Rethinking Web Services from First Principles

Extended Abstract

Justin R. Erenkrantz Michael M. Gorlick Richard N. Taylor

Institute for Software Research
University of California, Irvine
jerenkra@ics.uci.edu, mgorlick@acm.org, taylor@ics.uci.edu

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, proposed Web Services protocols abandon REST altogether in favor of SOAP (Simple Object Access Protocol) exchanges codified in XML that hijack HTTP (HyperText Transport Protocol) as transport. Another path is possible. Our investigation yields a set of extensions to REST, an architectural style called Computational REST (CREST), that embraces service exchanges as the fundamental element of web interaction, obviating Web Services as a separate, incompatible layer atop a web whose underlying architectural model is REST.

I. Introduction

The REpresentational State Transfer (REST) architectural style [1] 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 [1] where 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 (such as weather forecasts), a collection of other resources, 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.

RP3 *All interactions are context-free.* REST applications are not necessarily without state, but each interaction contains all of the information necessary to understand the request.

RP4 *Only a few primitive operations are available.* REST components can perform a small set of well-defined methods on a resource to produce a representation.

RP5 *Idempotent operations and representation metadata are encouraged in support of caching.* The metadata included in requests and responses permits REST components (such as user agents or caching proxies) to make sound judgements of the freshness and lifespan of representations. The 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 exchanges in a manner transparent to both endpoints.

II. Web Evolution

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.

AJAX and mashups 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.

In the world of computational exchange, an URL denotes a computational resource. There, clients issue requests in the form of programs p, origin servers evalute those programs, and the value v of that program is the response returned to the client. That value (response) may be a primitive value (`1`, `3.14`, or `"silly"` for example), a list of values `(1 3.14 "silly")`, a program, an expression, a closure, a continuation, or a binding environment (a set of name/value pairs 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 or expression composition. For example, the program (given in the concrete syntax of Scheme) issued by a client c to an URL u of origin server s

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

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.

III. CREST

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 (such as "the predicted weather in London over the next four days"), a generated collection of other resources, a simulation of an object (a spacecraft, for example), and so on.

CP2 *The representation of a resource is a value, expression, program, closure, continuation, or binding environment plus metadata to describe the value, expression, 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 core 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. p is 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 value, expression, program, closure, continuation, or binding environment (which itself may contain values, expressions, programs, closures, 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.

A. Protocols for CREST

The migration to computational exchange leads us to consider a more efficient set of wire-level protocols than HTTP/1.1; specifically, a new protocol that employs Scheme expressions as the base, wire-level transport mechanism. Transferring Scheme expressions at the network-level can be traced back to Halls [2]. By relying upon Scheme at the wire-level, CREST computations are expressed in a human-understandable programming language that is transportable among varied platforms and that relies on a minimal agreement for binding semantics among CREST nodes.

Given the near-universal adoption of HTTP, two primary challenges must be addressed by any potential replacement: compatibility and ease of deployment. Our protocol, by intention, is a superset of the existing HTTP/1.1 behaviors. This allows existing "legacy" HTTP/1.1 applications to operate through efficient protocol gateways that can seamlessly "upgrade" the protocol to interoperate with CREST-based servers. Deployment is faciliated by several protocol enhancements (such as the `Upgrade` connection header, first specified in HTTP/1.1 [3]) that allow a client to ask a server to switch the wire-level protocol to any arbitrary protocol. These two principles, compatibility and ease of deployment, pave the way for replacing HTTP over the long term.

IV. CREST and Web Services

Shifting the web from content exchange among clients and servers to computational exchange among participants simplifies service provisioning and transforms web services. Unlike the content-centric web described by REST, the computation-centric web envisioned by CREST reduces service exchange among peers to the commonplace. Since CREST URLs are reference points for language interpreters L with URL-specific binding environments, clients may compose client- and URL-specific programs or expressions that are interpreted in the binding environment of the target URL of the origin server. Thus, given an adequately expressive language interpreter L at URL u, it is trivial for a client to construct many services using nothing more than the core primitives of L and the specific values in the binding environment denoted by u. Hence CREST, courtesy of its fundamental principles, is service-friendly in a way, and to a degree, that is impossible for web services—whose principal mode of exchange, SOAP, is nothing more than a remote procedure call wearing an XML skin. If nothing else, CREST subsumes SOAP altogether, since any remote procedure call expressed in SOAP is trivially recast as an expression transmitted from client to server for evaluation and whose outcome (value) is returned to the client.

Web services, as envisioned by W3C (www.w3c.org), OASIS (www.oasis-open.org), OGC (www.opengeospatial.org), and other standards organizations, allow clients to invoke (as a remote procedure call) just those services exposed by the origin server and nothing more. It is impossible for the client to modify or extend those service points in any meaningful way outside of the narrow bounds defined by the specific service standard. Service composition, to the extent that it exists at all, requires an additional layer of protocols since the base web services do not define any service composition operators.

CREST, on the other hand, places far greater power in the hands of clients. A CREST request is an arbitrary program p that may compose and combine services using all of the functional semantics made

available by the interpreter L hosted at the target URL u, including functional composition, sequencing, conditionals, recursion, and iteration. CREST only requires that service providers expose the bare minimum services (reified as URL-specific functions in the binding environment of the target URL), since clients can simply construct and transmit programs that shape the service offering to their needs and circumstances.

In addition, since each such L contains core functions for transmitting programs or expressions, the program p may, from the site of URL u, issue requests for service directed to URLs other than u. Thus, service composition is a well-defined primitive operation in CREST, since any program may combine the services it finds at one URL u with the services found elsewhere at URLs $v \neq u$ hosted by other servers.

In the CREST world, services are client-driven and no client c need wait helplessly (as one would in the world of the W3C *WS-** standards) for some service provider to stand up exactly the "methods" that c requires. Instead, if it is possible to compose the service that c requires from other services, then c merely issues a program that does just that. Within the CREST model web services are client-driven and defined, not server-driven and defined.

CREST also greatly expands the range of primitive values that may be returned in response to a request and discourages application-specific binary return values in favor of general binding environments that cleanly separate data and metadata and are easily inspected by intermediaries. Binding environments structure complex data in a manner amenable to incremental inspection and decomposition, may offer multiple views or slices of the same underlying structure, and easily represent recursive data.

Finally, since CREST offers continuations as first-class values, it is possible to construct, on a network scale, control structures now found only within programming languages, including: iteration, recursion, generators, coroutines, exceptions, restart, replay, transactions, workflow, and synchronization [2], [4]. These control structures are either missing altogether from web services as they are understood by W3C and others, or require purpose-built protocols that are specific to just one class of web service. Within CREST continuations are uniformly available and, since services are client-defined rather than server-driven, clients may apply whatever higher-level control structures to those services that continuations and the semantics of URL-bound interpreters L permit. In short, since continuations are the universal primitive building block of almost every known control structure, clients—not servers or the authors of service standards—dictate the control flow of CREST-based services.

V. Summary

CREST generalizes the architectural principles enumerated by REST to transform the web from a medium designed for content exchange to one designed for service exchange. As a consequence, almost all of the protocols and standards now defined to promote and promulgate interoperable web services are either wholly superfluous or irrelevant. Architectural complexity may be a warning sign that the architecture is unsuitable for the task of hand. Perhaps the sheer volume of the W3C *WS-** standards is evidence that those standards are not only inconsistent with the fundamental architectural principles of the web (REST), but also that REST alone is inadequate for defining a service-friendly web. By embracing computational exchange as the fundamental web behavior, CREST paves the way for services that are as scale-friendly as the client/server structures of the original web.

This work supported in part by the National Science Foundation under grant CNS-0438996.

References

[1] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.

[2] D. A. Halls, "Applying mobile code to distributed systems," Ph.D. dissertation, University of Cambridge, June 1997.

[3] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol - HTTP/1.1," http://www.ietf.org/rfc/rfc2616.txt, June 1999.

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