (visual dimensions of in-line objects) within each representation.

An interesting observation is that the most efficient network request is one that doesn't use the network. In other words, the ability to reuse a cached response results in a considerable improvement in application performance. Although use of a cache adds some latency to each individual request due to lookup overhead, the average request latency is significantly reduced when even a small percentage of requests result in usable cache hits.

The next control state of an application resides in the representation of the first requested resource, so obtaining that first representation is a priority. REST interaction is therefore improved by protocols that "respond first and think later." In other words, a protocol that requires multiple interactions per user action, in order to do things like negotiate feature capabilities prior to sending a content response, will be perceptively slower than a protocol that sends whatever is most likely to be optimal first and then provides a list of alternatives for the client to retrieve if the first response is unsatisfactory.

The application state is controlled and stored by the user agent and can be composed of representations from multiple servers. In addition to freeing the server from the scalability problems of storing state, this allows the user to directly manipulate the state (e.g., a Web browser's history), anticipate changes to that state (e.g., link maps and prefetching of representations), and jump from one application to another (e.g., bookmarks and URI-entry dialogs).

The model application is therefore an engine that moves from one state to the next by examining and choosing from among the alternative state transitions in the current set of representations. Not surprisingly, this exactly matches the user interface of a hypermedia browser. However, the style does not assume that all applications are browsers. In fact, the application details are hidden from the server by the generic connector interface, and thus a user agent could equally be an automated robot performing information retrieval for an indexing service, a personal agent looking for data that matches certain criteria, or a maintenance spider busy patrolling the information for broken references or modified content [39].

## 5.4 Related Work

Bass, et al. [9] devote a chapter on architecture for the World Wide Web, but their description only encompasses the implementation architecture within the CERN/W3C developed libwww (client and server libraries) and Jigsaw software. Although those implementations reflect many of the design constraints of REST, having been developed by people familiar with the Web's architectural design and rationale, the real WWW architecture is independent of any single implementation. The modern Web is defined by

its standard interfaces and protocols, not how those interfaces and protocols are implemented in a given piece of software.

The REST style draws from many preexisting distributed process paradigms [6, 50], communication protocols, and software fields. REST component interactions are structured in a layered client-server style, but the added constraints of the generic resource interface create the opportunity for substitutability and inspection by intermediaries. Requests and responses have the appearance of a remote invocation style, but REST messages are targeted at a conceptual resource rather than an implementation identifier.

Several attempts have been made to model the Web architecture as a form of distributed file system (e.g., WebNFS) or as a distributed object system [83]. However, they exclude various Web resource types or implementation strategies as being "not interesting," when in fact their presence invalidates the assumptions that underlie such models. REST works well because it does not limit the implementation of resources to certain predefined models, allowing each application to choose an implementation that best matches its own needs and enabling the replacement of implementations without impacting the user.

The interaction method of sending representations of resources to consuming components has some parallels with event-based integration (EBI) styles. The key difference is that EBI styles are push-based. The component containing the state (equivalent to an origin server in REST) issues an event whenever the state changes, whether or not any component is actually interested in or listening for such an event. In the REST style, consuming components usually pull representations. Although this is less

efficient when viewed as a single client wishing to monitor a single resource, the scale of the Web makes an unregulated push model infeasible.

The principled use of the REST style in the Web, with its clear notion of components, connectors, and representations, relates closely to the C2 architectural style [128]. The C2 style supports the development of distributed, dynamic applications by focusing on structured use of connectors to obtain substrate independence. C2 applications rely on asynchronous notification of state changes and request messages. As with other event-based schemes, C2 is nominally push-based, though a C2 architecture could operate in REST's pull style by only emitting a notification upon receipt of a request. However, the C2 style lacks the intermediary-friendly constraints of REST, such as the generic resource interface, guaranteed stateless interactions, and intrinsic support for caching.

## 5.5 Summary

This chapter introduced the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I described the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles.

The next chapter presents an evaluation of the REST architecture through the experience and lessons learned from applying REST to the design, specification, and

deployment of the modern Web architecture. This work included authoring the current Internet standards-track specifications of the Hypertext Transfer Protocol (HTTP/1.1) and Uniform Resource Identifiers (URI), and implementing the architecture through the libwww-perl client protocol library and Apache HTTP server.

# CHAPTER 6
# Experience and Evaluation

Since 1994, the REST architectural style has been used to guide the design and development of the architecture for the modern Web. This chapter describes the experience and lessons learned from applying REST while authoring the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI), the two specifications that define the generic interface used by all component interactions on the Web, as well as from the deployment of these technologies in the form of the libwww-perl client library, the Apache HTTP Server Project, and other implementations of the protocol standards.

## 6.1 Standardizing the Web

As described in Chapter 4, the motivation for developing REST was to create an architectural model for how the Web should work, such that it could serve as the guiding framework for the Web protocol standards. REST has been applied to describe the desired Web architecture, help identify existing problems, compare alternative solutions, and ensure that protocol extensions would not violate the core constraints that make the Web successful. This work was done as part of the Internet Engineering Taskforce (IETF) and World Wide Web Consortium (W3C) efforts to define the architectural standards for the Web: HTTP, URI, and HTML.

My involvement in the Web standards process began in late 1993, while developing the libwww-perl protocol library that served as the client connector interface for

MOMspider [39]. At the time, the Web's architecture was described by a set of informal hypertext notes [14], two early introductory papers [12, 13], draft hypertext specifications representing proposed features for the Web (some of which had already been implemented), and the archive of the public www-talk mailing list that was used for informal discussion among the participants in the WWW project worldwide. Each of the specifications were significantly out of date when compared with Web implementations, mostly due to the rapid evolution of the Web after the introduction of the Mosaic graphical browser [NCSA]. Several experimental extensions had been added to HTTP to allow for proxies, but for the most part the protocol assumed a direct connection between the user agent and either an HTTP origin server or a gateway to legacy systems. There was no awareness within the architecture of caching, proxies, or spiders, even though implementations were readily available and running amok. Many other extensions were being proposed for inclusion in the next versions of the protocols.

At the same time, there was growing pressure within the industry to standardize on some version, or versions, of the Web interface protocols. The W3C was formed by Berners-Lee [20] to act as a think-tank for Web architecture and to supply the authoring resources needed to write the Web standards and reference implementations, but the standardization itself was governed by the Internet Engineering Taskforce [www.ietf.org] and its working groups on URI, HTTP, and HTML. Due to my experience developing Web software, I was first chosen to author the specification for Relative URL [40], later teamed with Henrik Frystyk Nielsen to author the HTTP/1.0 specification [19], became the primary architect of HTTP/1.1 [42], and finally authored the revision of the URL specifications to form the standard on URI generic syntax [21].

The first edition of REST was developed between October 1994 and August 1995, primarily as a means for communicating Web concepts as we wrote the HTTP/1.0 specification and the initial HTTP/1.1 proposal. It was iteratively improved over the next five years and applied to various revisions and extensions of the Web protocol standards. REST was originally referred to as the "HTTP object model," but that name would often lead to misinterpretation of it as the implementation model of an HTTP server. The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.

REST is not intended to capture all possible uses of the Web protocol standards. There are applications of HTTP and URI that do not match the application model of a distributed hypermedia system. The important point, however, is that REST does capture all of those aspects of a distributed hypermedia system that are considered central to the behavioral and performance requirements of the Web, such that optimizing behavior within the model will result in optimum behavior within the deployed Web architecture. In other words, REST is optimized for the common case so that the constraints it applies to the Web architecture will also be optimized for the common case.

## 6.2 REST Applied to URI

Uniform Resource Identifiers (URI) are both the simplest element of the Web architecture and the most important. URI have been known by many names: WWW addresses,

Universal Document Identifiers, Universal Resource Identifiers [15], and finally the combination of Uniform Resource Locators (URL) [17] and Names (URN) [124]. Aside from its name, the URI syntax has remained relatively unchanged since 1992. However, the specification of Web addresses also defines the scope and semantics of what we mean by *resource*, which has changed since the early Web architecture. REST was used to define the term resource for the URI standard [21], as well as the overall semantics of the generic interface for manipulating resources via their representations.

### 6.2.1 Redefinition of Resource

The early Web architecture defined URI as document identifiers. Authors were instructed to define identifiers in terms of a document's location on the network. Web protocols could then be used to retrieve that document. However, this definition proved to be unsatisfactory for a number of reasons. First, it suggests that the author is identifying the content transferred, which would imply that the identifier should change whenever the content changes. Second, there exist many addresses that corresponded to a service rather than a document — authors may be intending to direct readers to that service, rather than to any specific result from a prior access of that service. Finally, there exist addresses that do not correspond to a document at some periods of time, such as when the document does not yet exist or when the address is being used solely for naming, rather than locating, information.

The definition of *resource* in REST is based on a simple premise: identifiers should change as infrequently as possible. Because the Web uses embedded identifiers rather than link servers, authors need an identifier that closely matches the semantics they intend by a

hypermedia reference, allowing the reference to remain static even though the result of accessing that reference may change over time. REST accomplishes this by defining a resource to be the semantics of what the author intends to identify, rather than the value corresponding to those semantics at the time the reference is created. It is then left to the author to ensure that the identifier chosen for a reference does indeed identify the intended semantics.

### 6.2.2 Manipulating Shadows

Defining *resource* such that a URI identifies a concept rather than a document leaves us with another question: how does a user access, manipulate, or transfer a concept such that they can get something useful when a hypertext link is selected? REST answers that question by defining the things that are manipulated to be *representations* of the identified resource, rather than the resource itself. An origin server maintains a mapping from resource identifiers to the set of representations corresponding to each resource. A resource is therefore manipulated by transferring representations through the generic interface defined by the resource identifier.

REST's definition of resource derives from the central requirement of the Web: independent authoring of interconnected hypertext across multiple trust domains. Forcing the interface definitions to match the interface requirements causes the protocols to seem vague, but that is only because the interface being manipulated is only an interface and not an implementation. The protocols are specific about the intent of an application action, but the mechanism behind the interface must decide how that intention affects the underlying implementation of the resource mapping to representations.

Information hiding is one of the key software engineering principles that motivates the uniform interface of REST. Because a client is restricted to the manipulation of representations rather than directly accessing the implementation of a resource, the implementation can be constructed in whatever form is desired by the naming authority without impacting the clients that may use its representations. In addition, if multiple representations of the resource exist at the time it is accessed, a content selection algorithm can be used to dynamically select a representation that best fits the capabilities of that client. The disadvantage, of course, is that remote authoring of a resource is not as straightforward as remote authoring of a file.

### 6.2.3 Remote Authoring

The challenge of remote authoring via the Web's uniform interface is due to the separation between the representation that can be retrieved by a client and the mechanism that might be used on the server to store, generate, or retrieve the content of that representation. An individual server may map some part of its namespace to a filesystem, which in turn maps to the equivalent of an inode that can be mapped into a disk location, but those underlying mechanisms provide a means of associating a resource to a set of representations rather than identifying the resource itself. Many different resources could map to the same representation, while other resources may have no representation mapped at all.

In order to author an existing resource, the author must first obtain the specific source resource URI: the set of URI that bind to the handler's underlying representation for the target resource. A resource does not always map to a singular file, but all resources that are not static are derived from some other resources, and by following the derivation tree

an author can eventually find all of the source resources that must be edited in order to modify the representation of a resource. These same principles apply to any form of derived representation, whether it be from content negotiation, scripts, servlets, managed configurations, versioning, etc.

The resource is not the storage object. The resource is not a mechanism that the server uses to handle the storage object. The resource is a conceptual mapping — the server receives the identifier (which identifies the mapping) and applies it to its current mapping implementation (usually a combination of collection-specific deep tree traversal and/or hash tables) to find the currently responsible handler implementation and the handler implementation then selects the appropriate action+response based on the request content. All of these implementation-specific issues are hidden behind the Web interface; their nature cannot be assumed by a client that only has access through the Web interface.

For example, consider what happens when a Web site grows in user base and decides to replace its old Brand X server, based on an XOS platform, with a new Apache server running on FreeBSD. The disk storage hardware is replaced. The operating system is replaced. The HTTP server is replaced. Perhaps even the method of generating responses for all of the content is replaced. However, what doesn't need to change is the Web interface: if designed correctly, the namespace on the new server can mirror that of the old, meaning that from the client's perspective, which only knows about resources and not about how they are implemented, nothing has changed aside from the improved robustness of the site.

### 6.2.4 Binding Semantics to URI

As mentioned above, a resource can have many identifiers. In other words, there may exist two or more different URI that have equivalent semantics when used to access a server. It is also possible to have two URI that result in the same mechanism being used upon access to the server, and yet those URI identify two different resources because they don't mean the same thing.

Semantics are a by-product of the act of assigning resource identifiers and populating those resources with representations. At no time whatsoever do the server or client software need to know or understand the meaning of a URI — they merely act as a conduit through which the creator of a resource (a human naming authority) can associate representations with the semantics identified by the URI. In other words, there are no resources on the server; just mechanisms that supply answers across an abstract interface defined by resources. It may seem odd, but this is the essence of what makes the Web work across so many different implementations.

It is the nature of every engineer to define things in terms of the characteristics of the components that will be used to compose the finished product. The Web doesn't work that way. The Web architecture consists of constraints on the communication model between components, based on the role of each component during an application action. This prevents the components from assuming anything beyond the resource abstraction, thus hiding the actual mechanisms on either side of the abstract interface.

### 6.2.5 REST Mismatches in URI

Like most real-world systems, not all components of the deployed Web architecture obey every constraint present in its architectural design. REST has been used both as a means to define architectural improvements and to identify architectural mismatches. Mismatches occur when, due to ignorance or oversight, a software implementation is deployed that violates the architectural constraints. While mismatches cannot be avoided in general, it is possible to identify them before they become standardized.

Although the URI design matches REST's architectural notion of identifiers, syntax alone is insufficient to force naming authorities to define their own URI according to the resource model. One form of abuse is to include information that identifies the current user within all of the URI referenced by a hypermedia response representation. Such embedded user-ids can be used to maintain session state on the server, track user behavior by logging their actions, or carry user preferences across multiple actions (e.g., Hyper-G's gateways [84]). However, by violating REST's constraints, these systems also cause shared caching to become ineffective, reduce server scalability, and result in undesirable effects when a user shares those references with others.

Another conflict with the resource interface of REST occurs when software attempts to treat the Web as a distributed file system. Since file systems expose the implementation of their information, tools exist to "mirror" that information across to multiple sites as a means of load balancing and redistributing the content closer to users. However, they can do so only because files have a fixed set of semantics (a named sequence of bytes) that can be duplicated easily. In contrast, attempts to mirror the content of a Web server as files will fail because the resource interface does not always match the semantics of a file

system, and because both data and metadata are included within, and significant to, the semantics of a representation. Web server content can be replicated at remote sites, but only by replicating the entire server mechanism and configuration, or by selectively replicating only those resources with representations known to be static (e.g., cache networks contract with Web sites to replicate specific resource representations to the "edges" of the overall Internet in order to reduce latency and distribute load away from the origin server).

## 6.3 REST Applied to HTTP

The Hypertext Transfer Protocol (HTTP) has a special role in the Web architecture as both the primary application-level protocol for communication between Web components and the only protocol designed specifically for the transfer of resource representations. Unlike URI, there were a large number of changes needed in order for HTTP to support the modern Web architecture. The developers of HTTP implementations have been conservative in their adoption of proposed enhancements, and thus extensions needed to be proven and subjected to standards review before they could be deployed. REST was used to identify problems with the existing HTTP implementations, specify an interoperable subset of that protocol as HTTP/1.0 [19], analyze proposed extensions for HTTP/1.1 [42], and provide motivating rationale for deploying HTTP/1.1.

The key problem areas in HTTP that were identified by REST included planning for the deployment of new protocol versions, separating message parsing from HTTP semantics and the underlying transport layer (TCP), distinguishing between authoritative and non-authoritative responses, fine-grained control of caching, and various aspects of

the protocol that failed to be self-descriptive. REST has also been used to model the performance of Web applications based on HTTP and anticipate the impact of such extensions as persistent connections and content negotiation. Finally, REST has been used to limit the scope of standardized HTTP extensions to those that fit within the architectural model, rather than allowing the applications that misuse HTTP to equally influence the standard.

### 6.3.1 Extensibility

One of the major goals of REST is to support the gradual and fragmented deployment of changes within an already deployed architecture. HTTP was modified to support that goal through the introduction of versioning requirements and rules for extending each of the protocol's syntax elements.

#### *6.3.1.1 Protocol Versioning*

HTTP is a family of protocols, distinguished by major and minor version numbers, that share the name primarily because they correspond to the protocol expected when communicating directly with a service based on the "http" URL namespace. A connector must obey the constraints placed on the HTTP-version protocol element included in each message [90].

The HTTP-version of a message represents the protocol capabilities of the sender and the gross-compatibility (major version number) of the message being sent. This allows a client to use a reduced (HTTP/1.0) subset of features in making a normal HTTP/1.1 request, while at the same time indicating to the recipient that it is capable of supporting full HTTP/1.1 communication. In other words, it provides a tentative form of protocol

negotiation on the HTTP scale. Each connection on a request/response chain can operate at its best protocol level in spite of the limitations of some clients or servers that are parts of the chain.

The intention of the protocol is that the server should always respond with the highest minor version of the protocol it understands within the same major version of the client's request message. The restriction is that the server cannot use those optional features of the higher-level protocol which are forbidden to be sent to such an older-version client. There are no required features of a protocol that cannot be used with all other minor versions within that major version, since that would be an incompatible change and thus require a change in the major version. The only features of HTTP that can depend on a minor version number change are those that are interpreted by immediate neighbors in the communication, because HTTP does not require that the entire request/response chain of intermediary components speak the same version.

These rules exist to assist in the deployment of multiple protocol revisions and to prevent the HTTP architects from forgetting that deployment of the protocol is an important aspect of its design. They do so by making it easy to differentiate between compatible changes to the protocol and incompatible changes. Compatible changes are easy to deploy and communication of the differences can be achieved within the protocol stream. Incompatible changes are difficult to deploy because they require some determination of acceptance of the protocol before the protocol stream can commence.

#### *6.3.1.2 Extensible Protocol Elements*

HTTP includes a number of separate namespaces, each of which has differing constraints, but all of which share the requirement of being extensible without bound. Some of the

namespaces are governed by separate Internet standards and shared by multiple protocols (e.g., URI schemes [21], media types [48], MIME header field names [47], charset values, language tags), while others are governed by HTTP, including the namespaces for method names, response status codes, non-MIME header field names, and values within standard HTTP header fields. Since early HTTP did not define a consistent set of rules for how changes within these namespaces could be deployed, this was one of the first problems tackled by the specification effort.

HTTP request semantics are signified by the request method name. Method extension is allowed whenever a standardizable set of semantics can be shared between client, server, and any intermediaries that may be between them. Unfortunately, early HTTP extensions, specifically the HEAD method, made the parsing of an HTTP response message dependent on knowing the semantics of the request method. This led to a deployment contradiction: if a recipient needs to know the semantics of a method before it can be safely forwarded by an intermediary, then all intermediaries must be updated before a new method can be deployed.

This deployment problem was fixed by separating the rules for parsing and forwarding HTTP messages from the semantics associated with new HTTP protocol elements. For example, HEAD is the only method for which the Content-Length header field has a meaning other than signifying the message body length, and no new method can change the message length calculation. GET and HEAD are also the only methods for which conditional request header fields have the semantics of a cache refresh, whereas for all other methods they have the meaning of a precondition.

Likewise, HTTP needed a general rule for interpreting new response status codes, such that new responses could be deployed without significantly harming older clients. We therefore expanded upon the rule that each status code belonged to a class signified by the first digit of its three-digit decimal number: 100-199 indicating that the message contains a provisional information response, 200-299 indicating that the request succeeded, 300-399 indicating that the request needs to be redirected to another resource, 400-499 indicating that the client made an error that should not be repeated, and 500-599 indicating that the server encountered an error, but that the client may get a better response later (or via some other server). If a recipient does not understand the specific semantics of the status code in a given message, then they must treat it in the same way as the x00 code within its class. Like the rule for method names, this extensibility rule places a requirement on the current architecture such that it anticipates future change. Changes can therefore be deployed onto an existing architecture with less fear of adverse component reactions.

#### *6.3.1.3 Upgrade*

The addition of the Upgrade header field in HTTP/1.1 reduces the difficulty of deploying incompatible changes by allowing the client to advertise its willingness for a better protocol while communicating in an older protocol stream. Upgrade was specifically added to support the selective replacement of HTTP/1.x with other, future protocols that might be more efficient for some tasks. Thus, HTTP not only supports internal extensibility, but also complete replacement of itself during an active connection. If the server supports the improved protocol and desires to switch, it simply responds with a 101 status and continues on as if the request were received in that upgraded protocol.

### 6.3.2 Self-descriptive Messages

REST constrains messages between components to be self-descriptive in order to support intermediate processing of interactions. However, there were aspects of early HTTP that failed to be self-descriptive, including the lack of host identification within requests, failure to syntactically distinguish between message control data and representation metadata, failure to differentiate between control data intended only for the immediate connection peer versus metadata intended for all recipients, lack of support for mandatory extensions, and the need for metadata to describe representations with layered encodings.

#### *6.3.2.1 Host*

One of the worst mistakes in the early HTTP design was the decision not to send the complete URI that is the target of a request message, but rather send only those portions that were not used in setting up the connection. The assumption was that a server would know its own naming authority based on the IP address and TCP port of the connection. However, this failed to anticipate that multiple naming authorities might exist on a single server, which became a critical problem as the Web grew at an exponential rate and new domain names (the basis for naming authority within the http URL namespace) far exceeded the availability of new IP addresses.

The solution defined and deployed for both HTTP/1.0 and HTTP/1.1 was to include the target URL's host information within a Host header field of the request message. Deployment of this feature was considered so important that the HTTP/1.1 specification requires servers to reject any HTTP/1.1 request that doesn't include a Host field. As a result, there now exist many large ISP servers that run tens of thousands of name-based virtual host websites on a single IP address.

#### *6.3.2.2 Layered Encodings*

HTTP inherited its syntax for describing representation metadata from the Multipurpose Internet Mail Extensions (MIME) [47]. MIME does not define layered media types, preferring instead to only include the label of the outermost media type within the Content-Type field value. However, this prevents a recipient from determining the nature of an encoded message without decoding the layers. An early HTTP extension worked around this failing by listing the outer encodings separately within the Content-Encoding field and placing the label for the innermost media type in the Content-Type. That was a poor design decision, since it changed the semantics of Content-Type without changing its field name, resulting in confusion whenever older user agents encountered the extension.

A better solution would have been to continue treating Content-Type as the outermost media type, and use a new field to describe the nested types within that type. Unfortunately, the first extension was deployed before its faults were identified.

REST did identify the need for another layer of encodings: those placed on a message by a connector in order to improve its transferability over the network. This new layer, called a transfer-encoding in reference to a similar concept in MIME, allows messages to be encoded for transfer without implying that the representation is encoded by nature. Transfer encodings can be added or removed by transfer agents, for whatever reason, without changing the semantics of the representation.

#### *6.3.2.3 Semantic Independence*

As described above, HTTP message parsing has been separated from its semantics. Message parsing, including finding and globbing together the header fields, occurs entirely separate from the process of parsing the header field contents. In this way,

intermediaries can quickly process and forward HTTP messages, and extensions can be deployed without breaking existing parsers.

#### *6.3.2.4 Transport Independence*

Early HTTP, including most implementations of HTTP/1.0, used the underlying transport protocol as the means for signaling the end of a response message. A server would indicate the end of a response message body by closing the TCP connection. Unfortunately, this created a significant failure condition in the protocol: a client had no means for distinguishing between a completed response and one that was truncated by some erroneous network failure. To solve this, the Content-Length header fields was redefined within HTTP/1.0 to indicate the message body length in bytes, whenever the length is known in advance, and the "chunked" transfer encoding was introduced to HTTP/1.1.

The chunked encoding allows a representation whose size is unknown at the beginning of its generation (when the header fields are sent) to have its boundaries delineated by a series of chunks that can be individually sized before being sent. It also allows metadata to be sent at the end of the message as trailers, enabling the creation of optional metadata at the origin while the message is being generated, without adding to response latency.

#### *6.3.2.5 Size Limits*

A frequent barrier to the flexibility of application-layer protocols is the tendency to over-specify size limits on protocol parameters. Although there always exist some practical limits within implementations of the protocol (e.g., available memory), specifying those limits within the protocol restricts all applications to the same limits, regardless of their

implementation. The result is often a lowest-common-denominator protocol that cannot be extended much beyond the envisioning of its original creator.

There is no limit in the HTTP protocol on the length of URI, the length of header fields, the length of an representation, or the length of any field value that consists of a list of items. Although older Web clients have a well-known problem with URI that consist of more than 255 characters, it is sufficient to note that problem in the HTTP specification rather than require that all servers be so limited. The reason that this does not make for a protocol maximum is that applications within a controlled context (such as an intranet) can avoid those limits by replacing the older components.

Although we did not need to invent artificial limitations, HTTP/1.1 did need to define an appropriate set of response status codes for indicating when a given protocol element is too long for a server to process. Such response codes were added for the conditions of Request-URI too long, header field too long, and body too long. Unfortunately, there is no way for a client to indicate to a server that it may have resource limits, which leads to problems when resource-constrained devices, such as PDAs, attempt to use HTTP without a device-specific intermediary adjusting the communication.

#### *6.3.2.6 Cache Control*

Because REST tries to balance the need for efficient, low-latency behavior with the desire for semantically transparent cache behavior, it is critical that HTTP allow the application to determine the caching requirements rather than hard-code it into the protocol itself. The most important thing for the protocol to do is to fully and accurately describe the data being transferred, so that no application is fooled into thinking it has one thing when it

actually has something else. HTTP/1.1 does this through the addition of the Cache-Control, Age, Etag, and Vary header fields.

#### *6.3.2.7 Content Negotiation*

All resources map a request (consisting of method, identifier, request-header fields, and sometimes a representation) to a response (consisting of a status code, response-header fields, and sometimes a representation). When an HTTP request maps to multiple representations on the server, the server may engage in content negotiation with the client in order to determine which one best meets the client's needs. This is really more of a "content selection" process than negotiation.

Although there were several deployed implementations of content negotiation, it was not included in the specification of HTTP/1.0 because there was no interoperable subset of implementations at the time it was published. This was partly due to a poor implementation within NCSA Mosaic, which would send 1KB of preference information in the header fields on every request, regardless of the negotiability of the resource [125]. Since far less than 0.01% of all URI are negotiable in content, the result was substantially increased request latency for very little gain, which led to later browsers disregarding the negotiation features of HTTP/1.0.

Preemptive (server-driven) negotiation occurs when the server varies the response representation for a particular request method*identifier*status-code combination according to the value of the request header fields, or something external to the normal request parameters above. The client needs to be notified when this occurs, so that a cache can know when it is semantically transparent to use a particular cached response for a future request, and also so that a user agent can supply more detailed preferences than it

might normally send once it knows they are having an effect on the response received. HTTP/1.1 introduced the *Vary* header field for this purpose. Vary simply lists the request header field dimensions under which the response may vary.

In preemptive negotiation, the user agent tells the server what it is capable of accepting. The server is then supposed to select the representation that best matches what the user agent claims to be its capabilities. However, this is a non-tractable problem because it requires not only information on what the UA will accept, but also how well it accepts each feature and to what purpose the user intends to put the representation. For example, a user that wants to view an image on screen might prefer a simple bitmap representation, but the same user with the same browser may prefer a PostScript representation if they intend to send it to a printer instead. It also depends on the user correctly configuring their browser according to their own personal content preferences. In short, a server is rarely able to make effective use of preemptive negotiation, but it was the only form of automated content selection defined by early HTTP.

HTTP/1.1 added the notion of reactive (agent-driven) negotiation. In this case, when a user agent requests a negotiated resource, the server responds with a list of the available representations. The user agent can then choose which one is best according to its own capabilities and purpose. The information about the available representations may be supplied via a separate representation (e.g., a 300 response), inside the response data (e.g., conditional HTML), or as a supplement to the “most likely” response. The latter works best for the Web because an additional interaction only becomes necessary if the user agent decides one of the other variants would be better. Reactive negotiation is simply an

automated reflection of the normal browser model, which means it can take full advantage of all the performance benefits of REST.

Both preemptive and reactive negotiation suffer from the difficulty of communicating the actual characteristics of the representation dimensions (e.g., how to say that a browser supports HTML tables but not the INSERT element). However, reactive negotiation has the distinct advantages of not having to send preferences on every request, having more context information with which to make a decision when faced with alternatives, and not interfering with caches.

A third form of negotiation, transparent negotiation [64], is a license for an intermediary cache to act as an agent, on behalf of other agents, for selecting a better representation and initiating requests to retrieve that representation. The request may be resolved internally by another cache hit, and thus it is possible that no additional network request will be made. In so doing, however, they are performing server-driven negotiation, and must therefore add the appropriate Vary information so that other outbound caches won't be confused.

### 6.3.3 Performance

HTTP/1.1 focused on improving the semantics of communication between components, but there were also some improvements to user-perceived performance, albeit limited by the requirement of syntax compatibility with HTTP/1.0.

#### *6.3.3.1 Persistent Connections*

Although early HTTP's single request/response per connection behavior made for simple implementations, it resulted in inefficient use of the underlying TCP transport due to the

overhead of per-interaction set-up costs and the nature of TCP's slow-start congestion control algorithm [63, 125]. As a result, several extensions were proposed to combine multiple requests and responses within a single connection.

The first proposal was to define a new set of methods for encapsulating multiple requests within a single message (MGET, MHEAD, etc.) and returning the response as a MIME multipart. This was rejected because it violated several of the REST constraints. First, the client would need to know all of the requests it wanted to package before the first request could be written to the network, since a request body must be length-delimited by a content-length field set in the initial request header fields. Second, intermediaries would have to extract each of the messages to determine which ones it could satisfy locally. Finally, it effectively doubles the number of request methods and complicates mechanisms for selectively denying access to certain methods.

Instead, we adopted a form of persistent connections, which uses length-delimited messages in order to send multiple HTTP messages on a single connection [100]. For HTTP/1.0, this was done using the "keep-alive" directive within the Connection header field. Unfortunately, that did not work in general because the header field could be forwarded by intermediaries to other intermediaries that do not understand keep-alive, resulting in a dead-lock condition. HTTP/1.1 eventually settled on making persistent connections the default, thus signaling their presence via the HTTP-version value, and only using the connection-directive "close" to reverse the default.

It is important to note that persistent connections only became possible after HTTP messages were redefined to be self-descriptive and independent of the underlying transport protocol.

#### *6.3.3.2 Write-through Caching*

HTTP does not support write-back caching. An HTTP cache cannot assume that what gets written through it is the same as what would be retrievable from a subsequent request for that resource, and thus it cannot cache a PUT request body and reuse it for a later GET response. There are two reasons for this rule: 1) metadata might be generated behind-the-scenes, and 2) access control on later GET requests cannot be determined from the PUT request. However, since write actions using the Web are extremely rare, the lack of write-back caching does not have a significant impact on performance.

### 6.3.4 REST Mismatches in HTTP

There are several architectural mismatches present within HTTP, some due to 3rd-party extensions that were deployed external to the standards process and others due to the necessity of remaining compatible with deployed HTTP/1.0 components.

#### *6.3.4.1 Differentiating Non-authoritative Responses*

One weakness that still exists in HTTP is that there is no consistent mechanism for differentiating between authoritative responses, which are generated by the origin server in response to the current request, and non-authoritative responses that are obtained from an intermediary or cache without accessing the origin server. The distinction can be important for applications that require authoritative responses, such as the safety-critical information appliances used within the health industry, and for those times when an error response is returned and the client is left wondering whether the error was due to the origin or to some intermediary. Attempts to solve this using additional status codes did not succeed, since the authoritative nature is usually orthogonal to the response status.

HTTP/1.1 did add a mechanism to control cache behavior such that the desire for an authoritative response can be indicated. The 'no-cache' directive on a request message requires any cache to forward the request toward the origin server even if it has a cached copy of what is being requested. This allows a client to refresh a cached copy which is known to be corrupted or stale. However, using this field on a regular basis interferes with the performance benefits of caching. A more general solution would be to require that responses be marked as non-authoritative whenever an action does not result in contacting the origin server. A *Warning* response header field was defined in HTTP/1.1 for this purpose (and others), but it has not been widely implemented in practice.

#### *6.3.4.2 Cookies*

An example of where an inappropriate extension has been made to the protocol to support features that contradict the desired properties of the generic interface is the introduction of site-wide state information in the form of HTTP cookies [73]. Cookie interaction fails to match REST's model of application state, often resulting in confusion for the typical browser application.

An HTTP cookie is opaque data that can be assigned by the origin server to a user agent by including it within a Set-Cookie response header field, with the intention being that the user agent should include the same cookie on all future requests to that server until it is replaced or expires. Such cookies typically contain an array of user-specific configuration choices, or a token to be matched against the server's database on future requests. The problem is that a cookie is defined as being attached to any future requests for a given set of resource identifiers, usually encompassing an entire site, rather than being associated with the particular application state (the set of currently rendered

representations) on the browser. When the browser's history functionality (the "Back" button) is subsequently used to back-up to a view prior to that reflected by the cookie, the browser's application state no longer matches the stored state represented within the cookie. Therefore, the next request sent to the same server will contain a cookie that misrepresents the current application context, leading to confusion on both sides.

Cookies also violate REST because they allow data to be passed without sufficiently identifying its semantics, thus becoming a concern for both security and privacy. The combination of cookies with the Referer [sic] header field makes it possible to track a user as they browse between sites.

As a result, cookie-based applications on the Web will never be reliable. The same functionality should have been accomplished via anonymous authentication and true client-side state. A state mechanism that involves preferences can be more efficiently implemented using judicious use of context-setting URI rather than cookies, where judicious means one URI per state rather than an unbounded number of URI due to the embedding of a user-id. Likewise, the use of cookies to identify a user-specific "shopping basket" within a server-side database could be more efficiently implemented by defining the semantics of shopping items within the hypermedia data formats, allowing the user agent to select and store those items within their own client-side shopping basket, complete with a URI to be used for check-out when the client is ready to purchase.

#### *6.3.4.3 Mandatory Extensions*

HTTP header field names can be extended at will, but only when the information they contain is not required for proper understanding of the message. Mandatory header field extensions require a major protocol revision or a substantial change to method semantics,

such as that proposed in [94]. This is an aspect of the modern Web architecture which does not yet match the self-descriptive messaging constraints of the REST architectural style, primarily because the cost of implementing a mandatory extension framework within the existing HTTP syntax exceeds any clear benefits that we might gain from mandatory extensions. However, it is reasonable to expect that mandatory field name extensions will be supported in the next major revision of HTTP, when the existing constraints on backwards-compatibility of syntax no longer apply.

#### *6.3.4.4 Mixing Metadata*

HTTP is designed to extend the generic connector interface across a network connection. As such, it is intended to match the characteristics of that interface, including the delineation of parameters as control data, metadata, and representation. However, two of the most significant limitations of the HTTP/1.x protocol family are that it fails to syntactically distinguish between representation metadata and message control information (both transmitted as header fields) and does not allow metadata to be effectively layered for message integrity checks.

REST identified these as limitations in the protocol early in the standardization process, anticipating that they would lead to problems in the deployment of other features, such as persistent connections and digest authentication. Workarounds were developed, including adding the Connection header field to identify per-connection control data that is unsafe to be forwarded by intermediaries, as well as an algorithm for the canonical treatment of header field digests [46].

#### *6.3.4.5 MIME Syntax*

HTTP inherited its message syntax from MIME [47] in order to retain commonality with other Internet protocols and reuse many of the standardized fields for describing media types in messages. Unfortunately, MIME and HTTP have very different goals, and the syntax is only designed for MIME's goals.

In MIME, a user agent is sending a bunch of information, which is intended to be treated as a coherent whole, to an unknown recipient with which they never directly interact. MIME assumes that the agent would want to send all that information in one message, since sending multiple messages across Internet mail is less efficient. Thus, MIME syntax is constructed to package messages within a part or multipart in much the way postal carriers wrap packages in extra paper.

In HTTP, packaging different objects within a single message doesn't make any sense other than for secure encapsulation or packaged archives, since it is more efficient to make separate requests for those documents not already cached. Thus, HTTP applications use media types like HTML as containers for references to the "package" — a user agent can then choose what parts of the package to retrieve as separate requests. Although it is possible that HTTP could use a multipart package in which only the non-URI resources were included after the first part, there hasn't been much demand for it.

The problem with MIME syntax is that it assumes the transport is lossy, deliberately corrupting things like line breaks and content lengths. The syntax is therefore verbose and inefficient for any system not based on a lossy transport, which makes it inappropriate for HTTP. Since HTTP/1.1 has the capability to support deployment of incompatible protocols, retaining the MIME syntax won't be necessary for the next major version of

HTTP, even though it will likely continue to use the many standardized protocol elements for representation metadata.

### 6.3.5 Matching Responses to Requests

HTTP messages fail to be self-descriptive when it comes to describing which response belongs with which request. Early HTTP was based on a single request and response per connection, so there was no perceived need for message control data that would tie the response back to the request that invoked it. Therefore, the ordering of requests determines the ordering of responses, which means that HTTP relies on the transport connection to determine the match.

HTTP/1.1, though defined to be independent of the transport protocol, still assumes that communication takes place on a synchronous transport. It could easily be extended to work on an asynchronous transport, such as e-mail, through the addition of a request identifier. Such an extension would be useful for agents in a broadcast or multicast situation, where responses might be received on a channel different from that of the request. Also, in a situation where many requests are pending, it would allow the server to choose the order in which responses are transferred, such that smaller or more significant responses are sent first.

## 6.4 Technology Transfer

Although REST had its most direct influence over the authoring of Web standards, validation of its use as an architectural design model came through the deployment of the standards in the form of commercial-grade implementations.

### 6.4.1 Deployment experience with libwww-perl

My involvement in the definition of Web standards began with development of the maintenance robot MOMspider [39] and its associated protocol library, libwww-perl. Modeled after the original libwww developed by Tim Berners-Lee and the WWW project at CERN, libwww-perl provided a uniform interface for making Web requests and interpreting Web responses for client applications written in the Perl language [134]. It was the first Web protocol library to be developed independent of the original CERN project, reflecting a more modern interpretation of the Web interface than was present in older code bases. This interface became the basis for designing REST.

libwww-perl consisted of a single request interface that used Perl's self-evaluating code features to dynamically load the appropriate transport protocol package based on the scheme of the requested URI. For example, when asked to make a "GET" request on the URL <http://www.ebuilt.com/>, libwww-perl would extract the scheme from the URL ("http") and use it to construct a call to *wwwhttp'request()*, using an interface that was common to all types of resources (HTTP, FTP, WAIS, local files, etc.). In order to achieve this generic interface, the library treated all calls in much the same way as an HTTP proxy. It provided an interface using Perl data structures that had the same semantics as an HTTP request, regardless of the type of resource.

libwww-perl demonstrated the benefits of a generic interface. Within a year of its initial release, over 600 independent software developers were using the library for their own client tools, ranging from command-line download scripts to full-blown browsers. It is currently the basis for most Web system administration tools.

### 6.4.2 Deployment experience with Apache

As the specification effort for HTTP began to take the form of complete specifications, we needed server software that could both effectively demonstrate the proposed standard protocol and serve as a test-bed for worthwhile extensions. At the time, the most popular HTTP server (httpd) was the public domain software developed by Rob McCool at the National Center for Supercomputing Applications, University of Illinois, Urbana-Champaign (NCSA). However, development had stalled after Rob left NCSA in mid-1994, and many webmasters had developed their own extensions and bug fixes that were in need of a common distribution. A group of us created a mailing list for the purpose of coordinating our changes as "patches" to the original source. In the process, we created the Apache HTTP Server Project [89].

The Apache project is a collaborative software development effort aimed at creating a robust, commercial-grade, full-featured, open-source software implementation of an HTTP server. The project is jointly managed by a group of volunteers located around the world, using the Internet and the Web to communicate, plan, and develop the server and its related documentation. These volunteers are known as the Apache Group. More recently, the group formed the nonprofit Apache Software Foundation to act as a legal and financial umbrella organization for supporting continued development of the Apache open source projects.

Apache became known for both its robust behavior in response to the varied demands of an Internet service and for its rigorous implementation of the HTTP protocol standards. I served as the "protocol cop" within the Apache Group, writing code for the core HTTP parsing functions, supporting the efforts of others by explaining the standards, and acting

as an advocate for the Apache developers' views of "the right way to implement HTTP" within the standards forums. Many of the lessons described in this chapter were learned as a result of creating and testing various implementations of HTTP within the Apache project, and subjecting the theories behind the protocol to the Apache Group's critical review.

Apache httpd is widely regarded as one of the most successful software projects, and one of the first open-source software products to dominate a market in which there exists significant commercial competition. The July 2000 Netcraft survey of public Internet websites found over 20 million sites based on the Apache software, representing over 65% of all sites surveyed [http://www.netcraft.com/survey/]. Apache was the first major server to support the HTTP/1.1 protocol and is generally considered the reference implementation against which all client software is tested. The Apache Group received the 1999 ACM Software System Award as recognition of our impact on the standards for the Web architecture.

### 6.4.3 Deployment of URI and HTTP/1.1-compliant Software

In addition to Apache, many other projects, both commercial and open-source in nature, have adopted and deployed software products based on the protocols of the modern Web architecture. Though it may be only a coincidence, Microsoft Internet Explorer surpassed Netscape Navigator in the Web browser market share shortly after they were the first major browser to implement the HTTP/1.1 client standard. In addition, many of the individual HTTP extensions that were defined during the standardization process, such as the Host header field, have now reached universal deployment.

The REST architectural style succeeded in guiding the design and deployment of the modern Web architecture. To date, there have been no significant problems caused by the introduction of the new standards, even though they have been subject to gradual and fragmented deployment alongside legacy Web applications. Furthermore, the new standards have had a positive effect on the robustness of the Web and enabled new methods for improving user-perceived performance through caching hierarchies and content distribution networks.

## 6.5 Architectural Lessons

There are a number of general architectural lessons to be learned from the modern Web architecture and the problems identified by REST.

### 6.5.1 Advantages of a Network-based API

What distinguishes the modern Web from other middleware [22] is the way in which it uses HTTP as a network-based Application Programming Interface (API). This was not always the case. The early Web design made use of a library package, CERN libwww, as the single implementation library for all clients and servers. CERN libwww provided a library-based API for building interoperable Web components.

A library-based API provides a set of code entry points and associated symbol/ parameter sets so that a programmer can use someone else's code to do the dirty work of maintaining the actual interface between like systems, provided that the programmer obeys the architectural and language restrictions that come with that code. The assumption is that all sides of the communication use the same API, and therefore the internals of the interface are only important to the API developer and not the application developer.

The single library approach ended in 1993 because it did not match the social dynamics of the organizations involved in developing the Web. When the team at NCSA increased the pace of Web development with a much larger development team than had ever been present at CERN, the libwww source was “forked” (split into separately maintained code bases) so that the folks at NCSA would not have to wait for CERN to catch-up with their improvements. At the same time, independent developers such as myself began developing protocol libraries for languages and platforms not yet supported by the CERN code. The design of the Web had to shift from the development of a reference protocol library to the development of a network-based API, extending the desired semantics of the Web across multiple platforms and implementations.

A network-based API is an on-the-wire syntax, with defined semantics, for application interactions. A network-based API does not place any restrictions on the application code aside from the need to read/write to the network, but does place restrictions on the set of semantics that can be effectively communicated across the interface. On the plus side, performance is only bounded by the protocol design and not by any particular implementation of that design.

A library-based API does a lot more for the programmer, but in doing so creates a great deal more complexity and baggage than is needed by any one system, is less portable in a heterogeneous network, and always results in genericity being preferred over performance. As a side-effect, it also leads to lazy development (blaming the API code for everything) and failure to account for non-cooperative behavior by other parties in the communication.

However, it is important to keep in mind that there are various layers involved in any architecture, including that of the modern Web. Systems like the Web use one library API (sockets) in order to access several network-based APIs (e.g., HTTP and FTP), but the socket API itself is below the application-layer. Likewise, libwww is an interesting cross-breed in that it has evolved into a library-based API for accessing a network-based API, and thus provides reusable code without assuming other communicating applications are using libwww as well.

This is in contrast to middleware like CORBA [97]. Since CORBA only allows communication via an ORB, its transfer protocol, IIOP, assumes too much about what the parties are communicating. HTTP request messages include standardized application semantics, whereas IIOP messages do not. The “Request” token in IIOP only supplies directionality so that the ORB can route it according to whether the ORB itself is supposed to reply (e.g., “LocateRequest”) or if it will be interpreted by an object. The semantics are expressed by the combination of an object key and operation, which are object-specific rather than standardized across all objects.

An independent developer can generate the same bits as an IIOP request without using the same ORB, but the bits themselves are defined by the CORBA API and its Interface Definition Language (IDL). They need a UUID generated by an IDL compiler, a structured binary content that mirrors that IDL operation's signature, and the definition of the reply data type(s) according to the IDL specification. The semantics are thus not defined by the network interface (IIOP), but by the object's IDL spec. Whether this is a good thing or not depends on the application — for distributed objects it is a necessity, for the Web it isn't.

Why is this important? Because it differentiates a system where network intermediaries can be effective agents from a system where they can be, at most, routers.

This kind of difference is also seen in the interpretation of a message as a unit or as a stream. HTTP allows the recipient or the sender to decide that on their own. CORBA IDL doesn't even allow streams (yet), but even when it does get extended to support streams, both sides of the communication will be tied to the same API, rather than being free to use whatever is most appropriate for their type of application.

### 6.5.2 HTTP is not RPC

People often mistakenly refer to HTTP as a remote procedure call (RPC) [23] mechanism simply because it involves requests and responses. What distinguishes RPC from other forms of network-based application communication is the notion of invoking a procedure on the remote machine, wherein the protocol identifies the procedure and passes it a fixed set of parameters, and then waits for the answer to be supplied within a return message using the same interface. Remote method invocation (RMI) is similar, except that the procedure is identified as an {object, method} tuple rather than a service procedure. Brokered RMI adds name service indirection and a few other tricks, but the interface is basically the same.

What distinguishes HTTP from RPC isn't the syntax. It isn't even the different characteristics gained from using a stream as a parameter, though that helps to explain why existing RPC mechanisms were not usable for the Web. What makes HTTP significantly different from RPC is that the requests are directed to resources using a generic interface with standard semantics that can be interpreted by intermediaries almost

as well as by the machines that originate services. The result is an application that allows for layers of transformation and indirection that are independent of the information origin, which is very useful for an Internet-scale, multi-organization, anarchically scalable information system. RPC mechanisms, in contrast, are defined in terms of language APIs, not network-based applications.

### 6.5.3 HTTP is not a Transport Protocol

HTTP is not designed to be a transport protocol. It is a transfer protocol in which the messages reflect the semantics of the Web architecture by performing actions on resources through the transfer and manipulation of representations of those resources. It is possible to achieve a wide range of functionality using this very simple interface, but following the interface is required in order for HTTP semantics to remain visible to intermediaries.

That is why HTTP goes through firewalls. Most of the recently proposed extensions to HTTP, aside from WebDAV [60], have merely used HTTP as a way to move other application protocols through a firewall, which is a fundamentally misguided idea. Not only does it defeat the purpose of having a firewall, but it won't work for the long term because firewall vendors will simply have to perform additional protocol filtering. It therefore makes no sense to do those extensions on top of HTTP, since the only thing HTTP accomplishes in that situation is to add overhead from a legacy syntax. A true application of HTTP maps the protocol user's actions to something that can be expressed using HTTP semantics, thus creating a network-based API to services which can be understood by agents and intermediaries without any knowledge of the application.

### 6.5.4 Design of Media Types

One aspect of REST that is unusual for an architectural style is the degree to which it influences the definition of data elements within the Web architecture.

#### *6.5.4.1 Application State in a Network-based System*

REST defines a model of expected application behavior which supports simple and robust applications that are largely immune from the partial failure conditions that beset most network-based applications. However, that doesn't stop application developers from introducing features which violate the model. The most frequent violations are in regard to the constraints on application state and stateless interaction.

Architectural mismatches due to misplaced application state are not limited to HTTP cookies. The introduction of "frames" to the Hypertext Markup Language (HTML) caused similar confusion. Frames allow a browser window to be partitioned into subwindows, each with its own navigational state. Link selections within a subwindow are indistinguishable from normal transitions, but the resulting response representation is rendered within the subwindow instead of the full browser application workspace. This is fine provided that no link exits the realm of information that is intended for subwindow treatment, but when it does occur the user finds themself viewing one application wedged within the subcontext of another application.

For both frames and cookies, the failure was in providing indirect application state that could not be managed or interpreted by the user agent. A design that placed this information within a primary representation, thereby informing the user agent on how to manage the hypermedia workspace for a specified realm of resources, could have

accomplished the same tasks without violating the REST constraints, while leading to a better user interface and less interference with caching.

#### *6.5.4.2 Incremental Processing*

By including latency reduction as an architectural goal, REST can differentiate media types (the data format of representations) according to their user-perceived performance. Size, structure, and capacity for incremental rendering all have an impact on the latency encountered transferring, rendering, and manipulating representation media types, and thus can significantly impact system performance.

HTML [18] is an example of a media type that, for the most part, has good latency characteristics. Information within early HTML could be rendered as it was received, because all of the rendering information was available early — within the standardized definitions of the small set of mark-up tags that made up HTML. However, there are aspects of HTML that were not designed well for latency. Examples include: placement of embedded metadata within the HEAD of a document, resulting in optional information needing to be transferred and processed before the rendering engine can read the parts that display something useful to the user [93]; embedded images without rendering size hints, requiring that the first few bytes of the image (the part that contains the layout size) be received before the rest of the surrounding HTML can be displayed; dynamically sized table columns, requiring that the renderer read and determine sizes for the entire table before it can start displaying the top; and, lazy rules regarding the parsing of malformed mark-up syntax, often requiring that the rendering engine parse through an entire file before it can determine that one key mark-up character is missing.

#### *6.5.4.3 Java versus JavaScript*

REST can also be used to gain insight into why some media types have had greater adoption within the Web architecture than others, even when the balance of developer opinion is not in their favor. The case of Java applets versus JavaScript is one example.

Java™ [45] is a popular programming language that was originally developed for applications within television set-top boxes, but first gained notoriety when it was introduced to the Web as a means for implementing *code-on-demand* functionality. Although the language received a tremendous amount of press support from its owner, Sun Microsystems, Inc., and rave reviews from software developers seeking an alternative to the C++ language, it has failed to be widely adopted by application developers for code-on-demand within the Web.

Shortly after Java's introduction, developers at Netscape Communications Corporation created a separate language for embedded code-on-demand, originally called LiveScript, but later changed to the name JavaScript for marketing reasons (the two languages have relatively little in common other than that) [44]. Although initially derided for being embedded with HTML and yet not compatible with proper HTML syntax, JavaScript usage has steadily increased ever since its introduction.

The question is: why is JavaScript more successful on the Web than Java? It certainly isn't because of its technical quality as a language, since both its syntax and execution environment are considered poor when compared to Java. It also isn't because of marketing: Sun far outspent Netscape in that regard, and continues to do so. It isn't because of any intrinsic characteristics of the languages either, since Java has been more successful than JavaScript within all other programming areas (stand-alone applications,

servlets, etc.). In order to better understand the reasons for this discrepancy, we need to evaluate Java in terms of its characteristics as a representation media type within REST.

JavaScript better fits the deployment model of Web technology. It has a much lower entry-barrier, both in terms of its overall complexity as a language and the amount of initial effort required by a novice programmer to put together their first piece of working code. JavaScript also has less impact on the visibility of interactions. Independent organizations can read, verify, and copy the JavaScript source code in the same way that they could copy HTML. Java, in contrast, is downloaded as binary packaged archives — the user is therefore left to trust the security restrictions within the Java execution environment. Likewise, Java has many more features that are considered questionable to allow within a secure environment, including the ability to send RMI requests back to the origin server. RMI does not support visibility for intermediaries.

Perhaps the most important distinction between the two, however, is that JavaScript causes less user-perceived latency. JavaScript is usually downloaded as part of the primary representation, whereas Java applets require a separate request. Java code, once converted to the byte code format, is much larger than typical JavaScript. Finally, whereas JavaScript can be executed while the rest of the HTML page is downloading, Java requires that the complete package of class files be downloaded and installed before the application can begin. Java, therefore, does not support incremental rendering.

Once the characteristics of the languages are laid out along the same lines as the rationale behind REST's constraints, it becomes much easier to evaluate the technologies in terms of their behavior within the modern Web architecture.

## 6.6 Summary

This chapter described the experiences and lessons learned from applying REST while authoring the Internet standards for the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI). These two specifications define the generic interface used by all component interactions on the Web. In addition, I have described the experiences and lessons learned from the deployment of these technologies in the form of the libwww-perl client library, the Apache HTTP Server Project, and other implementations of the protocol standards.

# CONCLUSIONS

> *Each one of us has, somewhere in his heart, the dream to make a living world, a universe. Those of us who have been trained as architects have this desire perhaps at the very center of our lives: that one day, somewhere, somehow, we shall build one building which is wonderful, beautiful, breathtaking, a place where people can walk and dream for centuries.*
> — Christopher Alexander [3]

At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [19] and design the extensions for the new standards of HTTP/1.1 [42] and Uniform Resource Identifiers (URI) [21], I recognized the need for a model of how the World Wide Web *should* work. This idealized model of the interactions within an overall Web application, referred to as the Representational State Transfer (REST) architectural style, became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the preexisting architecture could be identified and extensions validated prior to deployment.

REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system.

The following contributions to the field of Information and Computer Science have been made as part of this dissertation:

- a framework for understanding software architecture through architectural styles, including a consistent set of terminology for describing software architecture;
- a classification of architectural styles for network-based application software by the architectural properties they would induce when applied to the architecture for a distributed hypermedia system;
- REST, a novel architectural style for distributed hypermedia systems; and,
- application and evaluation of the REST architectural style in the design and deployment of the architecture for the modern World Wide Web.

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include access to other styles of interaction, the central focus of its protocol and performance concerns is distributed hypermedia. REST elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction. Areas for improvement of the Web architecture can be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can be replaced with more efficient forms without changing the architecture capabilities. Likewise, proposed extensions can be compared to REST to see if they fit within the architecture; if not, it is more efficient to redirect that functionality to a system running in parallel with a more applicable architectural style.

In an ideal world, the implementation of a software system would exactly match its design. Some features of the modern Web architecture do correspond exactly to their design criteria in REST, such as the use of URI [21] as resource identifiers and the use of Internet media types [48] to identify representation data formats. However, there are also some aspects of the modern Web protocols that exist in spite of the architectural design,

due to legacy experiments that failed (but must be retained for backwards compatibility) and extensions deployed by developers unaware of the architectural style. REST provides a model not only for the development and evaluation of new features, but also for the identification and understanding of broken features.

The World Wide Web is arguably the world's largest distributed application. Understanding the key architectural principles underlying the Web can help explain its technical success and may lead to improvements in other distributed applications, particularly those that are amenable to the same or similar methods of interaction. REST contributes both the rationale behind the modern Web's software architecture and a significant lesson in how software engineering principles can be systematically applied in the design and evaluation of a real software system.

For network-based applications, system performance is dominated by network communication. For a distributed hypermedia system, component interactions consist of large-grain data transfers rather than computation-intensive tasks. The REST style was developed in response to those needs. Its focus upon the generic connector interface of resources and representations has enabled intermediate processing, caching, and substitutability of components, which in turn has allowed Web-based applications to scale from 100,000 requests/day in 1994 to 600,000,000 requests/day in 1999.

The REST architectural style has been validated through six years of development of the HTTP/1.0 [19] and HTTP/1.1 [42] standards, elaboration of the URI [21] and relative URL [40] standards, and successful deployment of several dozen independently developed, commercial-grade software systems within the modern Web architecture. It

has served as both a model for design guidance and as an acid test for architectural extensions to the Web protocols.

Future work will focus on extending the architectural guidance toward the development of a replacement for the HTTP/1.x protocol family, using a more efficient tokenized syntax, but without losing the desirable properties identified by REST. The needs of wireless devices, which have many characteristics in common with the principles behind REST, will motivate further enhancements for application-level protocol design and architectures involving active intermediaries. There has also been some interest in extending REST to consider variable request priorities, differentiated quality-of-service, and representations consisting of continuous data streams, such as those generated by broadcast audio and video sources.

# REFERENCES


1. G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4), Oct. 1995, pp. 319–364. A shorter version also appeared as: Using style to understand descriptions of software architecture. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'93)*, Los Angeles, CA, Dec. 1993, pp. 9–20.

2. Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

3. C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

4. C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

5. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997. A shorter version also appeared as: Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 71–80. Also as: Beyond Definition/Use: Architectural Interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, Portland, Oregon, *SIGPLAN Notices*, 29(8), Aug. 1994.

6. G. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1), Mar. 1991, pp. 49–90.

7. F. Anklesaria, et al. The Internet Gopher protocol (a distributed document search and retrieval protocol). *Internet RFC 1436*, Mar. 1993.

8. D. J. Barrett, L. A. Clarke, P. L. Tarr, A. E. Wise. A framework for event-based software integration. *ACM Transactions on Software Engineering and Methodology*, 5(4), Oct. 1996, pp. 378–421.

9. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, Reading, Mass., 1998.

10. D. Batory, L. Coglianese, S. Shafer, and W. Tracz. The ADAGE avionics reference architecture. In *Proceedings of AIAA Computing in Aerospace 10*, San Antonio, 1995.

11. T. Berners-Lee, R. Cailliau, and J.-F. Groff. World Wide Web. Flyer distributed at the *3rd Joint European Networking Conference*, Innsbruck, Austria, May 1992.

12. T. Berners-Lee, R. Cailliau, J.-F. Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy*, 2(1), Meckler Publishing, Westport, CT, Spring 1992, pp. 52–58.

13. T. Berners-Lee and R. Cailliau. World-Wide Web. In *Proceedings of Computing in High Energy Physics 92*, Annecy, France, 23–27 Sep. 1992.

14. T. Berners-Lee, R. Cailliau, C. Barker, and J.-F. Groff. W3 Project: Assorted design notes. Published on the Web, Nov. 1992. Archived at <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/WorkingNotes/Overview.html>, Sep. 2000.

15. T. Berners-Lee. Universal Resource Identifiers in WWW. *Internet RFC 1630*, June 1994.

16. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8), Aug. 1994, pp. 76–82.

17. T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). *Internet RFC 1738*, Dec. 1994.

18. T. Berners-Lee and D. Connolly. Hypertext Markup Language — 2.0. *Internet RFC 1866*, Nov. 1995.

19. T. Berners-Lee, R. T. Fielding, and H. F. Nielsen. Hypertext Transfer Protocol — HTTP/1.0. *Internet RFC 1945*, May 1996.

20. T. Berners-Lee. WWW: Past, present, and future. *IEEE Computer*, 29(10), Oct. 1996, pp. 69–77.

21. T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. *Internet RFC 2396*, Aug. 1998.

22. P. Bernstein. Middleware: A model for distributed systems services. *Communications of the ACM*, Feb. 1996, pp. 86–98.

23. A. D. Birrell and B. J. Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2, Jan. 1984, pp. 39–59.

24. M. Boasson. The artistry of software architecture. *IEEE Software*, 12(6), Nov. 1995, pp. 13–16.

25. G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, 12(2), Feb. 1986, pp. 211–221.

26. C. Brooks, M. S. Mazer, S. Meeks, and J. Miller. Application-specific proxy servers as HTTP stream transducers. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, Dec. 1995, pp. 539–548.

27. F. Buschmann and R. Meunier. A system of patterns. Coplien and Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 325–343.

28. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture: A system of patterns*. John Wiley & Sons Ltd., England, 1996.

29. M. R. Cagan. The HP SoftBench Environment: An architecture for a new qeneration of software tools. *Hewlett-Packard Journal*, 41(3), June 1990, pp. 36–47.

30. J. R. Cameron. An overview of JSD. *IEEE Transactions on Software Engineering*, 12(2), Feb. 1986, pp. 222–240.

31. R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), Mar. 1991, pp. 91–124.

32. D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM'90 Symposium*, Philadelphia, PA, Sep. 1990, pp. 200–208.

33. J. O. Coplien and D. C. Schmidt, ed. *Pattern Languages of Program Design*. Addison-Wesley, Reading, Mass., 1995.

34. J. O. Coplien. Idioms and Patterns as Architectural Literature. *IEEE Software*, 14(1), Jan. 1997, pp. 36–42.

35. E. M. Dashofy, N. Medvidovic, R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 3–12.

36. F. Davis, et. al. *WAIS Interface Protocol Prototype Functional Specification (v.1.5)*. Thinking Machines Corporation, April 1990.

37. F. DeRemer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2), June 1976, pp. 80–86.

38. E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 13–22.

39. R. T. Fielding. Maintaining distributed hypertext infostructures: Welcome to MOMspider's web. *Computer Networks and ISDN Systems*, 27(2), Nov. 1994, pp. 193–204.

40. R. T. Fielding. Relative Uniform Resource Locators. *Internet RFC 1808*, June 1995.

41. R. T. Fielding, E. J. Whitehead, Jr., K. M. Anderson, G. Bolcer, P. Oreizy, and R. N. Taylor. Web-based development of complex information products. *Communications of the ACM*, 41(8), Aug. 1998, pp. 84–92.

42. R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. *Internet RFC 2616*, June 1999. [Obsoletes RFC 2068, Jan. 1997.]

43. R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 407–416.

44. D. Flanagan. *JavaScript: The Definitive Guide, 3rd edition*. O'Reilly & Associates, Sebastopol, CA, 1998.

45. D. Flanagan. *Java™ in a Nutshell, 3rd edition*. O'Reilly & Associates, Sebastopol, CA, 1999.

46. J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, E. Sink, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. *Internet RFC 2617*, June 1999.

47. N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. *Internet RFC 2045*, Nov. 1996.

48. N. Freed, J. Klensin, and J. Postel. Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. *Internet RFC 2048*, Nov. 1996.

49. M. Fridrich and W. Older. Helix: The architecture of the XMS distributed file system. *IEEE Software*, 2, May 1985, pp. 21–29.

50. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5), May 1998, pp. 342–361.

51. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Reading, Mass., 1995.

52. D. Garlan and E. Ilias. Low-cost, adaptable tool integration policies for integrated environments. In *Proceedings of the ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, Dec. 1990, pp. 1–10.

53. D. Garlan and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), *Advances in Software Engineering & Knowledge Engineering*, vol. II, World Scientific Pub Co., Singapore, 1993, pp. 1–39.

54. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'94)*, New Orleans, Dec. 1994, pp. 175–188.

55. D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 269–274.

56. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch, or, Why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995. Also appears as: Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6), Nov. 1995, pp. 17–26.

57. D. Garlan, R. Monroe, and D. Wile. ACME: An architecture description language. In *Proceedings of CASCON'97*, Nov. 1997.

58. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

59. S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2), Nov. 1994, pp. 165–173.

60. Y. Goland, E. J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring — WEBDAV. *Internet RFC 2518*, Feb. 1999.

61. K. Grønbaek and R. H. Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM*, 37(2), Feb. 1994, pp. 41–49.

62. B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 288–301.

63. J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5), Oct. 1997, pp. 616–630.

64. K. Holtman and A. Mutz. Transparent content negotiation in HTTP. *Internet RFC 2295*, Mar. 1998.

65. P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 373–386.

66. ISO/IEC JTC1/SC21/WG7. *Reference Model of Open Distributed Processing*. ITU-T X.901: ISO/IEC 10746-1, 07 June 1995.

67. M. Jackson. Problems, methods, and specialization. *IEEE Software*, 11(6), [condensed from Software Engineering Journal], Nov. 1994. pp. 57–62.

68. R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994, pp. 81–90.

69. R. Kazman, M. Barbacci, M. Klein, S. J. Carrière, and S. G. Woods. Experience with performing architecture tradeoff analysis. In *Proceedings of the 1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 54–63.

70. N. L. Kerth and W. Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, 14(1), Jan. 1997, pp. 53–59.

71. R. Khare and S. Lawrence. Upgrading to TLS within HTTP/1.1. *Internet RFC 2817*, May 2000.

72. G. E. Krasner and S. T. Pope. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3), Aug.–Sep. 1988, pp. 26–49.

73. D. Kristol and L. Montulli. HTTP State Management Mechanism. *Internet RFC 2109*, Feb. 1997.

74. P. B. Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6), Nov. 1995, pp. 42–50.

75. D. Le Métayer. Describing software architectural styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), July 1998, pp. 521–533.

76. W. C. Loerke. On Style in Architecture. F. Wilson, *Architecture: Fundamental Issues*, Van Nostrand Reinhold, New York, 1990, pp. 203–218.

77. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 336–355.

78. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9), Sep. 1995, pp. 717–734.

79. A. Luotonen and K. Altis. World-Wide Web proxies. *Computer Networks and ISDN Systems*, 27(2), Nov. 1994, pp. 147–154.

80. P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, Orlando, Florida, Oct. 1987, pp. 147–155.

81. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, Sep. 1995, pp. 137–153.

82. J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96)*, San Francisco, Oct. 1996, pp. 3–14.

83. F. Manola. Technologies for a Web object model. IEEE Internet Computing, 3(1), Jan.–Feb. 1999, pp. 38–47.

84. H. Maurer. *HyperWave: The Next-Generation Web Solution*. Addison-Wesley, Harlow, England, 1996.

85. M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage interoperability in distributed systems: Experience Report. In *Proceedings 18th International Conference on Software Engineering*, Berlin, Germany, Mar. 1996.

86. N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, Sep. 1997, pp. 60–76.

87. N. Medvidovic. *Architecture-based Specification-time Software Evolution*. Ph.D. Dissertation, University of California, Irvine, Dec. 1998.

88. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the*

*1999 International Conference on Software Engineering*, Los Angeles, May 16–22, 1999, pp. 44–53.

89. A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: The Apache server. In *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000, pp. 263–272.

90. J. Mogul, R. Fielding, J. Gettys, and H. Frystyk. Use and Interpretation of HTTP Version Numbers. *Internet RFC 2145*, May 1997.

91. R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1), Jan. 1997, pp. 43–52.

92. M. Moriconi, X. Qian, and R. A. Riemenscheider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 356–372.

93. H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network Performance Effects of HTTP/1.1, CSS1, and PNG. *Proceedings of ACM SIGCOMM '97*, Cannes, France, Sep. 1997.

94. H. F. Nielsen, P. Leach, and S. Lawrence. HTTP extension framework, *Internet RFC 2774*, Feb. 2000.

95. H. Penny Nii. Blackboard systems. *AI Magazine*, 7(3):38–53 and 7(4):82–107, 1986.

96. Object Management Group. *Object Management Architecture Guide, Rev. 3.0*. Soley & Stone (eds.), New York: J. Wiley, 3rd ed., 1995.

97. Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA 2.1)*. <http://www.omg.org/>, Aug. 1997.

98. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, Kyoto, Japan, Apr. 1998.

99. P. Oreizy. Decentralized software evolution. Unpublished manuscript (Phase II Survey Paper), Dec. 1998.

100. V. N. Padmanabhan and J. C. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 28, Dec. 1995, pp. 25–35.

101. D. L. Parnas. Information distribution aspects of design methodology. In Proceedings of IFIP Congress 71, Ljubljana, Aug. 1971, pp. 339–344.

102. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), Dec. 1972, pp. 1053–1058.

103. D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(3), Mar. 1979.

104. D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3), 1985, pp. 259–266.

105. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), Oct. 1992, pp. 40–52.

106. J. Postel and J. Reynolds. TELNET Protocol Specification. *Internet STD 8, RFC 854*, May 1983.

107. J. Postel and J. Reynolds. File Transfer Protocol. *Internet STD 9, RFC 959*, Oct. 1985.

108. D. Pountain and C. Szyperski. Extensible software systems. *Byte*, May 1994, pp. 57–62.

109. R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4), Nov. 1986, pp. 307–334.

110. J. M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1), Jan. 1994, pp. 151–174.

111. M. Python. The Architects Sketch. *Monty Python's Flying Circus TV Show, Episode 17*, Sep. 1970. Transcript at <http://www.stone-dead.asn.au/sketches/architec.htm>.

112. J. Rasure and M. Young. Open environment for image processing and software development. In *Proceedings of the 1992 SPIE/IS&T Symposium on Electronic Imaging*, Vol. 1659, Feb. 1992.

113. S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4), July 1990, pp. 57–67.

114. D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, Sep. 1997, pp. 344–360.

115. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Usenix Conference*, June 1985, pp. 119–130.

116. M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, Cambridge, MA, May 1986, pp. 198–204.

117. M. Shaw. Toward higher-level abstractions for software systems. *Data & Knowledge Engineering*, 5, 1990, pp. 119–128.

118. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 314–335.

119. M. Shaw. Comparing architectural design styles. *IEEE Software*, 12(6), Nov. 1995, pp. 27–41.

120. M. Shaw. Some patterns for software architecture. Vlissides, Coplien & Kerth (eds.), *Pattern Languages of Program Design, Vol. 2*, Addison-Wesley, 1996, pp. 255–269.

121. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

122. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, Washington, D.C., Aug. 1997, pp. 6–13.

123. A. Sinha. Client-server computing. *Communications of the ACM*, 35(7), July 1992, pp. 77–98.

124. K. Sollins and L. Masinter. Functional requirements for Uniform Resource Names. *Internet RFC 1737*, Dec. 1994.

125. S. E. Spero. Analysis of HTTP performance problems. Published on the Web, <http://metalab.unc.edu/mdma-release/http-prob.html>, 1994.

126. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3), July 1992, pp. 229–268.

127. A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4), Dec. 1985, pp. 419–470.

128. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6), June 1996, pp. 390–406.

129. W. Tephenhart and J. J. Cusick. A unified object topology. *IEEE Software*, 14(1), Jan. 1997, pp. 31–35.

130. W. Tracz. DSSA (domain-specific software architecture) pedagogical example. *Software Engineering Notes*, 20(3), July 1995, pp. 49–62.

131. A. Umar. *Object-Oriented Client/Server Internet Environments*. Prentice Hall PTR, 1997.

132. S. Vestal. MetaH programmer's manual, version 1.09. *Technical Report*, Honeywell Technology Center, Apr. 1996.

133. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. *Technical Report SMLI TR-94-29*, Sun Microsystems Laboratories, Inc., Nov. 1994.

134. L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl, 2nd ed*. O'Reilly & Associates, 1996.

135. E. J. Whitehead, Jr., R. T. Fielding, and K. M. Anderson. Fusing WWW and link server technology: One approach. In *Proceedings of the 2nd Workshop on Open Hypermedia Systems, Hypertext'96*, Washington, DC, Mar. 1996, pp. 81–86.

136. A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems (USITS)*, Oct. 1999.

137. W. Zimmer. Relationships between design patterns. Coplien and Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995, pp. 345–364.

138. H. Zimmerman. OSI reference model — The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28, Apr. 1980, pp. 425–432.