Submitted to ESEC/FSE 99 on 01 March 1999

Principled Design of the Modern Web Architecture

Roy T. Fielding and Richard N. Taylor

University of California, Irvine[1]

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 (HTTP) and Uniform Resource Identifiers (URI). We describe the software engineering principles guiding REST, the interaction techniques chosen to retain those principles, and how the chosen techniques compare to those of other architectural styles. We then compare this idealized style to the currently deployed Web architecture in order to elicit mismatches with the existing protocols. Finally, we discuss the lessons learned from using an interaction style to guide the design of a distributed architecture.

1 Introduction

At the beginning of our efforts within the Internet Engineering Taskforce to define HTTP/1.0 [3] and design the extensions for the new standards of HTTP/1.1 [7] and Uniform Resource Identifiers (URI) [4], we recognized the need for a model of how the 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 current architecture could be identified and extensions validated prior to deployment.

The goal of REST is to meet the needs of an Internet-scale distributed hypermedia system by attempting to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations. It does so by placing restrictions on connector semantics where other styles have focused on component semantics, thus enabling the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries.

---

1. *Author's address:* Information and Computer Science, University of California, Irvine, CA 92697-3425. E-mail: {fielding, taylor}@ics.uci.edu

The current Web architecture is only one instance of REST. The style elaborates only those portions of the architecture that are considered essential for Internet-scale distributed hypermedia interaction. When viewed as an overall system, the Web includes access to many different styles of interaction, but the central focus of its protocols and performance constraints is distributed hypermedia. 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 often more efficient to redirect that functionality into a system running in parallel with a more applicable architectural style.

This paper presents REST after the completion of five years of work on architectural standards for the modern Web. In the process, we identify areas where the existing 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 Characteristics of the Application Domain

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

Berners-Lee [2] 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 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. However, attaining them was not without cost. Automatic enforcement of link consistency, a commonly desired feature for hypertext systems [11], was discarded because of the scalability and security concerns of keeping track of references (back-links) across multiple organizational domains. Likewise, the software engineering principles of analyzability and repeatability were sacrificed for the sake of incremental deployment and ephemeral links. Other shortcomings and missed opportunities are discussed in [8].

3 Representational State Transfer (REST)

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. Perry and Wolf [13] distinguish three classes of architectural elements: processing elements (a.k.a., components), data elements, and connecting elements (a.k.a., connectors). 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.

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

Using the software architecture framework of [13], 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.

3.1 Data Elements

Unlike the distributed object style [5], 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 contrast to most distributed processing paradigms ([1], [9]), where it is often more efficient to move the "processing entity" to the data rather than 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. The traditional client/server style (option 1) 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. The mobile object style (option 2) 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 vastly increases the amount of data transferred. The third option 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 enables 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. In addition, the mobile object style can be approximated by sending a representation that consists of instructions in the standard data format of an encapsulated rendering engine (e.g., Java).

3.1.1 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, temporal data (e.g. "today's weather in Los Angeles"), a collection of other resources, a monicker for a non-virtual object (e.g. a person), and so on. 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. 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."

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 validity of the mapping over time (i.e., ensuring that the membership function does not change).

3.1.2 Representations

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, message entity (in HTTP), or variant (in content negotiation).

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. The data format of a representation is known as a *media type*. 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. 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. 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 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 progressively 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 progressively 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: the dimensions of dynamically-sized tables and embedded objects must be determined before they can be rendered, so their occurrence within the viewing area of a hypermedia page will also increase latency.

3.1.3 Metadata

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 structure and semantics of the value are defined by the name. Response messages may include both representation metadata and *resource metadata*: information about the resource that is not specific to the supplied representation.

3.1.4 Control Data

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 behaviour of some connecting elements. For example, cache behaviour can be modified by control data included in the request or response message.

3.2 Connectors

REST uses heavy-weight 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.

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. A component may include both client and server connectors.

Table 1: REST Connector Types

connector	**purpose**
client	satisfies requests for resource actions through a standard interface.
server	listens for requests in order to supply access to services and provide a response.
cache	provides storage of, and access to, responses from prior requests so that they can be reused (if valid) for later requests; reduces latency and network usage.

A third connector type, the *cache* connector, can be located on the interface to a client or server connector in order to save cachable 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. 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 cache can be arranged hierarchically to respond to the shared needs of entire organizations, Internet service providers, or nations. However, it 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 behaviour 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 cachability of a response because the interface is polymorphic rather than specific to each resource. By default, the response to a retrieval

request is cachable and the responses to other requests are non-cachable. 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 cachable by a non-shared cache. A component can override these defaults by including control data that marks the interaction as cachable, non-cachable or cachable for only a limited time.

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.

3.3 Component Types

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

Table 2: REST Component Types

component	purpose	examples
name server	provides partial resolution or intermediate translation of resource identifiers; improves longevity of references through indirection.	DNS URI indirection
origin server	provides interface to services as a resource hierarchy; implementation details are hidden from clients.	Apache httpd Microsoft IIS
gateway	imposed by network or origin server to provide an interface encapsulation of other services, for data translation, performance enhancement, or security enforcement	Squid CGI
proxy	selected by user agent to provide an interface encapsulation of other services, for data translation, performance enhancement, or security protection	CERN Proxy Netscape Proxy Gauntlet
user agent	provides access to WWW information services and renders service responses according to the application needs	Navigator Lynx MOMspider

A *name server* translates partial or complete resource identifiers into the network address information needed to establish an inter-component connection. A *user agent* uses a client connector to initiate a request and becomes the ultimate recipient of the response. 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. 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, whereas a *gateway* (a.k.a., *reverse proxy*) component is an intermediary imposed on the client.

3.4 Architectural Views

Now that we have an understanding of the REST architectural elements in isolation, we can use architectural views [13] 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. However, space limitations require that we focus on the process view and only summarize the others.

3.4.1 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.

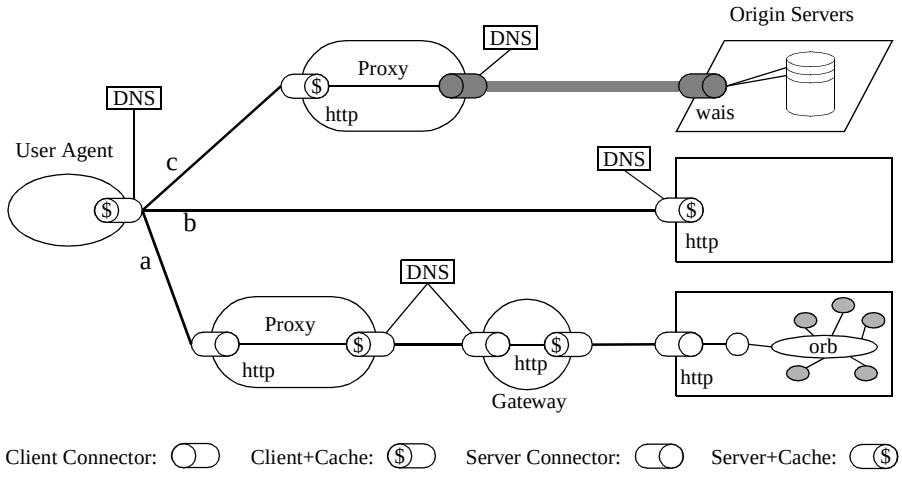

Figure 1 Process View of a REST-based Architecture

Figure 1 provides a sample of the process view from a REST-based architecture. A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according to the properties of each resource identifier and the configuration of the client connector. Request (a) has been sent to a local proxy,

which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by a hidden 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 yet can be accessed through 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.

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 the requested resource (e.g., the "GET" method in HTTP), which can often be cached for later reuse.

3.4.2 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 interface between the components and their client or server connectors.

Client connectors examine the resource identifier in order to select the appropriate communication mechanism for each request. Unlike most architectural styles, REST allows many different protocols to be in use within the same architecture. Each interaction uses a protocol corresponding to some aspect of the identifier for the requested resource. For example, interaction with a name server (e.g., DNS) will use whatever protocol has been configured for use with that type of name. Likewise, a client may be configured to connect to a specific proxy component in order to make requests for a given set of resource identifiers.

REST interaction exhibits properties of both the client/server and heartbeat styles [1]. The client/server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. The heartbeat style of interaction removes the need for an awareness of the overall component topology (an impossible task for an Internet-scale architecture) and allows components to act as either destinations or intermediaries, determined dynamically by the target of the request.

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. Connectors need only be aware of each other's existence during the scope of their connection. (A connector may cache the existence and capabilities of other components for performance reasons.)

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, Gopher, and WAIS. However, the interaction with all of these services is restricted to the semantics of a REST connector, even when the protocol is not specifically designed to support it. This restriction 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 currently interfaces with TELNET and "mailto" resources.

3.4.3 Data View

A data view of an architecture reveals the application state as information flows through the components. Since REST is specifically targeted to distributed information systems, we view 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. A REST-based architecture is capable of supporting many different types of application, but certainly the most prominent one is a browser in the process of viewing a hypermedia information service.

REST concentrates all of the control state into the representations received in response to interactions. An application's state is therefore defined by its pending requests, the topology of connected components (some of which may be filtering), 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 an application 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.
- Rendering the first non-redirect response representation becomes the next point in the critical path. Latency can be considerably reduced if the media type is capable of progressive rendering, since then the representation can be rendered as it is being received rather than after the response has been completed. Progressive rendering is, in turn, impacted by the availability of layout information (style sheets and the 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 [6].

4 Matching an Architecture to its Style

In an ideal world, the implementation of a software system would exactly match its design. REST provides a model against which the actual Web architecture can be compared, but the reality is that many of the current architecture's design decisions were based on constraints outside the architectural style. Other features, such as the use of URI [4] as resource identifiers and the use of Internet media types to identify representation data formats, are entirely consistent with the REST style.

HTTP [7] 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. 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. Fixing these limitations will require incompatible changes to the protocol.

However, there are also areas where the design of HTTP has been unfairly criticized due to a lack of appreciation for its architectural constraints, and where inappropriate extensions have been made to the protocol to support features that contradict the desired properties of the generic interface. For example, the introduction of stateful information

in the form of HTTP cookies 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 cookies were 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) was later used to back-up to a view prior to that reflected by the cookie, the browser's application state would no longer match the assumed state represented within the cookie.

5 Related Work

REST draws from several sources. On the one hand, REST draws from the tradition of client/server architectures, yet with the major difference that the deep interactions between the client and server are not revealed by a classical API. The payload of an HTTP request may contain significant metadata that determines what actions the server will take in response to the message.

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 [12]. However, such efforts usually attempt to exclude various 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, and therefore allows each application to choose an implementation that best matches its own needs, and allows implementations to be replaced without impacting the user.

The key characteristic of the REST style, sending representations of resources to consuming components, may be seen to have some basis in the long tradition of event-based integration schemes. These schemes include Smalltalk's model-view-controller paradigm, Reiss's Field system, HP's Softbench, and the like. The key difference is that all those schemes 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, the consuming components always "pull" the representations across.

Naturally, the advantages of a push model are lost in the REST style, but it is clear that, with the scale of the Web, an unregulated push model presents numerous scaling problems. It is interesting to note, however, that several projects are currently exploring how to establish a push model in the web, and hence gain the benefits of reactive execution rather than polling [14].

The principled use of the REST style in the Web, with its clear notion of components, connectors, and representations, also shows significant overlap with current work in the software architecture community, particularly the C2 style [14]. The C2 style supports the development of distributed, dynamic applications by focusing on structured use of heavy-weight connectors to obtain substrate independence. C2 applications

rely on asynchronous notification of state changes (representations) and request messages. As with other event-based schemes, C2 is nominally push based, though a C2 architecture could easily operate in REST's pull style by only emitting a notification upon receipt of a request. The C2 style does not have intrinsic support for optimization of topologies in response to patterns of usage (e.g., caching).

6 Conclusions and Future Work

The World Wide Web is arguably the world's largest distributed application. It is also highly dynamic and successful in the extreme. Understanding the key architectural principles underlying the Web can help explain its technical success. Such understanding is the basis for achieving similar success in other distributed applications, particularly those that might be amenable to the same style.

For network-based applications such as the Web, system performance is dominated by network communication. The performance cost of interactions across a wide-area network must be balanced against the performance gains of distributed processing of the actions. For a distributed hypermedia system, the actions are predominantly large-grain data transfers rather than computation-intensive tasks. The REST style was developed in response to these needs; its focus upon flexible topologies, transmission of representations of resources, connector interfaces, and heavy-weight connectors has enabled its unparalleled success.

The benefit of drawing correspondences between REST and other architectural styles is that cross-fertilization can occur. REST suggests several ways in which, e.g., C2 architectures could be optimized. In return C2 and other architecture results demonstrate how one could reason about REST-based applications, including the Web. C2 and other push technologies have demonstrated significant benefits derivable from event notification. The models, techniques, and optimizations developed in other architecture work can be used to guide future evolution of the Web. Architecture-based support for principled evolution of applications on the fly also provides a model for guiding complex network-based applications. We have just begun to explore such possibilities.

7 Acknowledgments

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 Internet Engineering Taskforce (IETF). We would particularly like to thank Tim Berners-Lee, Henrik Frystyk Nielsen, Dan Connolly, Rohit Khare, Jim Whitehead, Larry Masinter, and Dan LaLiberte for many thoughtful conversations regarding the nature and goals of the WWW architecture.

8 References

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

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

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

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

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

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

[7] R.T. Fielding, J. Gettys, J.C. Mogul, H.F. Nielsen, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. *Internet RFC 2068*, Jan. 1997.

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

[9] A. Fugetta, G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering 24*, 5 (May 1998), pp. 342-361.

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

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

[12] F. Manola. Technologies for a Web object model. *IEEE Internet Computing 3*, 1 (Jan.-Feb. 1999), pp. 38-47.

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

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