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

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.

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.