Principled Design of the Modern Web Architecture

ROY T. FIELDING
Day Software
and
RICHARD N. TAYLOR
University of California, Irvine

The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an Internet-scale distributed hypermedia application. The modern Web architecture emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. In this article we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture and used to guide our redesign and definition of the Hypertext Transfer Protocol and Uniform Resource Identifiers. We describe the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, contrasting them to the constraints of other architectural styles. We then compare the abstract model to the currently deployed Web architecture in order to elicit mismatches between the existing protocols and the applications they are intended to support.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; H.5.4 [**Hypertext/Hypermedia**]: Architectures; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services*

General Terms: Design, Performance, Standardization

Additional Key Words and Phrases: Network-based applications, REST, World Wide Web

This work was partially supported by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021, and originated while the first author was at the University of California, Irvine.
An earlier version of this paper appeared in the *Proceedings of the 22nd International Conference on Software Engineering*, Limerick, Ireland, June 2000 (ICSE 2000), 407–416.
Authors' addresses: R. T. Fielding, Day Software, 2 Corporate Plaza, Suite 150, Newport Beach, CA 92660; email: roy.fielding@day.com; R. N. Taylor, Information and Computer Science, University of California, Irvine CA 92697-3425; email: taylor@ics.uci.edu.

1. INTRODUCTION

At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [Berners-Lee et al. 1996] and design the extensions for the new standards of HTTP/1.1 [Fielding et al. 1999] and Uniform Resource Identifiers (URI) [Berners-Lee et al. 1998], we recognized the need for a model of how the World Wide Web (WWW, or simply Web) *should* work. This idealized model of the interactions within an overall Web application—what we refer to as the Representational State Transfer (REST) architectural style—became the foundation for the modern Web architecture, providing the guiding principles by which flaws in the existing architecture could be identified and extensions validated prior to deployment.

A software architecture is an abstraction of the runtime elements of a software system during some phase of its operation [Fielding 2000]. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture. An architecture determines how system elements are identified and allocated, how the elements interact to form a system, the amount and granularity of communication needed for interaction, and the interface protocols used for communication. An architectural style is a coordinated set of architectural constraints that restricts the roles and features of architectural elements, and the allowed relationships among those elements, within any architecture that conforms to the style. Thus, a style provides a name by which we can refer to a packaged set of architectural design decisions and the set of architectural properties that are induced by applying the style.

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

The first edition of REST was developed between October 1994 and August 1995, primarily as a means for communicating Web concepts while developing the HTTP/1.0 specification and the initial HTTP/1.1 proposal. It was iteratively improved over the next five years and applied to various revisions and extensions of the Web protocol standards. REST was originally referred to as the "HTTP object model," but that name often led to its misinterpretation 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 forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user.

The modern Web is one instance of a REST-style architecture. Although Web-based applications can include access to other styles of interaction, the central focus of its protocol and performance concerns is distributed hypermedia. REST

elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction. Areas for improvement of the Web architecture can be seen where existing protocols fail to express all of the potential semantics for component interaction, and where the details of syntax can be replaced with more efficient forms without changing the architecture capabilities. Likewise, proposed extensions can be compared to REST to see if they fit within the architecture; if not, it is usually more efficient to redirect that functionality to a system running in parallel with a more applicable architectural style.

This article presents REST after many years of work on architectural standards for the modern (post-1993) Web. It does not present the details of the Web architecture, since those are found within the standards. Instead, we focus on the rationale behind the modern Web's architectural design and the software engineering principles upon which it is based. In the process, we identify areas where the Web protocols have failed to match the style, the extent to which these failures can be fixed within the immediate future via protocol enhancements, and the lessons learned from using an interaction style to guide the design of a distributed architecture.

2. WWW REQUIREMENTS

Architecting the Web requires an understanding of its requirements. Berners-Lee [1996] writes that the "Web's major goal was to be a shared information space through which people and machines could communicate." What was needed was a way for people to store and structure their own information, whether permanent or ephemeral in nature, such that it could be usable by themselves and others, and to be able to reference and structure the information stored by others so that it would not be necessary for everyone to keep and maintain local copies.

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

2.1 Low Entry-Barrier

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

Hypermedia was chosen as the user interface due to its simplicity and generality: the same interface can be used regardless of the information source, the

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

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

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

2.2 Extensibility

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

2.3 Distributed Hypermedia

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

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

2.4 Internet-Scale

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

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

The anarchic scalability requirement applies to all architectural elements. Clients cannot be expected to maintain knowledge of all servers. Servers cannot be expected to retain knowledge of state across requests. Hypermedia data elements cannot retain "back-pointers," an identifier for each data element that references them, since the number of references to a resource is proportional to the number of people interested in that information. Particularly newsworthy information can also lead to "flash crowds": sudden spikes in access attempts as news of its availability spreads across the world.

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

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

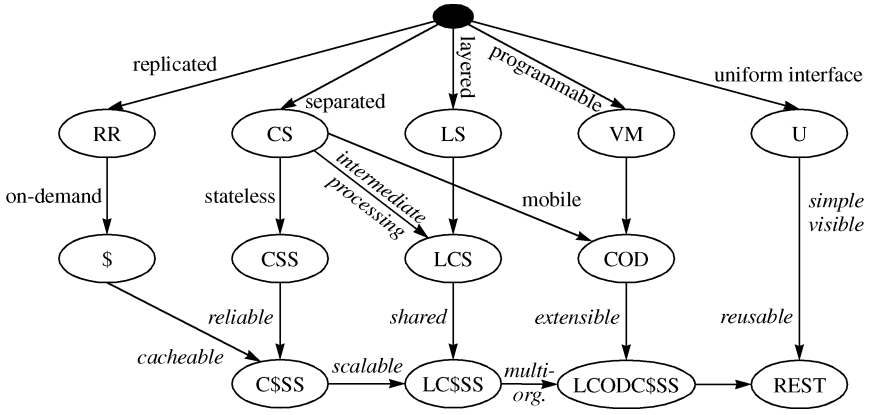

Fig. 1. REST derivation by style constraints.

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

2.5 Evolving Requirements

Each of these project goals and information system characteristics fed into the design of the Web's architecture. As the Web has matured, additional goals have been added to support greater collaboration and distributed authoring [Fielding et al. 1998]. The introduction of each new goal presents us with a challenge: how do we introduce a new set of functionality to an architecture that is already widely deployed, and how do we ensure that its introduction does not adversely impact, or even destroy, the architectural properties that have enabled the Web to succeed? These questions motivated our development of the REST architectural style.

3. DERIVING REST AS A HYBRID ARCHITECTURAL STYLE

The REST architectural style consists of a set of architectural constraints chosen for the properties they induce on candidate architectures. Although each of these constraints can be considered in isolation, describing them in terms of their derivation from common architectural styles makes it easier to understand the rationale behind their selection. Figure 1 depicts the derivation of REST's constraints graphically in terms of the network-based architectural styles examined in Fielding [2000]. The relevant base styles from which REST was derived include replicated repository (RR), cache ($), client-server (CS), layered system (LS), stateless (S), virtual machine (VM), code on demand (COD), and uniform interface (U).

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

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

We next add a constraint to the client-server interaction: communication must be stateless in nature, as in the client-stateless-server (CSS) style, such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client. This constraint induces the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures [Waldo et al. 1994]. Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests.

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

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

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

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

Developers of Web implementations had already exceeded the early design. In addition to static documents, requests could identify services that dynamically generated responses, such as image maps and server-side scripts. Work had also begun on intermediary components, in the form of proxies [Luotonen and Altis 1994] and shared caches [Glassman 1994], but extensions to the protocols were needed in order for them to communicate reliably. The remaining constraints were added to the Web's architectural style in order to guide the extensions that form the modern Web architecture.

The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.

In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state. These constraints are discussed in Section 4.

In order to further improve behavior for Internet-scale requirements, we add layered system constraints. The layered system style (LS) allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.

The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance [Clark and Tennenhouse 1990]. For a network-based system that supports cache constraints, this can be offset by the benefits of shared caching at intermediaries. Placing shared caches at the boundaries of an organizational domain can result in significant performance benefits [Wolman et al. 1999]. Such layers also allow security policies to be enforced on data crossing the organizational boundary, as is required by firewalls [Luotonen and Altis 1994].

The combination of layered system and uniform interface constraints induces architectural properties similar to those of the uniform pipe-and-filter style. Although REST interaction is two-way, the large-grain dataflows of hypermedia

interaction can each be processed like a dataflow network, with filter components selectively applied to the data stream in order to transform the content as it passes [Brooks et al. 1995]. Within REST, intermediary components can actively transform the content of messages because the messages are self-descriptive and their semantics are visible to intermediaries.

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

The notion of an optional constraint may seem like an oxymoron. However, it does have a purpose in the architectural design of a system that encompasses multiple organizational boundaries. It means that the architecture only gains the benefit (and suffers the disadvantages) of the optional constraints when they are known to be in effect for some realm of the overall system. For example, if all of the client software within an organization is known to support Java™ applets [Flanagan 1999], then services within that organization can be constructed such that they gain the benefit of enhanced functionality via downloadable Java™ classes. At the same time, however, the organization's firewall may prevent the transfer of Java™ applets from external sources, and thus to the rest of the Web it will appear as if those clients do not support code-on-demand. An optional constraint allows us to design an architecture that supports the desired behavior in the general case, but with the understanding that it may be disabled within some contexts.

4. REST ARCHITECTURAL ELEMENTS

The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. Perry and Wolf [1992] distinguish three classes of architectural elements: processing elements (a.k.a., components), data elements, and connecting elements (a.k.a., connectors). REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It establishes the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.

Using the software architecture framework of Perry and Wolf [1992], we first define the architectural elements of REST and then examine sample process, connector, and data views of prototypical architectures to gain a better understanding of REST's design principles.

4.1 Data Elements

Unlike the distributed object style [Chin and Chanson 1991], where all data is encapsulated within and hidden by the *processing* components, the nature and

Table I. REST Data Elements

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

state of an architecture's data elements is a key aspect of REST. The rationale for this design can be seen in the nature of distributed hypermedia.

When a link is selected, information needs to be moved from the location where it is stored to the location where it will be used by, in most cases, a human reader. This is in distinct contrast to most distributed processing paradigms [Andrews 1991; Fuggetta et al. 1998], where it is often more efficient to move the "processing entity" to the data rather than move the data to the processor. A distributed hypermedia architect has only three fundamental options: (1) render the data where it is located and send a fixed-format image to the recipient; (2) encapsulate the data with a rendering engine and send both to the recipient; or (3) send the raw data to the recipient along with metadata that describes the data type, so that the recipient can choose their own rendering engine.

Each option has its advantages and disadvantages. Option 1, the traditional client/server style [Sinha 1992], allows all information about the true nature of the data to remain hidden within the sender, preventing assumptions from being made about the data structure and making client implementation easier. However, it also severely restricts the functionality of the recipient and places most of the processing load on the sender, leading to scalability problems. Option 2, the mobile object style [Fuggetta et al. 1998], provides information hiding while enabling specialized processing of the data via its unique rendering engine, but limits the functionality of the recipient to what is anticipated within that engine, and may vastly increase the amount of data transferred. Option 3 allows the sender to remain simple and scalable while minimizing the bytes transferred, but loses the advantages of information hiding and requires that both sender and recipient understand the same data types.

REST provides a hybrid of all three options by focusing on a shared understanding of data types with metadata, but limiting the scope of what is revealed to a standardized interface. REST components communicate by transferring a representation of the data in a format matching one of an evolving set of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the data. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface. The benefits of the mobile object style are approximated by sending a representation that consists of instructions in the standard data format of an encapsulated rendering engine (e.g., Java™). REST therefore gains the separation of concerns of the client/server style without the server scalability problem, allows information hiding through

a generic interface to enable encapsulation and evolution of services, and provides for a diverse set of functionality through downloadable feature-engines.

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

More precisely, a resource R is a temporally varying membership function $M_R(t)$, which for time t maps to a set of entities, or values, which are equivalent. The values in the set may be *resource representations* and/or *resource identifiers*. A resource can map to the empty set, which allows references to be made to a concept before any realization of that concept exists—a notion that was foreign to most hypertext systems prior to the Web [Grønbaek and Trigg 1994]. Some resources are static in the sense that, when examined at any time after their creation, they always correspond to the same value set. Others have a high degree of variance in their value over time. The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another.

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

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

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

Traditional hypertext systems [Grønbaek and Trigg 1994], which typically operate in a closed or local environment, use unique node or document identifiers that change every time the information changes, relying on link servers to maintain references separately from the content. Since centralized link servers are anathema to its immense scale and multiorganizational domain requirements, REST relies instead on the author choosing a resource identifier that best fits the nature of the concept being identified. Naturally, the quality of an identifier is often proportional to the amount of money spent to retain its validity, which leads to broken links as ephemeral (or poorly supported) information moves or disappears over time.

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

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

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

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

The data format of a representation is known as a *media type* [Postel 1996]. A representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type. Some media types are intended for automated processing, some to be rendered for viewing by a user, and a few are appropriate for both. Composite media types can be used to enclose multiple representations in a single message.

The design of a media type can directly impact the user-perceived performance of a distributed hypermedia system. Any data that must be received before the recipient can begin rendering the representation adds to the latency of an interaction. A data format that places the most important rendering

information up front, such that the initial information can be incrementally rendered while the rest of the information is being received, results in much better user-perceived performance than a data format that must be received entirely before rendering can begin.

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

4.2 Connectors

REST uses various connector types to encapsulate the activities of accessing resources and transferring resource representations. The connectors present an abstract interface for component communication, enhancing simplicity by providing a clean separation of concerns and hiding the underlying implementation of resources and communication mechanisms. The generality of the interface also enables substitutability: if the users' only access to the system is via an abstract interface, the implementation can be replaced without impacting the users. Since a connector manages network communication for a component, information can be shared across multiple interactions in order to improve efficiency and responsiveness.

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

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

The primary connector types are client and server. The essential difference between the two is that a *client* initiates communication by making a request,

Table II. REST Connector Types

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

whereas a *server* listens for connections and responds to requests in order to supply access to its services. A component may include both client and server connectors.

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

Some cache connectors are shared, meaning that its cached responses may be used in answer to a client other than the one for which the response was originally obtained. Shared caching can be effective at reducing the impact of "flash crowds" on the load of a popular server, particularly when the caching is arranged hierarchically to cover large groups of users, such as those within a company's intranet, the customers of an Internet service provider, or universities sharing a national network backbone. However, shared caching can also lead to errors if the cached response does not match what would have been obtained by a new request. REST attempts to balance the desire for transparency in cache behavior with the desire for efficient use of the network, rather than assuming that absolute transparency is always required.

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

A *resolver* translates partial or complete resource identifiers into the network address information needed to establish an intercomponent connection. For example, most URI include a DNS hostname as the mechanism for identifying the naming authority for the resource. In order to initiate a request, a Web browser will extract the hostname from the URI and make use of a DNS resolver to obtain the Internet protocol address for that authority. Another example is that some identification schemes (e.g., URN [Sollins and Masinter 1994]) require an intermediary to translate a permanent identifier to a more transient address in order to access the identified resource. Use of one or more intermediate resolvers can improve the longevity of

Table III. REST Component Types

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

resource references through indirection, though doing so adds to the request latency.

The final form of connector type is a *tunnel*, which simply relays communication across a connection boundary, such as a firewall or lower-level network gateway. The only reason it is modeled as part of REST and not abstracted away as part of the network infrastructure is that some REST components may dynamically switch from active component behavior to that of a tunnel. The primary example is an HTTP proxy that switches to a tunnel in response to a CONNECT method request, thus allowing its client to directly communicate with a remote server using a different protocol, such as TLS, which doesn't allow proxies. The tunnel disappears when both ends terminate their communication.

4.3 Components

REST components (processing elements) are typed by their roles in an overall application action.

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

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

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

5. REST ARCHITECTURAL VIEWS

Now that we have an understanding of the REST architectural elements in isolation, we can use architectural views [Perry and Wolf 1992] to describe how

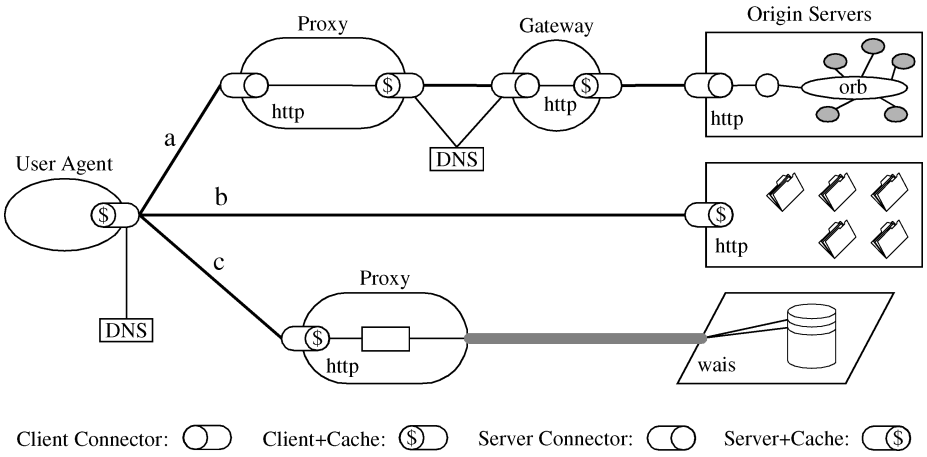

Fig. 2. Process view of a REST-based architecture at one instance in time. A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according to the properties of each resource identifier and the configuration of the client connector. Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture. Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache. Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface. Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.

the elements work together to form an architecture. All three types of view—process, connector, and data—are useful for illuminating the design principles of REST.

5.1 Process View

A process view of an architecture is primarily effective at eliciting the interaction relationships among components by revealing the path of data as it flows through the system. Unfortunately, the interaction of a real system usually involves an extensive number of components, resulting in an overall view that is obscured by the details. Figure 2 provides a sample of the process view from a REST-based architecture at a particular instance during the processing of three parallel requests.

The client/server [Andrews 1991] separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components.

Since the components are connected dynamically, their arrangement and function for a particular application action has characteristics similar to a pipe-and-filter style. Although REST components communicate via bidirectional streams, the processing of each direction is independent and therefore

susceptible to stream transducers (filters). The generic connector interface allows components to be placed on the stream based on the properties of each request or response.

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

5.2 Connector View

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

Client connectors examine the resource identifier in order to select an appropriate communication mechanism for each request. For example, a client may be configured to connect to a specific proxy component, perhaps one acting as an annotation filter, when the identifier indicates that it is a local resource. Likewise, a client can be configured to reject requests for some subset of identifiers.

Although the Web's primary transfer protocol is HTTP, the architecture includes seamless access to resources that originate on many pre-existing network servers, including FTP [Postel and Reynolds 1985], Gopher [Anklesaria et al. 1993], and WAIS [Davis et al. 1990]. However, interaction with these services is restricted to the semantics of a REST connector. This constraint sacrifices some of the advantages of other architectures, such as the stateful interaction of a relevance feedback protocol like WAIS, in order to retain the advantages of a single, generic interface for connector semantics. This generic interface makes it possible to access a multitude of services through a single proxy connection. If an application needs the additional capabilities of another architecture, it can implement and invoke those capabilities as a separate system running in parallel, similar to how the Web architecture interfaces with "telnet" and "mailto" resources.

5.3 Data View

A data view of an architecture reveals the application state as information flows through the components. Since REST is specifically targeted at distributed information systems, it views an application as a cohesive structure of information and control alternatives through which a user can perform a desired task. For example, an online dictionary is one application, as is a museum tour or a set of class notes.

Component interactions occur in the form of dynamically-sized messages. Small- or medium-grain messages are used for control semantics, but the bulk

of application work is accomplished via large-grain messages containing a complete resource representation. The most frequent form of request semantics is retrieving a representation of a resource (e.g., the "GET" method in HTTP), which can often be cached for later reuse.

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

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

The user-perceived performance of a browser application is determined by the latency between steady states: the period of time between the selection of a hypermedia link or submit button on one Web page and the point when usable information has been rendered for the next Web page. The optimization of browser performance is therefore centered around reducing this latency, which leads to the following observations:

—The most efficient network request is one that doesn't use the network. In other words, reusing a cached response results in the best performance. Although use of a cache adds some latency to each individual request due to lookup overhead, the average request latency is significantly reduced when even a small percentage of requests result in usable cache hits.

—The next control state of the application resides in the representation of the first requested resource, so obtaining that first representation is a priority.

—Incremental rendering of the first nonredirect response representation can considerably reduce latency, since then the representation can be rendered as it is being received rather than after the response has been completed. Incremental rendering is impacted by the design of the media type and the early availability of layout information (visual dimensions of in-line objects).

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

The model application is therefore an engine that moves from one state to the next by examining and choosing from among the alternative state transitions in the current set of representations. Not surprisingly, this exactly matches the user interface of a hypermedia browser. However, the style does not assume that all applications are browsers. In fact, the application details are hidden from the server by the generic connector interface, and thus a user agent could equally be an automated robot performing information retrieval for an indexing service, a personal agent looking for data that matches certain criteria, or a maintenance spider busy patrolling the information for broken references or modified content [Fielding 1994].

6. RELATED WORK

Garlan and Shaw [1993] provide an introduction to software architecture research and describe several "pure" styles. Their work differs significantly from the framework of Perry and Wolf [1992] used in this article due to a lack of consideration for data elements. As observed above, the characteristics of data elements are fundamental to understanding the modern Web architecture—it simply cannot be adequately described without them. The same conclusion can be seen in the comparison of mobile code paradigms by Fuggetta et al. [1998], where the analysis of when to go mobile depends on active comparison of the size of the code that would be transferred versus the preprocessed information that would otherwise be transferred.

Bass et al. [1998] devote a chapter on architecture for the World Wide Web, but their description only encompasses the implementation architecture within the CERN/W3C-developed libwww (client and server libraries) and Jigsaw software. Although those implementations reflect some of the design constraints of REST, having been developed by people familiar with the intended architectural style, the real WWW architecture is independent of any single implementation. The Web is defined by its standard interfaces and protocols, not how those interfaces and protocols are implemented in a given piece of software.

The REST style draws from many preexisting distributed process paradigms [Andrews 1991; Fuggetta et al. 1998], communication protocols, and software fields. REST component interactions are structured in a layered client-server style, but the added constraints of the generic resource interface create the opportunity for substitutability and inspection by intermediaries. Requests and responses have the appearance of a remote invocation style, but REST messages are targeted at a conceptual resource rather than an implementation identifier.

Several attempts have been made to model the Web architecture as a form of distributed file system (e.g., WebNFS) or as a distributed object system [Manola 1999]. However, they exclude various Web resource types or implementation strategies as being "not interesting," when in fact their presence invalidates the assumptions that underlie such models. REST works well because it does not limit the implementation of resources to certain predefined models, allowing each application to choose an implementation that best matches its own needs and enabling the replacement of implementations without impacting the user.

The interaction method of sending representations of resources to consuming components has some parallels with event-based integration (EBI) styles [Barrett et al. 1996; Rosenblum and Wolf 1997; Sullivan and Notkin 1992]. The key difference is that EBI styles are push-based. The component containing the state (equivalent to an origin server in REST) issues an event whenever the state changes, whether or not any component is actually interested in or listening for such an event. In the REST style, consuming components usually pull representations. Although this is less efficient when viewed as a single client wishing to monitor a single resource, the scale of the Web makes an unregulated push model infeasible.

The principled use of the REST style in the Web, with its clear notion of components, connectors, and representations, relates closely to the C2 architectural style [Taylor et al. 1996]. The C2 style supports the development of distributed, dynamic applications by focusing on structured use of connectors to obtain substrate independence. C2 applications rely on asynchronous notification of state changes and request messages. As with other event-based schemes, C2 is nominally push-based, though a C2 architecture could operate in REST's pull style by only emitting a notification upon receipt of a request. However, the C2 style lacks the intermediary-friendly constraints of REST, such as the generic resource interface, guaranteed stateless interactions, and intrinsic support for caching.

7. EXPERIENCE AND EVALUATION

In an ideal world, the implementation of a software system would exactly match its design. Some features of the modern Web architecture do correspond exactly to their design criteria in REST, such as the use of URI [Berners-Lee et al. 1998] as resource identifiers and the use of Internet media types [Postel 1996] to identify representation data formats. However, there are also some aspects of the modern Web protocols that exist in spite of the architectural design, due to legacy experiments that failed (but must be retained for backwards compatibility) and extensions deployed by developers unaware of the architectural style. REST provides a model not only for the development and evaluation of new features, but also for the identification and understanding of broken features.

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.

7.1 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 [Berners-Lee 1994], and finally the combination of Uniform Resource Locators (URL) [Berners-Lee et al. 1994] and Names (URN) [Sollins and Masinter 1994] that are collectively referred to as URI. 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 [Berners-Lee et al. 1998], as well as the overall semantics of the generic interface for manipulating resources via their representations.

7.1.1 *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, 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.

7.1.2 *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 just 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.

7.1.3 *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.

7.1.4 *Binding Semantics to URI.* Semantics are a byproduct 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.

7.1.5 *REST Mismatches in URI.* Like most real-world systems, not all components of the deployed Web architecture obey every constraint 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, to track user behavior by logging their actions, or carry user preferences across multiple actions (e.g., Hyper-G's gateways [Maurer 1996]). 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., content distribution 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).

7.2 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 [Berners-Lee et al. 1996], analyze proposed extensions for HTTP/1.1 [Fielding et al. 1999], and provide motivating rationale for deploying HTTP/1.1.

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

7.2.1 *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.

Protocol versioning. HTTP is a family of protocols, distinguished by major and minor version numbers, which 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 [Mogul et al. 1997].

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 that are forbidden to send 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 interpreted by immediate neighbors in the

communication, since HTTP does not require that the entire request/response chain of intermediary components speak the same version.

These rules exist to assist in deploying multiple protocol revisions and preventing 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.

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 [Berners-Lee et al. 1998], media types [Freed et al. 1996], MIME header field names [Freed and Borenstein 1996], charset values, language tags), while others are governed by HTTP, including the namespaces for method names, response status codes, nonMIME 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 among 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.

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.

7.2.2 *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.

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.

Layered encodings. HTTP inherited its syntax for describing representation metadata from the Multipurpose Internet Mail Extensions (MIME) [Freed and Borenstein 1996]. 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.

Semantic independence. As described above, HTTP message parsing has been separated from its semantics. Message parsing, including finding and cobbling the header fields, occurs separately 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.

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 network failure. To solve this, the Content-Length header field was redefined within HTTP/1.0 to indicate the message body length in bytes, whenever the length was 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.

Size limits. A frequent barrier to the flexibility of application-layer protocols is the tendency to over-specify size limits on protocol parameters. Although some practical limits within implementations of the protocol always exist (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 vision of its original creator.

In the HTTP protocol there is no limit on the length of the URI, the length of header fields, the length of a 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 following conditions: 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.

Cache control. Because REST tries to balance the need for efficient, low-latency behavior against 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.

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 implementations of content negotiation deployed, they were 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 1 KB of preference information in the header fields on every request, regardless of the negotiability of the resource [Spero 1994]. 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 received response. HTTP/1.1 introduced the *Vary* header field for this purpose. Vary simply lists those request header field dimensions under which the response may vary.

In preemptive negotiation, the user agent tells the server what it can accept. 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 nontractable 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 users intend to put the representation. For example, users who want to view an image on screen might prefer a simple bitmap representation, but the same users with the same browsers may prefer a PostScript representation if they intend to send it to a printer instead. It also depends on the users correctly configuring their browsers 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 purposes. 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 [Holtman and Mutz 1998], 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.

7.2.3 *Performance.* HTTP/1.1 was 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.

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 [Heidemann et al. 1997; Spero 1994]. 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 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 [Padmanabhan and Mogul 1995]. For HTTP/1.0, this was done using the "keep-alive" directive within the Connection header field. Unfortunately, this 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 became possible only after HTTP messages were redefined to be self-descriptive and independent of the underlying transport protocol.

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.

7.2.4 *REST Mismatches in HTTP Extensions.* 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.

Differentiating nonauthoritative responses. One weakness that still exists in HTTP is that there is no consistent mechanism for differentiating between authoritative responses generated by the origin server in response to the current request, and nonauthoritative responses 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 in 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 so 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 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 is to require that responses be marked as nonauthoritative 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.

Cookies. The introduction of site-wide state information in the form of HTTP cookies [Kristol and Montulli 1997] is an example of an inappropriate extension to the protocol. Cookie interaction fails to match REST's application state model, 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 that the user agent include the same cookie in 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 in 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 function (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 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 users as they browse between sites.

As a result, cookie-based applications on the Web will never be reliable. The same functionality should have been achieved via anonymous authentication

and true client-side state. A state mechanism that involves preferences can be more efficiently implemented with judicious use of context-setting URI rather than cookies, where *judicious* means one URI per state rather than an unbounded number of URIs 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 his or her own client-side shopping basket, complete with a URI to be used for check-out when the client is ready to purchase.

Mandatory extensions. HTTP header field names can be extended at will, but only when the information they contain is not required for properly understanding the message. Mandatory header field extensions require a major protocol revision or a substantial change to method semantics, such as that proposed in [Nielsen et al. 2000]. This is an aspect of modern Web architecture that 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 might be gained 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.

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 for forwarding by intermediaries, as well as an algorithm for the canonical treatment of header field digests [Franks et al. 1999].

MIME syntax. HTTP inherited its message syntax from MIME [Freed and Borenstein 1996] 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, user agents send a bunch of information, which is to be treated as a coherent whole, to unknown recipients with whom they never directly interact. MIME assumes that the agents 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"—user agents 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 that the transport is lossy, deliberately corrupting things like line breaks and content lengths. Hence the syntax is verbose and inefficient for any system not based on a lossy transport, which makes it inappropriate for HTTP. Since HTTP/1.1 can 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.

8. CONCLUSIONS AND FUTURE WORK

The World Wide Web is arguably the world's largest distributed application. Understanding the key architectural principles underlying the Web can help explain its technical success and may lead to improvements in other distributed applications, particularly those amenable to the same or similar methods of interaction.

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

The REST architectural style has succeeded in guiding the design and deployment of modern Web architecture. It has been validated through development of the HTTP/1.0 [Berners-Lee et al. 1996] and HTTP/1.1 [Fielding et al. 1999] standards, elaboration of the URI [Berners-Lee et al. 1998] and relative URL [Fielding 1995] standards, and successful deployment of several dozen independently developed, commercial-grade software systems within the modern Web architecture. REST has served as both a model for design guidance and as an acid test for architectural extensions to the Web protocols. 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.

Future work will focus on extending the architectural guidance toward the development of a replacement for the HTTP/1.x protocol, using a more efficient tokenized syntax, but without losing the desirable properties identified by REST. There has also been some interest in extending REST to consider variable request priorities, differentiated quality-of-service, and representations consisting of continuous data streams, such as those generated by broadcast audio and video sources. Independent of the original authors, a public discussion list and collaboratively authored Web forum [Baker et al. 2002] have been created to further explore the REST architectural style and examine how its constraints might be selectively applied to other software systems.

ACKNOWLEDGMENTS

This article is a subset of the first author's dissertation work [Fielding 2000]. He would like to acknowledge Mark Ackerman for introducing him to the Web developer community and David Rosenblum whose work on Internet-scale software architectures inspired this research in terms of architecture, rather than simply hypermedia or application-layer protocol design. Kari A. Nies assisted in writing this article by extracting the relevant parts of the dissertation that had been updated since its original publication and helping to make sense of them with less context.

The Web's architectural style was developed iteratively over a six-year period, but primarily during the first six months of 1995. It was influenced by countless discussions with researchers at UCI, staff at the World Wide Web Consortium (W3C), and engineers within the HTTP and URI working groups of the Internet Engineering Taskforce (IETF). We thank Tim Berners-Lee, Henrik Frystyk Nielsen, Dan Connolly, Dave Raggett, Rohit Khare, Jim Whitehead, Larry Masinter, and Dan LaLiberte for many thoughtful conversations regarding the nature and goals of the WWW architecture.

REFERENCES

ANDREWS, G. 1991. Paradigms for process interaction in distributed programs. *ACM Computing Surv. 23*, 1 (March 1991), 49–90.

ANKLESARIA, F., ET AL. 1993. The Internet Gopher protocol (a distributed document search and retrieval protocol). *Internet RFC 1436*, March 1993.

BAKER, M., ET AL. 2002. RESTwiki. <*http://conveyor.com/RESTwiki/moin.cgi*>, April 2002.

BARRETT, D. J., CLARKE, L. A., TARR, P. L., AND WISE, A. E. 1996. A framework for event-based software integration. *ACM Trans. Soft. Eng. Methodol. 5*, 4 (Oct. 1996), 378–421.

BASS, L., CLEMENTS, P., AND KAZMAN, R. 1996. *Software Architecture in Practice*. Addison Wesley, Reading, MA.

BERNERS-LEE, T. 1994. Universal resource identifiers in WWW. *Internet RFC 1630*, June 1994.

BERNERS-LEE, T., MASINTER, L., AND MCCAHILL, M. 1994. Uniform resource locators (URL). *Internet RFC 1738*, Dec. 1994.

BERNERS-LEE, T. AND CONNOLLY, D. 1995. Hypertext markup language—2.0. *Internet RFC 1866*, Nov. 1995.

BERNERS-LEE, T. 1996. WWW: Past, present, and future. *Computer 29*, 10 (Oct. 1996), 69–77.

BERNERS-LEE, T., FIELDING, R. T., AND NIELSEN, H. F. 1996. Hypertext transfer protocol—HTTP/1.0. *Internet RFC 1945*, May 1996.

BERNERS-LEE, T., FIELDING, R. T., AND MASINTER, L. 1998. Uniform resource identifiers (URI): Generic syntax. *Internet RFC 2396*, Aug. 1998.

BROOKS, C., MAZER, M. S., MEEKS, S., AND MILLER, J. 1995. Application-specific proxy servers as HTTP stream transducers. In *Proceedings of the Fourth International World Wide Web Conference* (Boston, MA, Dec. 1995), 539–548.

CHIN, R. S. AND CHANSON, S. T. 1991. Distributed object-based programming systems. *ACM Comput. Surv. 23*, 1 (March 1991), 91–124.

CLARK, D. D. AND TENNENHOUSE, D. L. 1990. Architectural considerations for a new generation of protocols. In *Proceedings ACM SIGCOMM'90 Symposium* (Philadelphia, PA, Sept. 1990), 200–208.

DAVIS, F., ET. AL. 1990. WAIS interface protocol prototype functional specification (v.1.5). Thinking Machines Corp., Apr. 1990.

FIELDING, R. T. 1995. Relative uniform resource locators. *Internet RFC 1808*, June 1995.

FIELDING, R. T. 1994. Maintaining distributed hypertext infostructures: Welcome to MOMspider's web. *Comput. Net. ISDN Syst. 27*, 2 (Nov. 1994), 193–204.

FIELDING, R. T. 2000. Architectural styles and the design of network-based software architectures. PhD Dissertation. Dept. of Information and Computer Science, University of California, Irvine.

FIELDING, R. T., GETTYS, J., MOGUL, J. C., NIELSEN, H. F., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. 1999. Hypertext transfer protocol—HTTP/1.1. *Internet RFC 2616*, June 1999.

FIELDING, R. T., WHITEHEAD, E. J., JR., ANDERSON, K. M., BOLCER, G., OREIZY, P., AND TAYLOR, R. N. 1998. Web-based development of complex information products. *Commun. ACM 41*, 8 (Aug. 1998), 84–92.

FLANAGAN, D. 1999. *Java™ in a Nutshell, 3rd ed.* O'Reilly & Associates, Sebastopol.

FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., SINK, E., AND STEWART, L. 1999. HTTP authentication: Basic and digest access authentication. *Internet RFC 2617*, June 1999.

FREED, N. AND BORENSTEIN, N. 1996. Multipurpose internet mail extensions (MIME) Part One: Format of internet message bodies. *Internet RFC 2045*, Nov. 1996.

FREED, N., KLENSIN, J., AND J. POSTEL, J. 1996. Multipurpose internet mail extensions (MIME) Part Four: Registration procedures. *Internet RFC 2048*, Nov. 1996.

FUGGETTA, A., PICCO, G. P., AND VIGNA, G. 1998. Understanding code mobility. *IEEE Trans. Soft. Eng. 24*, 5 (May 1998), 342–361.

GARLAN, D. AND SHAW, M. 1993. An introduction to software architecture. Ambriola and Tortola, eds. In *Advances in Software Engineering & Knowledge Engineering, vol. II*, World Scientific 1993, 1–39.

GLASSMAN, S. 1994. A caching relay for the World Wide Web. *Comput. Net. ISDN Syst. 27*, 2 (Nov. 1994), 165–173.

GRØNBAEK, K. AND TRIGG, R. H. 1994. Design issues for a Dexter-based hypermedia system. *Commun. ACM 37*, 2 (Feb. 1994), 41–49.

HEIDEMANN, J., OBRACZKA, K., AND TOUCH, J. 1997. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Trans. Net. 5*, 5 (Oct. 1997), 616–630.

HOLTMAN, K. AND MUTZ, A. 1998. Transparent content negotiation in HTTP. *Internet RFC 2295*, March 1998.

KRISTOL, D. AND MONTULLI, L. 1997. HTTP state management mechanism. *Internet RFC 2109*, Feb. 1997.

LUOTONEN, A. AND ALTIS, K. 1994. World-Wide Web proxies. *Comput. Net. ISDN Syst. 27*, 2 (Nov. 1994), 147–154.

MANOLA, F. 1999. Technologies for a Web object model. *IEEE Internet Comput. 3*, 1 (Jan.-Feb. 1999), 38–47.

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

MOGUL, J., FIELDING, R. T., GETTYS, J., AND NIELSEN, H. F. 1997. Use and interpretation of HTTP version numbers. *Internet RFC 2145*, May 1997.

NIELSEN, H. F., LEACH, P., AND LAWRENCE, S. 2000. HTTP extension framework, *Internet RFC 2774*, Feb. 2000.

PADMANABHAN, V. N. AND MOGUL, J. C. 1995. Improving HTTP latency. *Comput. Net. ISDN Syst. 28* (Dec. 1995), 25–35.

PERRY, D. E. AND WOLF, A. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Soft. Eng. Notes 17*, 4 (Oct. 1992), 40–52.

POSTEL, J. 1996. Media type registration procedure. *Internet RFC 1590*, Nov. 1996.

POSTEL, J. AND REYNOLDS, J. 1985. File transfer protocol. *Internet STD 9, RFC 959*, Oct. 1985.

ROSENBLUM, D. S. AND WOLF, A. L. 1997. A design framework for Internet-scale event observation and notification. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Zurich, Switzerland, Sept. 1997), 344–360.

SINHA, A. 1992. Client-server computing. *Commun. ACM 35*, 7 (July 1992), 77–98.

SOLLINS, K. AND MASINTER, L. 1994. Functional requirements for Uniform Resource Names. *Internet RFC 1737*, Dec. 1994.

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

SULLIVAN, K. J. AND NOTKIN, D. 1992. Reconciling environment integration and software evolution. *ACM Trans. Soft. Eng. Methodol. 1*, 3 (July 1992), 229–268.

TAYLOR, R. N., MEDVIDOVIC, N., ANDERSON, K. M., WHITEHEAD JR., E. J., ROBBINS, J. E., NIES, K. A., OREIZY, P., AND DUBROW, D. L. 1996. A component- and message-based architectural style for GUI software. *IEEE Trans. Soft. Eng. 22*, 6 (June 1996), 390–406.

WALDO, J., WYANT, G., WOLLRATH, A., AND KENDALL, S. 1994. A note on distributed computing. *Tech. Rep. SMLI TR-94-29*, Sun Microsystems Laboratories, Inc., Nov. 1994.

WOLMAN, A., VOELKER, G., SHARMA, N., CARDWELL, N., BROWN, M., LANDRAY, T., PINNEL, D., KARLIN, A., AND LEVY, H. 1999. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the 2nd USENIX Conference on Internet Technologies and Systems* (Oct. 1999).

Received December 2001; revised February 2002; accepted April 2002