Principled Design of the Modern Web Architecture

Roy T. Fielding and Richard N. Taylor
Information and Computer Science
University of California, Irvine
Irvine, CA 92697–3425 USA
+1.949.824.4121
{fielding,taylor}@ics.uci.edu

ABSTRACT
The World Wide Web has succeeded in large part because its software architecture has been designed to meet the needs of an *Internet-scale* distributed hypermedia system. The 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 paper, we introduce the Representational State Transfer (REST) architectural style, developed as an abstract model of the Web architecture 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.

Keywords
software architecture, software architectural style, WWW

1 INTRODUCTION

At the beginning of our efforts within the Internet Engineering Taskforce to define the existing Hypertext Transfer Protocol (HTTP/1.0) [5] and design the extensions for the new standards of HTTP/1.1 [10] and Uniform Resource Identifiers (URI) [6], 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 preexisting architecture could be identified and extensions validated prior to deployment.

A software 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 an abstraction of the key aspects within a set of potential architectures (instantiations of the style), encapsulating important decisions about the architectural elements and emphasizing constraints on the elements and their relationships [17]. In other words, a style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to 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, thereby meeting the needs of an Internet-scale distributed hypermedia system.

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

This paper presents REST after the completion of six years' work on architectural standards for the modern (post-1993) Web. It does not present the details of the architecture itself, since those are found within the standards. Instead, we focus

on the unpublished 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 DOMAIN CHARACTERISTICS

In order to understand the REST rationale, we must first examine the goals of the WWW project and the information system characteristics needed to achieve those goals.

Berners-Lee [4] 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 people intended to use 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.

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

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 would be distributed across the global Internet, the architecture needed to minimize network interactions (round-trips within the protocol). Latency occurs at several points in the processing of a distributed application action: 1) the time needed for the user agent to recognize the event that initiated the action; 2) the time required to setup any interaction(s) between components; 3) the time required to transmit each interaction to the components; 4) the time required to process each interaction on those components; and, 5) the time required to complete sufficient transfer and processing of the result of the interaction(s) before the user agent is able to begin rendering a usable result. It is important to note that, although only (3) and (5) represent actual network communication, all five points can be impacted by the architectural style. Furthermore, multiple interactions are additive to latency unless they take place in parallel.

Scalability was also a concern, since the number of references to a resource would be directly proportional to the number of people interested in that information, and particularly newsworthy information would lead to "flash crowds": sudden spikes in access attempts.

Since participation in the creation and structuring of information was voluntary, a low entry-barrier was necessary to enable sufficient adoption. 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 long-lived system like the Web must be prepared for change. Furthermore, because the components participating in Web applications often span multiple organizational boundaries, 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.

All 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 [11]. 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? It was this question that motivated our development of the REST architectural style.

3 REPRESENTATIONAL STATE TRANSFER (REST)

The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. Perry and Wolf [17] 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 encompasses the fundamental

constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.

Using the software architecture framework of [17], 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.

Data Elements

Unlike the distributed object style [7], where all data is encapsulated within and hidden by the processing components, the nature and state of an architecture's data elements is a key aspect of REST. The rationale for this design can be seen in the nature of distributed hypermedia.

When a link is selected, information needs to be moved from the location where it is stored to the location where it will be used by, in most cases, a human reader. This is in distinct contrast to most distributed processing paradigms [1, 12], 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 [20], 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 [12], 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.

Table 1: REST Data Elements

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

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 moniker for a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

More precisely, a resource R is a temporally varying membership function $M_R(t)$, which for time t maps to a set of entities, or values, which are equivalent. The values in the set may be *resource representations* and/or *resource identifiers*. A resource can map to the empty set, which allows references to be made to a concept before any realization of that concept exists — a notion that was foreign to most hypertext systems prior to the Web [14]. 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 the

"published version in the proceedings" is static. These are two distinct resources, even though 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 (assuming the author used the right identifier).

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

Traditional hypertext systems [14], 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 an anathema to its immense scale and multi-organizational domain requirements, the Web 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.

Representations

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

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

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

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

The data format of a representation is known as a *media type* [18]. A representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type. Some media types are intended for automated processing, some are intended to be rendered for viewing by a user, and a few are capable of both. Composite media types can be used to enclose multiple representations in a single message.

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

For example, a Web browser that can incrementally render a large HTML document while it is being received provides significantly better user-perceived performance than one that waits until the entire document is completely received prior to rendering, even though the network performance is the same. Note that the rendering ability of a representation can also be impacted by the choice of content. If the dimensions of dynamically-sized tables and embedded objects must be

determined before they can be rendered, their occurrence within the viewing area of a hypermedia page will increase its latency.

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.

Table 2: 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

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

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

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

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

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

A *resolver* translates partial or complete resource identifiers into the network address information needed to establish an inter-component connection. For example, most URI include a DNS hostname as the mechanism for identifying the naming authority for the resource. In order to initiate a request, a Web browser will extract the hostname from the URI and make use of a DNS resolver to obtain the Internet Protocol address for that authority. Another example is that some identification schemes (e.g., URN [21]) require an intermediary to translate a permanent identifier to a more

transient address in order to access the identified resource. Use of one or more intermediate resolvers can improve the longevity of resource references through indirection, though doing so adds to the request latency.

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

Component Types

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

Table 3: 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

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.

Architectural Views

Now that we have an understanding of the REST architectural elements in isolation, we can use architectural views [17] to describe how 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.

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 1 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 [1] 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.

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

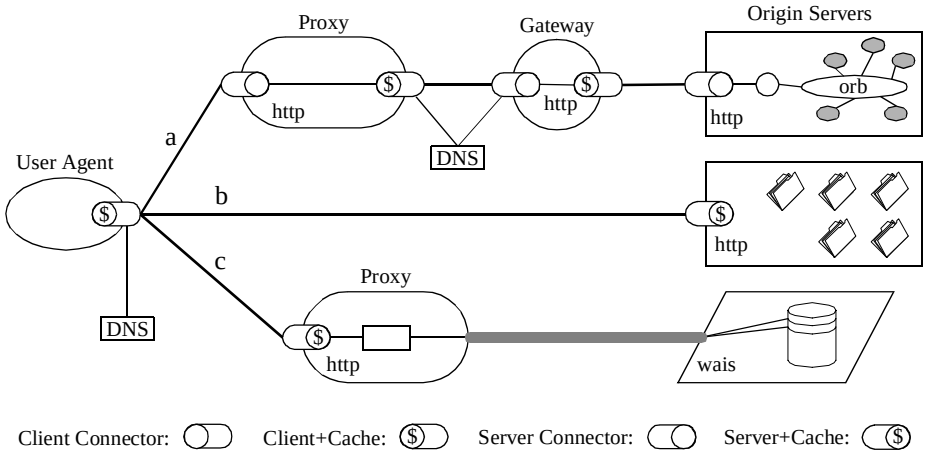

Figure 1: Process view of a REST-based architecture at one instance in time. A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according 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.

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 [19], Gopher [2], and WAIS [8]. 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 likeWAIS, 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.

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 on-line 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 that of retrieving a representation of a resource (e.g., the "GET" method in HTTP), which can often be cached for later reuse.

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

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

impact on both user-perceived performance and the burstiness of network request traffic.

The user-perceived performance of a browser application is determined by the latency between steady-states: the period of time between the selection of a hypermedia link on one web page and the point when usable information has been rendered for the next web page. The optimization of browser performance is therefore centered around reducing this 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 non-redirect 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 [9].

4 MATCHING AN ARCHITECTURE TO ITS STYLE

In an ideal world, the implementation of a software system would exactly match its design. Some features of the modern Web architecture do correspond exactly to their design criteria in REST, such as the use of URI [6] as resource identifiers and the use of Internet media types [18] 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.

HTTP [10] has a central role in determining the capabilities and limitations of the Web architecture. 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 a new header field to identify per-connection control data that is unsafe to be forwarded by intermediaries and an algorithm for canonical treatment of header field digests, but more efficient solutions must wait until an HTTP without backwards compatibility restraints can be deployed.

An example of where an inappropriate extension has been made to the protocol to support features that contradict the desired properties of the generic interface is the introduction of site-wide state information in the form of HTTP cookies [15]. Cookie interaction failed to match REST's model of distributed application state, resulting in substantial confusion for the typical browser application. A cookie could be assigned by the origin server as opaque data, typically containing an array of user-specific configuration choices or a token to be matched against the server's database on future requests. The problem is that a cookie is defined as being attached to any future requests for a given set of resource identifiers, usually encompassing an entire site, rather than being associated with the particular application state (the set of currently rendered representations) on the browser. When the browser's history functionality (the "Back" button) is subsequently used to back-up to a view prior to that reflected by the cookie, the browser's application state no longer matches the stored state represented within the cookie. Therefore, the next request sent to the same server will contain a cookie that misrepresents the current application context, leading to confusion on both sides.

Architectural mismatches are not limited to HTTP. Introduction of "frames" to the Hypertext Markup Language (HTML) caused similar confusion within an application state. Frames allow a browser window to be partitioned into subwindows, each with its own navigational state. Link selections within a subwindow are indistinguishable from normal transitions, but the resulting response representation is rendered within the subwindow instead of the full browser application workspace. This was fine provided that no link exited the realm of information that was intended for subwindow treatment, but as soon as that did occur the user would find themselves viewing one application wedged within the subcontext of another application.

In both these cases, the failure was in providing indirect application state that could not be managed or interpreted by the user agent. A design that placed this information within a primary representation that informed the user agent on how to manage the hypermedia workspace for a specified realm of resources could have accomplished the same tasks without violating the REST constraints, leading to both a better user interface and less interference with caching.

5 RELATED WORK

Garlan and Shaw [13] provide an introduction to software architecture research and describe several "pure" styles. Their work differs significantly from the framework of Perry and Wolf [17] used in this paper 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. [12], where the analysis of when to go mobile depends on active comparison of the size of the code that would be transferred versus the pre-processed information that would otherwise be transferred.

Bass, et al. [3] 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 [1, 12], 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 [16]. However, they exclude various Web resource types or implementation strategies as being "not interesting," when in fact their presence invalidates the assumptions that underlie such models. REST works well because it does not limit the implementation of resources to certain predefined models, allowing each application to choose an implementation that best matches its own needs and enabling the replacement of implementations without impacting the user.

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

The principled use of the REST style in the Web, with its clear notion of components, connectors, and representations, relates closely to the C2 architectural style [22]. 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.

6 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 that are 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 been validated through six years of development of the HTTP/1.0 and HTTP/1.1 standards, elaboration of the URI and relative URL standards, and successful deployment of several dozen independently developed, commercial-grade software systems within the modern Web architecture. 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.

ACKNOWLEDGEMENTS

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

Effort sponsored by the Defense Advanced Research Projects Agency, and Airforce Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Airforce Research Laboratory or the U.S. Government.

REFERENCES

1. Andrews, G. Paradigms for process interaction in distributed programs. *ACM Computing Surveys 23*, 1 (Mar. 1991), pp. 49–90.
2. Anklesaria, F., et al. The Internet Gopher protocol (a distributed document search and retrieval protocol). *Internet RFC 1436*, Mar. 1993.
3. Bass, L., P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.
4. Berners-Lee, T. WWW: Past, present, and future. *Computer 29*, 10 (Oct. 1996), pp. 69–77.
5. Berners-Lee, T., R.T. Fielding, and H.F. Nielsen. Hypertext Transfer Protocol — HTTP/1.0. *Internet RFC 1945*, May 1996.
6. Berners-Lee, T., R.T. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic syntax. *Internet RFC 2396*, Aug. 1998.
7. Chin, R.S., and S.T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys 23*, 1 (Mar. 1991), pp. 91–124.
8. Davis, F., et. al. WAIS interface protocol prototype functional specification (v.1.5). Thinking Machines Corporation, Apr. 1990.
9. Fielding, R.T. Maintaining distributed hypertext infostructures: Welcome to MOMspider's web. *Computer Networks and ISDN Systems 27*, 2 (Nov. 1994), pp. 193¦–204.
10. Fielding, R.T., J. Gettys, J.C. Mogul, H.F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. *Internet RFC 2616*, June 1999. [Obsoletes RFC 2068, Jan. 1997.]
11. Fielding, R.T., E.J. Whitehead Jr., K.M. Anderson, G. Bolcer, P. Oreizy, and R.N. Taylor. Web-based development of complex information products. *Comm. of the ACM 41*, 8 (Aug. 1998), pp. 84–92.
12. Fuggetta, A., G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering 24*, 5 (May 1998), pp. 342–361.
13. Garlan, D., and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), *Advances in Software Engineering & Knowledge Engineering, vol. II*, World Scientific Pub Co., 1993, pp. 1–39.
14. Grønbaek, K., and R.H. Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM 37*, 2 (Feb. 1994), pp. 41–49.
15. Kristol, D., and L. Montulli. HTTP State Management Mechanism. *Internet RFC 2109*, Feb. 1997.
16. Manola, F. Technologies for a Web object model. *IEEE Internet Computing 3*, 1 (Jan.-Feb. 1999), pp. 38–47.
17. Perry, D.E., and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes 17*, 4 (Oct. 1992), pp. 40–52.
18. Postel, J. Media type registration procedure. *Internet RFC 1590*, Nov. 1996.
19. Postel, J., and J. Reynolds. File Transfer Protocol. *Internet STD 9, RFC 959*, Oct. 1985.
20. Sinha, A. Client-server computing. *Communications of the ACM 35*, 7 (July 1992), pp. 77–98.
21. Sollins, K., and L. Masinter. Functional requirements for Uniform Resource Names. *Internet RFC 1737*, Dec. 1994.
22. Taylor, R.N., N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering 22*, 6 (Jun. 1996), pp. 390–406.