From Representations to Computations: The Evolution of Web Architectures

Justin R. Erenkrantz
jerenkra@ics.uci.edu

Michael M. Gorlick
mgorlick@acm.org

Girish Suryanarayana
sgirish@ics.uci.edu

Richard N. Taylor
taylor@ics.uci.edu

Institute for Software Research
University of California, Irvine

ABSTRACT

REpresentational State Transfer (REST) guided the creation and expansion of the modern web. What began as an internet-scale distributed hypermedia system is now a vast sea of shared and interdependent services. However, despite the expressive power of REST, not all of its benefits are consistently realized by working systems. To resolve the dissonance between the promise of REST and the reality of fielded systems, we critically examine numerous web architectures. Our investigation yields a set of extensions to REST, an architectural style called Computational REST (CREST), that not only offers additional design guidance, but pinpoints, in many cases, the root cause of the apparent dissonance between style and implementation. Furthermore, CREST explains emerging web architectures (such as mashups) and points to novel computational structures.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design

Keywords

Representational State Transfer, mobile code, network continuations, web services

1. INTRODUCTION

REpresentational State Transfer (REST) is an architectural style that characterizes and constrains the macro interactions of the active elements of the web—its servers, caches, proxies, and clients. However, REST is silent on the architecture of the individual participant; that is, the components, relationships, and constraints *within* a single active participant. We explore the thesis that to maintain the fidelity of REST's principles at the level of the web requires previously unspecified constraints on both the architecture of those individual participants and the system-level architecture of the web itself.

We pursue three related lines of investigation. We first draw upon our experience as developers struggling to build web applications that conform to the REST style. Here we discover both the consequences of failing to hew to the constraints of REST and how participant architectures (on the scale of a single element) must be rearranged to align with REST's goals. We then turn to emerging web services to evaluate their fit with REST principles. Finally, we examine recent innovations in web practices to better understand the role of REST in supporting novel forms of web interaction and service composition.

These three lines of inquiry lead to a deeper understanding and broadening of the fundamental REST principles, resulting in a new architectural style: Computational REST (CREST). However, that elaboration is but an example of a more general form—network continuations—the exchange of the representations of the execution state of distributed computations. It is the presence and exchange of continuations among web participants, in their various forms, that induces new constraints among and within participants. With this in mind, both prior complications in the structure of individual clients and elaborations of the web such as AJAX or mashups are accounted for by a single fundamental mechanism: network continuations as web resources—an insight codified in the principles of CREST.

The remainder of the paper is structured as follows. Section 2 recaps the REST style. Section 3 relates our experiences in creating REST-based web applications. Section 4 explores in detail the uneasy fit between REST and web services, while Section 5 reviews web applications that stretch the boundaries of the REST style. Section 6 reflects on the lessons one might draw from these varied applications and experiments, while Section 7 examines related work. Section 8 introduces the CREST architectural style and details its influence on web architecture "in-the-large" and participant architecture "in-the-small"; Section 9 revisits the analyses of Sections 3-6, but this time from the perspective of CREST. Finally, Section 10 reviews our contributions and sketches our plans for future work.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

2. WHAT IS REST?

The REpresentational State Transfer (REST) architectural style [9, 10] governs the proper behavior of participants on the World Wide Web. In a typical REST interaction on the modern Web, a user agent (say, a web browser, such as Mozilla Firefox) requests a representation of a resource (web page, such as HTML content) from an origin server (web server, such as Apache HTTP Server), which may pass through multiple caching proxies (such as Squid) before ultimately being delivered.

REST elaborates those portions of the web architecture devoted to interaction with Internet-scale hypermedia [9]. In that context, REST's goal is to reduce network latency while facilitating component implementations that are independent and scalable. Instead of focusing on the semantics of components, REST places constraints on the communication between components. There are six core REST design principles:

RP1 *The key abstraction of information is a resource, named by an URL.* Any information that can be named can be a resource: a document or image, a temporal service (e.g., "today's weather in Dubrovnik"), a collection of other resources, a moniker for a nonvirtual object (e.g., a person), and so on.

RP2 *The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes.* Hence, REST introduces a layer of indirection between an abstract resource and its concrete representation. Of particular note, the particular form of the representation can be negotiated between REST components.

RP3 *All interactions are context-free.* This is not to imply that REST applications are without state, but that each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

RP4 *Only a few primitive operations are available.* REST components can perform a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST—for instance, all resources exposed via HTTP are expected to support each operation identically.

RP5 *Idempotent operations and representation metadata are encouraged in support of caching.* Caches are important to the goal of reducing latency. The metadata included in requests and responses permits REST components (such as user agents, caching proxies) to make sound judgements of the freshness and lifespan of representations. Additionally, the repeatability (idempotence) of specific request operations (methods) permits representation reuse.

RP6 *The presence of intermediaries is promoted.* Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the client and the origin server.

3. DISSONANCE: APPLICATIONS

Despite their apparent simplicity, the REST principles have proven difficult to understand and apply consistently, leaving many web systems without the benefits promised by REST, including sites that do not support caches or require stateful interactions. To better understand why, we first recount our personal experiences building and repairing two web systems: `mod_mbox`, a service for mail archiving, and `Subversion`, a configuration management system. In Section 4, we turn our eye toward other web systems, in which we had no personal involvement, and discover similar confusion there in the practical application of REST. Section 5 examines REST from an entirely different perspective: web applications that stretch the boundaries of REST while remaining true to its core principles.

3.1 mod_mbox

As core contributors to the open-source Apache HTTP Server, we required a scalable, web-based mail archive to permanently record the design rationales and decisions made on the project mailing lists. At the time, no archivers fully met our requirements. Some systems, such as MHonArc [18], converted each incoming message into HTML and recomputed the list index as the messages arrived. Constant index regeneration is problematic for high-traffic archives; ours serves almost 500 lists, some of which receive over 1,000 messages a month and have been active for over a decade. For any server hosting such a large volume of message traffic, constantly regenerating the message index is wasteful and expensive. Given the scale of the Apache archive, caching and dynamic computation of the list index are of critical importance (RP5, RP2).

Other web-based archivers, such as Eyebrowse [5], generated message links whose ordering reflected the sequence in which messages were loaded into the archive. If we regenerated the archive (following hardware failure or other corruption), any previous links would either be stale or refer to a different message. Conseqently, persistent hypertext links—such as those from our own code or other emails—might be broken after the archive was regenerated. The key abstraction of information in REST, a resource named by an URL, would not be consistent after an outage or upgrade of the archive (RP1).

Informed by our experience with the Apache HTTP Server, we were confident that we could create a new web-based archiver, mod_mbox [2], based on the REST principles, that would not suffer from these shortcomings. However, as we discovered, REST alone was neither sufficiently expressive nor definitive. `mod_mbox` required two additional constraints beyond those dictated by REST: dynamic representations of the original messages and the definition of a consistent namespace.

Instead of creating HTML representations as messages arrive, mod_mbox delays that transformation until a request for a specific message is received. On arrival, only a metadata entry is created for a message M. Only later, when message M is fetched from the archive, does an Apache module create an HTML representation of M (with the help of $M's$ metadata entry). This sharp distinction between the resource and its representation (RP1, RP2) minimizes the up-front computational costs of the archive—allowing mod_mbox to handle more traffic. To achieve a consistent namespace, mod_mbox relies upon M's metadata

(the Message-ID header). Consequently, if the metadata index is recreated, the URLs of the resources (messages) remain constant—guaranteeing the long-term persistence of links. After adopting these constraints, mod_mbox scaled to archive all of the mailing lists for The Apache Software Foundation (in excess of 2.5 million messages to date) with consistent and persistent links.

3.2 Subversion

Subversion [4], a source code manager designed to be a compelling replacement for the popular CVS, made a decision early on to use WebDAV, a set of extensions to HTTP, for repository updates and modifications. Hence, by conforming to the REST constraints, we expected that Subversion's implementation would support caching and the easy construction of intermediaries (RP5, RP4, RP6). Following these constraints, a REST-based versioning client (in essence, a user agent) must fetch resources from an origin server. For practicality, in order to limit the number of connections and minimize network overhead, a Subversion client reuses a small number of connections. Since the client must make multiple requests for distinct resources on the same connection, network latency can dominate performance. This problem was anticipated in the early days of standardizing the HTTP protocol, but was not clearly articulated within REST; instead "pipelining"—where clients issue multiple requests without waiting for responses—was simply recommended. However, lacking detailed design guidance, Subversion developers, failing to appreciate the performance penalty, did not implement pipelining and fetched resources serially. Unsurprisingly, the checkout performance turned out to be unacceptable at first.

To improve serial performance and reduce network roundtrip latency, Subversion implemented a *custom* WebDAV method. Rather than request each resource individually, the new client issued a bulk request for all needed resources. As expected, this change improved overall network utilization and reduced latency. However, this custom WebDAV method required that all of the data (including the content stored in Subversion—which, of course, could be binary and of arbitrary size) be XML-encoded. The encoding increased transfer volumes by approximately 33% [24], further clogging the network. Rather than adhering to the small set of operations prescribed by HTTP (RP4) and implementing pipelining, a custom operation was added. Consequently, simple intermediaries could no longer be deployed to ease the load on an upstream Subversion server as they would not understand this custom WebDAV method (RP6).

Subversion developers (including one of the authors) undertook to write an alternate client framework, `serf`, that adhered to REST principles and constraints and implemented pipelining along with asynchronous requests and responses. Internally, `serf` adopted two mechanisms: chained data transforms (buckets) and nonblocking network connections [8]. "Buckets" are independent data streams to which successive transforms are applied on-the-fly, allowing the client to delay transforms until needed. Nonblocking connections improved network efficiency and reduced latency, as the buckets never had to wait to write or read data. By decoupling communication and transformation, Subversion clients could now efficiently exploit pipelining. Reducing latency obviated the need for a custom WebDAV method (RP4). This, in turn, eliminated the overhead of XML encoding and permitted the reintroduction of simple caching intermediaries (RP6). In the `serf` framework, the new Subversion client scales gracefully yet adheres to the REST constraints.

```
POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml
Content-Length: ...
<?xml version="1.0" ?>
<env:Envelope...>
   <env:Body>
      <m:retrieveItinerary...>
         <m:reference>FT35ZBQ</m:reference>
      </m:retrieveItinerary>
   </env:Body>
</env:Envelope>
```

(a) SOAP example (modified from Example 12a in [23])

```
GET /Reservations/itinerary?id=FT35ZBQ HTTP/1.1
Host: travelcompany.example.org
```

(b) REST example (modified from Example 12b in [23])

Figure 1: A SOAP example and its REST-compliant equivalent.

4. DISSONANCE: WEB SERVICES

"Web services" have emerged to expose finer-grained computational resources than the coarser-grained content resources of the traditional web. There are two popular approaches to web services: one that claims to conform to REST principles, and another that relies upon SOAP [22]. Figures 1a and 1b compare these two approaches. One clear distinction between the two is their use of resources. The REST-compliant request of Figure 1b leverages resources by requesting virtual resources within the server namespace (RP1), while the SOAP request of Figure 1a adds a level of indirection to the naming of a resource through an XML entity.

4.1 Web Services: SOAP

Tracing its origins from the prior XML-RPC specification [31], Simple Object Access Protocol (SOAP) was introduced as a lightweight "protocol" for exchanging structured information on the web [16]. However, SOAP is not a protocol but a descriptive language transferred via a transport protocol such as HTTP. In practice, SOAP corrupts the integrity of the REST architecture, and most of the problems can be traced to architectural mismatches with REST and HTTP; two such problems are highlighted here. A deeper analysis of SOAP may be found in our prior work [7].

As discussed earlier, idempotent operations (RP5) are a fundamental REST concept. Some HTTP methods (such as `GET`) must be idempotent: if a `GET` is performed multiple times on a static resource, the results must be identical. Other methods (such as `POST`) are non-idempotent: if a `POST` is performed multiple times on the same resource, the side effects are undefined by the specification.

In practice, most SOAP interactions employ the `POST` method of HTTP. Consequently, no intermediary (RP6) can know whether a service call will be idempotent without specific semantic knowledge of the actions taken by the SOAP re-

Intent	**HTTP Protocol** (adapted from [19])
User identifies self via a form	`POST /acme/login HTTP/1.1` *form data*
Server sends a cookie to reflect identity	`HTTP/1.1 200 OK` `Set-Cookie: CUST-ID="WILE_E_COYOTE"; path=/; expires=Sun, 17-Jan-2038 12:00:00 GMT;`

User returns cookie in a later request	`POST /acme/pickitem HTTP/1.1`
	`Cookie: CUST-ID="WILE_E_COYOTE"`
	form data

Figure 2: HTTP Cookie Example

ceiver. Within REST the protocol alone defines whether the operation is idempotent, without any relationship to the resource. The (apparent) lack of idempotency in SOAP interactions is a significant obstacle to intermediaries that wish to intelligently cache SOAP messages sent over HTTP.

Tunneling a separate SOAP envelope within HTTP also violates REST's separation of data and metadata (RP2). SOAP encapsulates the entire message—including all SOAP metadata—in the body of the HTTP request as an XML entity. In order to properly parse (and understand) the request, a SOAP intermediary must examine the *entire* body of the HTTP request. In contrast, an HTTP proxy need only examine the metadata found in the request headers and can pass the request body through without any inspection, as it is irrelevant for routing or caching. By hiding the routing and metadata for SOAP inside the body of the HTTP request, a SOAP intermediary must peek inside the body to ensure proper routing—a clear violation of RP2, the strict separation of metadata (the HTTP headers) and representation (the SOAP message payload).

4.2 Web Services: Mixed APIs

Many content providers expose REST-compliant APIs for their products. As there is no commonly accepted litmus test for REST compliance, we term those services that explicitly acknowlege REST in their description as "RESTful." eBay provides an API to query current auctions [6], the Flickr API implements uploads and photo tagging [32], and Amazon.com facilitates access to their centralized storage repository [1]. While these RESTful services have not yet completely displaced their SOAP counterparts, they do have well-known advantages. In a 2003 talk, an Amazon spokesperson mentioned that 85% of their service requests employed their RESTful API, not SOAP [25], and that querying Amazon using REST was roughly six times faster than with the SOAP equivalents [28].

Despite their prevelance and popularity, these implementations of REST services are uneven at best—especially in comparison to other alternatives from the same provider. eBay's REST API is extremely limited, permiting just read-only operations (such as queries). More advanced functions, such as posting new items, are available only through their SOAP API. Others have REST interfaces that are roughly equivalent to their SOAP counterparts; with Flickr, photos can be uploaded, tagged, and deleted in both interfaces with only minimal differences. Finally, there are examples, such as Amazon's S3, for which their RESTful APIs are a superset of the SOAP counterpart.

Overall, we observe that the closer the service semantics are to those of content, the more likely the service is to have a rich REST API. For Amazon's storage service, a mapping to the content-centric nature of REST is straightforward and free of complications. REST principles RP4 and RP5are well-preserved in Amazon's interface. On the other hand, eBay's service model is strongly tilted toward SOAP. How to explain the differences? In part, the division between REST and SOAP may reflect a lack of design guidance; how can services that are not content-centric, such as auction bidding (which are prominently displayed in eBay's SOAP interface but absent from the REST interface) be cleanly constructed in a REST-consistent manner? We opine that web services whose focus is nontraditional resources are clearly underserved with REST alone and that their developers lack adequate design guidance. Consequently, it is unsurprising that service providers offer alternatives to fill this gap.

4.3 Cookies

Besides RESTful APIs and SOAP, cookies [19] are another common mechanism employed by developers to support services that span more than one request. Comparatively lightweight, cookies are a means for a site to assign a "session" identifier to a "user." To start the process (illustrated in Figure 2), an origin server provides a cookie in response to a request via the "Set-Cookie" HTTP header. Inside of this cookie are several attributes, the most important of which is an opaque identity token, a path representing where the cookie should subsequently be presented, and an expiration date.

Cookies have poor interactions with the history mechanisms of a browser [9]. Once a cookie is received by a user agent, the cookie should be presented in all future requests to that server until the cookie expires. Therefore, cookies do not have any direct relationship to the sequence of requests that caused the cookie to be set in the first place. Hence, if the browser's back button is used to navigate in the page history prior to the initial setting of the cookie, the cookie will still be sent to the server. This can lead to a "cookie that misrepresents the current application context, leading to confusion on both sides" [9].

4.4 Web Services: Challenges

The formulation and development of web services has been a complex undertaking with vigorous discussion within the community as to whether such services are even feasible [30]. Service-oriented architectures have emerged, in part, as a response to the problems of service compostion; however, the composition and encapsulation of web services still awaits resolution [30]. We next take a look at ways in which composition and encapsulation have been addressed through REST—albeit not under the traditional banner of web services.

5. AJAX AND MASHUPS

Emerging classes of Web applications extend the notion of a REST interface in interesting ways. One such example is Google Maps [14], which employs an application model known as AJAX [13], consisting of XHTML and CSS, a browser-side Document Object Model interface, asynchronous acquisition of data resources, and client-side Javascript.

AJAX expands on an area for which REST is deliberately silent—the interpretation environment for delivered content—as content interpretation and presentation is highly content- and application-specific. Browsers have long accomodated helper applications that are executed when "unknown" content arrives. However, instead of running the helper in a different execution environment, AJAX blurs the distinction between browser and helper by leveraging client-side scripting to download the helper application dynamically and run it within the browser's execution environment.

Dynamically downloading the code to the browser moves the computational locus away from the server. Instead of performing computations solely on the server, some computations (for example, presentation logic) can now be executed locally. By reducing the computational latency of presentation events, AJAX makes possible a new class of interactive applications with a degree of responsiveness that may be impossible in purely server-side implementations.

The innovation of AJAX is the transfer, from server to client, of a computational context whose execution is "resumed" client-side. Thus, we begin to move the computational locus away from the server and onto other nodes. REST's goal was to reduce server-side state load; in turn, AJAX reduces server-side computational load. AJAX also improves responsivity since user interactions are interpreted (via client-side Javascript) at the client. Thus AJAX, respecting all of the constraints of REST, expands our notion of resource.

Mashups, another computation-centric REST-based service, are characterized by the participation of at least three distinct entities:

- A web site, the mashup host M
- One or more web sites, $c_1, c_2, \ldots, c_n$, $c_i \neq M$, the content providers
- An execution environment E, usually a client web browser

The mashup host M generates an "active" page P for the execution environment E. P contains references (URLs) to resources maintained by content providers $c_1, c_2, \ldots, c_n$ and a set of client-side scripts in Javascript. The client-side scripts, executing in E, interpret user actions and manage access to, and presentation of, the combined representations served by $c_1, c_2, \ldots, c_n$.

Mashups where Google Maps is one of the content providers c_i are especially popular; examples include plotting the location of stories from the Associated Press RSS feed [33], and Goggles, a flight simulator [3].

Mashups offer a fresh perspective on REST intermediaries. Conventionally, an intermediary intercepts and interprets all request and response traffic between client and server. In contrast, the mashup host M (the origin server), constructs a virtual "redirection" comprising a set of client-side scripts that reference resources hosted elsewhere at web sites (content providers) c_i, $c_i \neq M$. Thereafter, the client interacts only with the content providers c_i. Thus, mashup host M "synthesizes" a virtual compound resource for the client. Though a mashup could be implemented entirely server-side by M, it would increase server load and (for highly interactive mashups) latency. Mashups illustrate both the power of combining multiple resources under computational control to synthesize a new resource and the benefits of moving the locus of computation away from the origin server.

6. LESSONS FROM REST EXPERIENCES

What lessons can we draw from the spectrum of REST experiences? `mod_mbox` demonstrates the critical importance of the structure of the namespace (URL) in REST transactions and the value of decoupling resources from representations. The saga of Subversion speaks to us on a different level; i.e., the internal architecture of web participants. It was not possible to fully align Subversion with REST principles until Subversion clients embraced asynchronous (nonblocking) network transfers and "just-in-time" data transforms that together minimized latency. This suggests that the benefits of REST may be difficult to realize unless the individual web participants align their internal architectures to accomodate both asynchronous communications and concurrent computations.

Web services, in the guise of SOAP-mediated exchanges, not only violate numerous REST principles (RP1, RP4, RP5, RP6), but suffer from poor performance, exacerbate latency, fail to scale, and exhibit a high degree of complexity. Since SOAP cannot compose services, cumbersome higher-order services (such as WS-Workflow) must be layered above it to remedy the deficiency. Even so, these services do not directly support a form of discovery that is amenable to web crawling, the strategy employed under HTTP to discover and catalog web pages.

In sharp contrast, modern REST extensions, such as AJAX and mashups, suggest a pivotal, and perhaps unappreciated, role for mobile code in greatly expanding the scope and subtlety of REST interactions. AJAX employs server-generated code that is transferred client-side to inject a degree of application "responsivity" that is difficult to achieve server-side. Mashups also illustrate the utility of code transfer from server to client to implement resource fusion—a complex task that is easier done computationally than declaratively.

REST guided the early development of the web and is a unified architectural model that explains much of the behavior of the modern web. However, there are many issues on which REST is either silent or fails to adequately address. The examples of Section 3, `mod_mbox` and `Subversion`, demonstrate that even experienced developers falter when building applications that conform to REST constraints. Section 4 illustrated the broad range of interpretations major services apply in promulgating "REST-compliant" application programming interfaces. Finally, Section 5 offered examples of surprising and novel REST extensions that were not anticipated when REST was first articulated.

With these examples in mind, we reexamine REST, reformulating and expanding the core REST principles and constraints to accommodate the recent evolution of the web. REST addresses Internet-scale hypermedia, but we can now see the web in a different light, where computational ex-

change, rather than content exchange, dominates web activity.

7. COMPUTATATIONAL EXCHANGE

Both AJAX and mashups employ a primitive form of mobile code (Javascript embedded in resource representations) to expand the range of REST exchanges. However, far more powerful forms of *computational exchange*, based on a combination of mobile code and continuations, have been demonstrated. A *continuation* is a snapshot (representation) of the execution state of a computation such that the computation may be later resumed at the execution point immediately following the generation of the continuation. Continuations are a well-known control mechanism in programming language semantics: many languages, including Scheme, Smalltalk, and Standard ML, implement continuations.

We borrow liberally from a rich body of prior work on mobile code and continuations to articulate our view of web-centric computational exchange. An excellent survey and taxonomy of mobile code systems may be found in [12] and, in particular, there are several examples of mobile code implementations based on Scheme. Halls' `Tubes` [17] explores the role of "higher-order state saving" (that is, continuations) in distributed systems. Using Scheme as the base language for mobile code, Tubes provides a small set of primitive operations for transmitting and receiving mobile code among Tubes sites. Tubes automatically rewrites Scheme programs in continuation-passing style to produce an implementation-independent representation of continuations acceptable to any Scheme interpreter or compiler. Halls demonstrates the utility of continuations in implementing mobile distributed objects, stateless servers, active web content, event-driven location awareness, and location-aware multimedia.

`MAST` [29] is also a Scheme variant for distributed and peer-to-peer programming that introduces first-class distributed binding environments and distributed continuations (in the spirit of Tubes) accompanied by a sound security model. Like Tubes, MAST also provides primitives for mobility. MAST offers a developer fine-grain network control while supplying potent control and execution primitives.

Both Tubes and MAST achieve "computation mobility," the movement of "live code" from one network host to another. Other language bases are also feasible. Tarau and Dahl [27] achieve the same for a logic programming language `BinProlog`, again employing serialized continuations that are transferred from one host to another and then reconsituted.

Mobile objects are a weaker form of computation mobility. Scheme appears in this context as `Dreme` [11], in pursuit of distributed applications with little concern for process or network boundaries. There, extensions to Scheme include object mobility, network-addressable objects, object mutability, network-wide garbage collection, and concurrency. Dreme also includes a distributed graphical user interface that relied upon continuations, rather than even-driven programming, to manage the interface and respond to user interactions and network events.

Continuations have an important role to play in many forms of web interactions and services. For example, Queinnec [26] demonstrates that server-side continuations are an elegant mechanism to capture and transparently restart the state of ongoing evolving web interactions; in other words, server/client interactions are a form of "web computation," (represented by a program evaluated by the server) for which continuations are required to suspend and resume stateful page tours or service-oriented sessions that are client-parameterized but generated server-side.

Matthews et al. [20] extend this work, offering a set of automated transformations based on continuation-passing style, lambda lifting, and defunctionalization that serialize the server-side continuation and embed it in the web page returned to the client. When the client responds to the web page the (serialized) continuation is returned to the server where it is "reconstituted," with the server resuming execution of the continuation. This is an example of computational exchange (from server to client and back again) that preserves context-free interaction and allows the server to scale by remaining largely stateless.

Finally, motivated by the richness of web interactions, browsing through data- and decision-trees, bookmarking, and backtracking, Graunke and Krishnamurthi [15] explore bringing the same interaction techniques to non-web graphical user interfaces. They describe transformations, based on the continuation-passing style, that confer the power and flexibility of web interactions on graphical user interfaces.

8. CREST

We see the web realigning, from applications that are content-centric to applications that are computation-centric: where delivered content is a "side-effect" of computational exchange. In the computation-centric web the goal is the distribution of service and the composition of alternative, novel, or higher-order services from established services.

Drawing from the prior work enumerated in Section 7 we explore a computation-centric web in which Scheme is the language of computational exchange and Scheme programs, closures, continuations, and binding environments are the responses to web requests. Further, AJAX and mashups, as detailed in Section 5, illustrate the power of computation, in the guise of mobile code, as a mechanism for framing responses as interactive computations (AJAX) or for "synthetic redirection" and service composition (mashups). Raising mobile code to the level of a primitive among web peers and embracing continuations as a principal mechanism of state exchange permits a fresh and novel restatement of all forms of web services, including serving traditional web content, *and suggests the construction of new services for which no web equivalent now exists.*

In the world of computational exchange, an URL denotes a computational resource. There, clients issue requests in the form of programs (expressions) e, origin servers evalute those programs (expressions) e, and the value v of that program (expression) e is the response returned to the client. That value (response) v may be a primitive value (`1`, `3.14`, or `"silly"` for example), a list of values `(1 3.14 "silly")`, a program (expression), a closure, a continuation, or a binding environment (a set of name/value pairs and whose values may include (recursively) any of those just enumerated).

Under computational exchange the putative role of SOAP is an oxymoron, service discovery can be a side-effect of execution, and service composition reduces to program (expression) composition. For example, the program (expression) issued by a client C to an URL u of origin server S

```
(if (defined? 'word-count)
    (word-count (GET "http://www.yahoo.com")))
```

(rendered in the concrete syntax of Scheme) tests the execution environment of S for a function `word-count` (service discovery) and if the function (service) is available, fetches the HTML representation of the home page of `www.yahoo.com`, counts the number of words in that representation (service composition), and returns that value to C.

To provide developers concrete guidance in the implementation and deployment of computational exchange we offer Computational REST (CREST) as an architectural style to guide the construction of computational web elements. There are five core CREST principles:

CP1 *The key abstraction of computation is a resource, named by an URL.* Any computation that can be named can be a resource: word processing or image manipulation, a temporal service (e.g., "the predicted weather in London over the next four days"), a generated collection of other resources, a simulation of an object (e.g., a spacecraft), and so on.

CP2 *The representation of a resource is a program, a closure, a continuation, or a binding environment plus metadata to describe the program, closure, continuation, or binding environment.* Hence, CREST introduces a layer of indirection between an abstract resource and its concrete representation.

CP3 *All computations are context-free.* This is not to imply that applications are without state, but that each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it. Prior representations can be used to transfer state between computations; for example, a continuation (representation) provided earlier by a resource can be used to resume a computation at a later time merely by presenting that continuation.

CP4 *Only a few primitive operations are always available, but additional per-resource operations are also encouraged.* Participant A sends a representation p to URL u hosted by participant B for interpretation. These p are interpreted in the context of operations defined by u's specific binding environment. The outcome of the interpretation will be a new representation—be it a program, a continuation, or a binding environment (which itself may contain programs, continuations, or other binding environments). Of note, a common set of primitives are expected to be exposed for all CREST resources, but each u's binding environment may define additional resource-specific operations.

CP5 *The presence of intermediaries is promoted.* Filtering or redirection intermediaries may also use both the metadata and the computations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the client and the origin server.

8.1 CREST Guidelines

As the REST experience demonstrates, it is insufficient to merely enumerate a set of architectural principles; concrete design guidance is also required. To this end, we explore some of the consequences of the CREST principles, cautioning that the discussion here is neither exhaustive nor definitive. Nonetheless, it draws heavily upon both our experiences as implementors of web services and web clients and the lessons of the analyses of Sections 2-7.

8.1.1 Computational namespaces

Under CREST, URLs name computations (CP1)—in the near-literal sense that the program, closure, continuation, or binding environment transmitted from client to origin server is physically embedded in the URL. For example, if a is the ASCII text of a program p sent by client c to URL $P : //S/u_0/\ldots/u_{m-1}/$ of origin server S under scheme P then the URL used by c is $P : //S/u_0/\ldots/u_{m-1}/a/$. Closures, continuations and binding environments may contain arbitrary recursive structures. There a serialized representation is required, a detail that we ignore for the sake of brevity.

CREST URLs are not intended for human consumption, as they are the base mechanisms of computational exchange among CREST peers; human-readable namings may be provided as needed by higher layers. Among computational peers, the length of the URL or its encoding is irrelevant and ample computational and network resources are readily available among modern peers to assemble, transmit, and consume URLs that are tens of megabytes long. In effect, the URL $u = P : //S/u_0/\ldots/u_{m-1}/$ is the root of an infinite virtual namespace of all possible finite programs (or closures, continuations, and binding environments) that may be interpreted by the interpreter denoted by u.

From the perspective of a client, binding a specific mobile program p to a specific URL u reduces code mobility to a triviality since code motion and URL construction are now one and the same. It simultaneously elevates the transparency of the exchange since such an URL $u' = P : //S/u_0/\ldots/u_{m-1}/a/$ may be inspected or modified by intermediaries (CP5). Finally u', a moderately compact and host-independent representation of a computational exchange, may be recorded and archived for reuse at a later point in time (CP3).

8.1.2 Transparency of computation

When URLs name computational resources, the burden of managing the namespace shifts from a generic container (such as a file system in the case of REST) to the binding environment of the interpreter named by the URL. REST, being content-centric, is completely silent on the interpretation (meaning) placed upon the URL by the origin server. CREST, in contrast, is computation-centric and the "meaning" of an URL is a full interpreter whose binding environment is far richer, more finely-grained, and nuanced than that of any single web REST URL, even one that accepts a large number of parameters (CP4). Consequently, the range of metadata and query or advisory functions that a CREST-based interpreter may offer can far outstrip that available to an HTTP client from an HTTP origin server (CP2). For example, content negotiation becomes an active computational process under the explicit control of the client since the negotiation is simply a program (CP2, CP3), composed and dispatched by the client to the server, that may inspect the binding environment for various representations and evaluate them according to client-specific critera or invoke URL-specific functions (CP4) to generate representations that are custom-tailored to client needs.

Similarly, since a request (program) may return a bind-

ing environment e as a representation to the client (CP2), e may contain detailed and specific metadata, multiple representations, and/or access functions for obtaining alternate representations from other origin servers (CP3, CP4).

The utility of computation transparency strongly suggests that CREST interpreters contain a rich selection of reflective functions that allow mobile codes to inspect or adjust ongoing computations. Possible applications include remote monitoring, debugging, control and post-mortem, dynamic reconfiguration, or mobile workflow.

8.1.3 Mitigate latency

With the transfer of computation (CP1, CP2, CP3) there is no guarantee that the origin server S to which client c issued a request (computation) will be the eventual responder, since S may easily ship the partial computation to a third-party, unknown to c, for completion. The locus of computation is therefore fluid; there is no fixed relationship between request and response since one request may generate zero, one, or many responses and, in the latter case, from many distinct peers. Nor is there any guarantee as to the order in which multiple responses to multiple requests may arrive at the client c.

Given the dominance of computation and computational transfer within CREST, peers should adopt a strategy of minimizing the impact of network and computational latency on themselves and others. If these interactions are synchronous, then any latencies in-the-large will be reflected in latencies in-the-small; for example, a client becomes unresponsive while it fetches a resource, or an intermediary stalls while composing multiple representations from upstream sources.

Therefore, both clients and origin servers must have mechanisms for reducing or hiding the impact of latency. For example, clients must be hospitable to concurrent computation and event-driven reactions (such as the nondeterministic arrival of responses). Since those responses may be continuations, origin servers, in an analogous manner, must be open to receiving previously generated continuations long after they were first created (on timespans of seconds to months) and restarting them seamlessly.

Time is also fundamental to CREST peers as both origin servers and clients require timers to bound waiting for a response or the completion of a computation. In addition, CREST peers may employ timestamps and cryptographic signatures embedded within a continuation to reliably determine its age and may refuse to resume continuations that are past their "expiration dates."

8.1.4 Migrate computations

CREST directly fosters computational migration since it is openly hospitable to mobile code in many forms (as programs, closures, or continuations) and allows the direct exchange of binding environments (CP1, CP2). This allows a service to scale up as needed by sharing network resources; a single origin server may now be dynamically reconstituted as a cooperative of origin servers and intermediaries (CP3, CP4). Clients must be willing to dynamically stitch partial computations from origin servers into a comprehensive whole to obtain desired results, as no one origin server may be capable of supplying all of the functional capability that the client requires.

Computational namespaces and migration also suggest that every time content is transformed, it must be completed without an implied reference to another transformation. This is not to say that transformations can't be explicitly chained together; however, if an implicit dependency chain emerged, it would jeopardize the integrity of the computation as implicit dependencies may not be preserved during migration.

8.1.5 Provide multiple interfaces

One of the persistent criticisms of REST has been the lack of supporting frameworks that enforce its style and basic design principles. To support computational exchange and shorten the learning curve, there should be a well-supported framework for quickly creating custom CREST applications. For example, basic tasks (such as fetching a resource) should be easily supported in a minimum of code. But, as an architect becomes familiar with the framework, there should be a corresponding gradual increase in the span of control and expressiveness.

8.2 CREST Architectural Style

Broadly speaking, we can gather these guidelines and constraints together to codify a new architectural style that expands REST's concept of state transfer to encompass computational transfer. Just as REST requires transparent state exchange, our new style, Computational REST (CREST), further requires the transparent exchange of computation. Thus, in the CREST architectural style, the client is no longer merely a presentation agent for content—it is now an execution environment explicitly supporting computation.

9. CREST EXPLANATIONS

9.1 Explaining mod_mbox and Subversion

As highlighted in Section 3.1, mod_mbox took specific advantage of exposing a computational namespace and the use of dynamic representations. With CREST as our guide, we now understand that computational choices had a noticeable impact on the overall architecture. Instead of exposing storage details in the namespace, mod_mbox only exposed content-specific metadata: the Message-ID header. This level of indirection shielded the web-level namespaces from irrelevant implementation decisions made at a lower level (CP2, CP3). Similarly, by delaying the creation of message representation, mod_mbox was able to later evolve gracefully. Subsequent mod_mbox development added an AJAX interface. Instead of creating new representations at indexing time, mod_mbox could simply dynamically expose a new XML-based namespace and representation suitable for AJAX when an AJAX-capable client requested it (CP2, CP3). Adding AJAX supported the further shifting of the computational locus away from the original server.

As discussed in Section 3.2, Subversion initially suffered from network latency issues. With the explanatory powers of CREST, we view the first attempt to solve this latency issue (performing, on the server, the aggregation of resources to check out) as moving the computational locus back from the client to the server. Unfortunately, while addressing the initial latency issue, this only served to increase the overall computational load on the server and made it such that simple intermediaries could not be deployed to reduce load. But, with considered changes to the client's framework, consistent with latency reduction, we could repair the

deficiencies. A new client was deployed that addressed the issues of latency through independent transformational elements (buckets) and asynchronous network calls. Hence, the computational locus (the aggregation of resources to check out) could rightfully return to the client. The server's load is thereby lessened and intermediaries can be redeployed. With the CREST constraints, we can express the problem, its deficiencies, and its ultimate solution.

9.2 Explaining Web Services

As CREST is intrinsically focused on the exchange of computations (rather than just content), it seamlessly supports service exposition (CP1, CP2, CP3). As discussed in Section 4, supporting web services that are not directly query-like with a RESTful API is difficult, and SOAP clearly conflicts with REST. However, by using a well-defined binding environment (CP4) and continuations (CP3), CREST provides more substantial design guidance on how to create arbitrary dynamic services that are flexible and content-independent.

For instance, for a service that may return a large number of results (such as a list of all live auctions), the exchange of continuations (as generators) is more natural, responsive, and elegant than building pagination and chunking explicitly into the interface. On each round i the service S may return a ordered pair (a_i, C_i) where:

- $a_i = (a_{i,0}, \ldots, a_{i,m_i})$ is a list of the representations of the next m_i out of n total live auctions and
- C_i is the continuation that, when returned to S at sometime in the future, will generate the next set of $(a_{i+1,0}, \ldots, a_{i+1,m_{i+1}})$ live auction representations.

Note that continuation C_i need not be returned to the origin server immediately; many seconds or minutes may pass before the client returns the continuation for the next round of exchange. This allows the client to pace its consumption of the results, a freedom that pagination and chunking do not provide. Further, the client may pass a continuation C_i onto one or more fellow clients for other purposes; for example, parallel searching for auctions with particular items for sale. In CREST, these elaborations are transparent and straightforward, whereas with web services, byzantine supporting protocols and mechanisms are required.

9.3 Explaining Cookies

Under CREST, cookies are now reinterpreted as a weak form of continuation (CP2). When a client wants to resume the transaction represented by the cookie (continuation), it simply returns the last cookie (continuation) to the server. The key distinction under CREST is that cookies (continuations) are bound to a specific, time-ordered, request sequence. A full continuation is, by definition, bound to a particular sequence of resource access; there is no ambiguity server-side. Thus the continuation restarts resource access at a particular known point in the sequence of all resource accesses (CP2, CP3). This stands in sharp contrast to the current use of cookies—*generic* tokens to all subsequent server requests.

Cookies are also currently presented with an explicit expiration date (in practice, many sites set their cookies to the latest expiration time supported by 32-bit platforms: January 19, 2038 [21]). However, in CREST, no such expiration date must be supplied (though it may, at the discretion of the origin server) as the continuation itself contains all of the necessary state to resume the dialog of exchange. Finally, continuations, like cookies, can be hardened against tampering using digital signatures or even encryption to prevent security or service attacks (CP2).

9.4 Explaining AJAX and Mashups

From the CREST perspective, mashups are nothing more than a closure (CP2) that makes reference to resources on multiple web sites $w_1, w_2, \ldots, w_n$. Note that for CREST, there is no requirement that a web browser be the execution environment for the mashup.

By using CREST, we can predict two future elaborations of mashups. A *derived* mashup is one in which one or more content provider web sites w_i are themselves mashups—with the lower-level mashups of the w_i executing on an intermediary (CP5) rather than a browser. CREST also speaks to a future web populated by *higher-order* mashups. Similar to a higher-order function in lambda calculus, a higher-order mashup is a mashup that accepts one or more mashups as input and/or outputs a mashup (CP2). This suggests a formal system of web calculus, by which web-like servers, clients, and peers may be cast as the application of identifiable, well-understand, combinators to the primitive values, functions, and terms of a given semantic domain. Thus, CREST hints at the existence of future formalisms suitable for the proof of REST and CREST properties.

10. EVALUATION AND CONCLUSIONS

How are we to evaluate the validity of CREST as an explanatory model of modern and emerging web behavior? First, note that REST is silent on many issues of architectural mismatch, repeatedly neglects to offer explicit design guidance, lacks a bright line separating REST-feasible web services from those that are not, fails to predict novel services that are consistent with REST principles and is frequently silent on many issues of web behavior. In each of these cases, CREST fills the gap and provides detailed guidance and explanantions where none existed previously.

This is particularly acute in the case of SOAP and web services. Why are developers so focused on ignoring REST constraints for the sake of web services? Developers are struggling toward service interactions far finer-grained than fetching hypermedia content. But, absent a comprehensive computational model, the only mechanism even remotely suggested by REST is parameterized request/response that relies on the ill-suited semantics of the GET and POST methods of HTTP. CREST tackles the problem directly, since content exchange is nothing but a side-effect of its primary focus: computational exchange. Further, it demonstrates why SOAP and the tower of web service standards and protocols stacked above it utterly fail; computational exchange requires the full semantics of powerful programming languages: conditionals, loops, recursion, binding environments, functions, closures, and continuations, to name only a few. Without these tools, web service developers are condemned to recapitulate the evolution of programming languages.

CREST identifies the precise reasons why the evolution to web services is so difficult, pinpoints the mechanisms that must be applied to achieve progress, and offers detailed architectural and design guidance for the construction

of service-friendly servers and clients. Thus, CREST offers guidance where REST and all others have failed so far. In future work, we intend to apply CREST to the entire spectrum of web services—recasting all major elements of the web services protocol stack in the light of computational exchange—as well as address other outstanding problems in web behavior, including content negotiation and effective caching in service-oriented architectures.

11. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant Numbers CNS-0438996 and CCF-0524033.

12. REFERENCES

[1] Amazon Web Services. Amazon S3. http://docs.amazonwebservices.com/AmazonS3/2006-03-01/, March 1, 2006.

[2] Apache. http://httpd.apache.org/mod_mbox/, 2006. The Apache Software Foundation.

[3] Caswell-Daniels, M. Goggles :: The Google Maps flight sim. http://www.isoma.net/games/goggles.html, 2007.

[4] CollabNet. http://subversion.tigris.org/, 2003.

[5] CollabNet. http://eyebrowse.tigris.org/, 2006.

[6] eBay Inc. REST - eBay developers program. http://developer.ebay.com/developercenter/rest, 2007.

[7] Erenkrantz, J. R. Web Services: SOAP, UDDI, and Semantic Web. Tech. Rep. UCI-ISR-04-3, Institute for Software Research, University of California, Irvine, May 2004.

[8] Erenkrantz, J. R. Architectural Styles of Extensible REST-based Applications. Tech. Rep. UCI-ISR-06-12, Institute for Software Research, University of California, Irvine, August 2006.

[9] Fielding, R. T., and Taylor, R. N. Principled design of the modern web architecture. In *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland, May 2000), IEEE, pp. 407–416.

[10] Fielding, R. T., and Taylor, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology 2*, 2 (May 2002), 115–150.

[11] Fuchs, M. *Dreme: for Life in the Net*. PhD thesis, New York University, September 1995.

[12] Fuggetta, A., Picco, G. P., and Vigna, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering 24*, 5 (1998), 342–361.

[13] Garrett, J. J. AJAX: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, February 18, 2005.

[14] Google. Google maps API. http://maps.google.com/apis/maps/, 2007.

[15] Graunke, P. T., and Krishnamurthi, S. Advanced control flows for flexible graphical user interfaces: or, growing guis on trees or, bookmarking guis. In *Proceedings of the 24th International Conference on Software Engineering* (New York, NY, USA, 2002), ACM Press, pp. 277–287.

[16] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., and Nielsen, H. F. Simple Object Access Protocol (SOAP) 1.2: Adjuncts. http://www.w3.org/TR/soap12-part2/, June 24 2003.

[17] Halls, D. A. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, June 1997.

[18] Hood, E. http://www.mhonarc.org/, 2004. Version 2.6.10.

[19] Kristol, D., and Montulli, L. HTTP state management mechanism. http://www.ietf.org/rfc/rfc2109.txt, February 1997.

[20] Matthews, J., Findler, R. B., Graunke, P., Krishnamurthi, S., and Felleisen, M. Automatically restructuring programs for the web. *Automated Software Engineering. 11*, 4 (2004), 337–364.

[21] Microsoft Corporation. ASP 200 error setting cookie expiration past January 19, 2038. http://support.microsoft.com/kb/247348, November 21, 2006.

[22] Mitchell, K. A matter of style: Web services architectural patterns. In *XML 2002* (Baltimore, MD, December 8-13 2002).

[23] Mitra, N. SOAP Version 1.2 Part 0: Primer. http://www.w3.org/TR/soap12-part0/, June 24 2003.

[24] Nottingham, M. Understanding web services attachments. http://dev2dev.bea.com/pub/a/2004/05/websvcs_nottingham.html, May 24, 2004.

[25] O'Reilly, T. REST vs. SOAP at Amazon. http://www.oreillynet.com/pub/wlg/3005, April 3, 2003.

[26] Queinnec, C. The influence of browser on evaluators or, continuations to program web servers. In *Proceedings of the International Conference on Functional Programming* (Montreal, Canada, 2000), ACM.

[27] Tarau, P., and Dahl, V. High-level networking with mobile code and first-order AND-continuations. *Theory and Practice of Logic Programming 1*, 3 (May 2001), 359–380.

[28] Trachtenberg, A. PHP web services without SOAP. http://www.onlamp.com/pub/a/php/2003/10/30/amazon_rest.html, October 30, 2003.

[29] Vyzovitis, D., and Lippman, A. MAST: A dynamic language for programmable networks. Tech. rep., MIT Media Laboratory, May 2002.

[30] W3C. Web of services for enterprise computing. http://www.w3.org/2007/01/wos-ec-program.html, February 27-28 2007.

[31] Winer, D. XML-RPC Specification. http://www.xml-rpc.com/spec, June 15 1999.

[32] Yahoo! Inc. Flickr services. http://www.flickr.com/services/api/, 2007.

[33] Young, M. AP News + Google Maps. http://81nassau.com/apnews/, 2007.