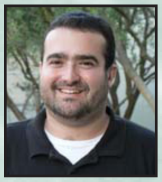
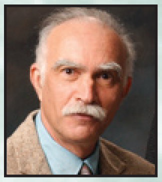
Harmonizing Architectural Dissonance in REST-based Architectures


Justin R. Erenkrantz
University of California, Irvine
jerenkra@ics.uci.edu

Richard N. Taylor
University of California, Irvine
taylor@ics.uci.edu


Michael Gorlick
University of California, Irvine
mgorlick@acm.org

Girish Suryanarayana
University of California, Irvine
sgirish@ics.uci.edu

December 2006

ISR Technical Report # UCI-ISR-06-18

Harmonizing Architectural Dissonance in REST-based Architectures

Justin R. Erenkrantz[1], Michael Gorlick[2], Girish Suryanarayana[1], Richard N. Taylor[1]

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3440 USA
[1]{jerenkra,sgirish,taylor}@ics.uci.edu, [2]mgorlick@acm.org

ISR Technical Report # UCI-ISR-06-18

December 2006

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 difficulties experienced, we sought insights from numerous architectures in both web and non-web domains. 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 in domains such as distributed computation and multimedia streaming.

Harmonizing Architectural Dissonance in REST-based Architectures

Justin R. Erenkrantz[1], Michael Gorlick[2], Girish Suryanarayana[1], Richard N. Taylor[1]

Institute for Software Research
University of California, Irvine
[1]{jerenkra,sgirish,taylor}@ics.uci.edu, [2]mgorlick@acm.org

ISR Technical Report # UCI-ISR-06-18

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 difficulties experienced, we sought insights from numerous architectures in both web and non-web domains. 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 in domains such as distributed computation and multimedia streaming.

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 micro-architecture of the individual element; that is, the structure, components, and micro-interactions within a single active element. In this paper, we explore the thesis that to maintain the fidelity of REST's principles on the macro-level interactions requires previously unspecified constraints on the micro-architecture of those individual elements.

We pursue here two parallel, but related, lines of investigation. Firstly, we draw upon our experience as application developers struggling to build web applications that conform to the REST style. Here, we discover both the macro consequences of failing to hew to the constraints of REST, and how micro architectures (on the scale of a single element) must be rearranged to align with REST's goals. Second, we contend that REST, as an applied architectural style, is far more pervasive and common than previously thought. Identifying REST-based constructions and interactions in settings other than the modern web clarifies the role of REST in influencing the micro-architectures of elements, irrespective of domain.

These two lines of inquiry lead to a deeper understanding and broadening of the fundamental REST principles. From these insights, we arrived at CREST, an elaboration of REST that includes a few key additional constraints, as the macro demands of REST require specific, but subtle, computational mechanisms deep within the micro structures of clients. However, these mechanisms, in the case of classic REST-based clients, are a simplification of a more general form - which we term network continuations - the exchange of the representations of the execution state of distributed computations. It is the presence and exchange of network continuations, in their various forms, that induces the micro-architectural constraints observed in classic web clients. With this in mind, both prior complications in the structure of clients and the elaborations of the Web such as AJAX or mashups are accounted for by a single fundamental mechanism - network continuations as a legitimate and rightful constraint.

Finally, the network continuations of CREST generalize a large class of loosely coupled, distributed computations and point the way toward more general and adaptive large-grain computational structures. In other words, the principles of CREST indicate new forms of computational interaction, just as REST reflected an emergent class of request/response behaviors among server/client interactions. Thus, CREST not only explains phenomena for which REST alone cannot account, but also predicts the macro- and micro-architectures of new forms of distributed services.

The rest of the paper is structured as follows. Sec-

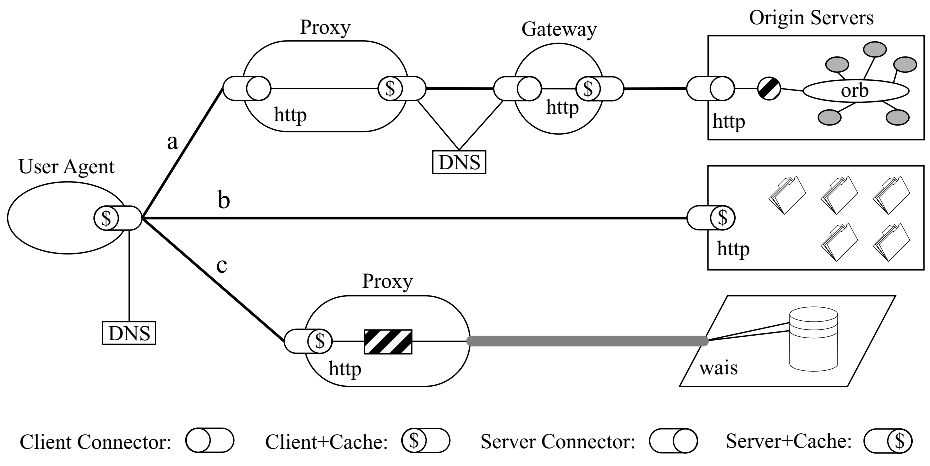


FIGURE 1. Process view of a REST-based architecture at one instance of time (From [9])

tion 2 recaps the REST style. Section 3 recounts our experience creating REST-based web applications. Section 4 reviews how the concepts of REST have been applied to non-web domains. Section 5 introduces the CREST architectural style with an evaluation of the architectural power of CREST in Section 6. The paper concludes with a summary of CREST in Section 7.

2. Representation State Transfer (REST) architectural style overview

The Representational State Transfer (REST) architectural style, first presented at ICSE 2000 [8] with a more complete description in [9], governs the proper behavior of participants on the World Wide Web. As depicted in Figure 1, in a typical REST interaction on the modern Web, a user agent (e.g. 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.

As presented in [9], 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. REST enables the caching and reuse of previous interactions, dynamic substitutability of components, and processing of actions by intermediaries, thereby meeting the needs of an Internet-scale distributed hypermedia system [8].

All REST interactions 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. *The key abstraction of information in REST is a resource*. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Minneapolis"), a collection of other resources, a moniker 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. *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. In turn, *REST components can perform a small set of well-defined operations on a resource* by using a representation to capture the current or intended state of that resource and transfer that representation between components.

2.1. Macro and micro REST architectures

Looking to the REST architectural style for answers on how to construct REST-based applications leads to a partially unfulfilling experience. REST provides guidance on how to construct *macro-architectures* in the REST style by informing how independent REST nodes should properly communicate. REST purposely provides little-to-no guidance as to how to build *micro-architectures* in a principled manner - that is, what types of components are required, and how those components should be built, arranged, and interact within the context of a single REST node. Yet, as we discuss below, we believe there is a compelling and previously unexplored relationship between how REST governs the interaction of nodes at the macro level and how those nodes should be designed at the micro level.

3. Insights from Web-based examples

A simple Google search for "REST" yields many websites devoted to REST that provide evidence regarding the difficulty of understanding and applying REST principles. A number of web systems do not exhibit the benefits anticipated by REST's constraints. To help us better understand this phenomenon, we recount our experiences building and repairing two such systems: mod_mbox, a service for mail archiving, and Subversion, a configuration management system.

3.1. mod_mbox

As core contributors to the open-source Apache HTTP Server, we wanted a scalable web-based mail archiver to permanently capture the design rationales and decisions that we made on our project mailing lists. However, early web-based mail archivers presented scalability and persistence problems. Some systems, such as MHonArc [13], converted each incoming message into HTML and recomputed the list index as the messages arrived. Constant regeneration is problematic for high-traffic lists - such as our target lists. To host the amount of anticipated messages, the server would unnecessarily and constantly spend its time updating the archive. Therefore, though written as a web service, the scalability of the server was in doubt.

Other web-based archivers, such as Eyebrowse [6], used a relational data store that was internally dependent upon the ordering as to when messages were loaded into the archives. Therefore, if we suffered hardware failures and regenerated the archive, any previous links could be stale or now point at a different message. This meant that any persistent hypertext links - such as those from our own code - could be broken if we had to regenerate the archive.

Therefore, relying upon our experience with the Apache HTTP Server, we felt we could create a new web-based archiver based on the REST style that would not suffer from these shortcomings. However, as we found, REST on its own was not expressive enough to guide the creation of our system. *The architecture of our archiver, mod_mbox, added two additional constraints to its REST architecture drawn from our study of MHonArc and Eyebrowse: dynamic representations of the original messages as well as the use of a consistent namespace [19].*

Instead of creating HTML representations as the messages arrive, mod_mbox delays that transformation until a request for a specific message is received. When a message arrives for archival, a metadata store is updated with the new message's information. When that message is requested from the archive, a plug-in to the Apache HTTP Server creates a dynamic HTML representation of the message with the help of the metadata store. This clear distinction between the resource and representation minimizes the computational costs for preparing the archive which allows mod_mbox to handle more traffic. To achieve a consistent namespace, mod_mbox relies upon relevant metadata (the Message-ID header) accompanying the original message. Therefore, if the metadata index has to be recreated, the exposed resources will remain the same allowing for long-term persistence of links. By adopting these constraints, mod_mbox now archives all of the mailing lists for The Apache Software Foundation (in excess of 2.5 million messages to date).

3.2. Subversion

Subversion [5], a SCM designed to be a compelling replacement for the popular CVS system, made a decision early on in the design to use WebDAV, a set of extensions to HTTP. Hence, Subversion was expected to conform to the REST constraints and gain from REST's anticipated benefits such as minimizing latency and permitting intermediaries. A REST-based versioning system depends upon fetching resources from an origin server. To limit the number of connections and minimize network overhead, the client would not create a new connection per resource - instead, it would try to reuse a small number of connections. Since the client must then make multiple requests on the same connection, network latency becomes a concern. This problem was anticipated in the early days of standardization of the HTTP protocol, but was not clearly articulated within REST - instead a mechanism called pipelining was simply suggested to issue requests without waiting for previous responses. However, due to the lack of detailed design guidance, Subversion developers did not anticipate the seriousness of not having pipelining and instead used a straightforward serial approach to checking out resources.

To solve this performance issue in Subversion, a custom WebDAV method on the server was implemented to reduce network round-trip latency. Instead of requesting each resource individually, the new client would request them in bulk. As expected, this change led to a better overall network utilization by having the server stream responses back to the client in one large response. However, this custom method created several dissonances with REST. The computational and network load on the server was increased as roughly 33% more bytes had to be transferred to satisfy XML encapsulation requirements [15]. Additionally, simple HTTP caches could no longer be deployed to ease the load on upstream Subversion servers as they would not understand the custom method.

Some core Subversion developers, including one of the authors of this paper, felt that adopting a different client framework could restore the lost REST benefits. Therefore, an effort was undertaken to rewrite the micro-architecture of the Subversion client. *This new framework, serf, permitted the reintroduction of serial checkouts with pipelining over multiple connections without significant performance loss. It achieved this through two new micro-architectural constraints: the concept of a bucket to perform independent data transformations and enforcing non-blocking network connections [7].* These buckets represent data streams to which successive transformations are applied on-the-fly allowing the client to use less computational resources. Non-blocking connections improved network efficiency and reduced latency, as these buckets would not have to wait to write or read data. Once serf was introduced to Subversion, the need for XML-encoded data was removed and intermediary caches could now be re-considered to help Subversion scale. Hence, there were subtle interactions between the macro- and micro-architectural constraints that prevented the developers from fully realizing the benefits of REST.

4. Insights from the network stack

To better understand the gap between the wholesale success of REST on the macro-scale of the web and the difficulty of applying REST on the micro-scale of specific clients or applications, we also looked further afield for examples of REST. Though REST was first codified as an application-level architectural style, its principles are far more pervasive than one might first think, appearing in various guises in the network stack. Even though these systems predate REST, we discover analogs to REST elements (server, client, etc.) which make specific accommodations in their micro-architecture for the sake of migrating computation, accommodating state or content transformations, minimizing core functions to reduce complexity, or mitigating latency.

4.1. Network layer

The internet offers a single service, best-effort delivery, implemented by routers that forward incoming packets as they arrive by interpreting the destination address of each packet in the context of a *routing table*. The state of the routing table is finite and largely fixed and does not need to maintain any fine-grain state with respect to traffic, history, or reachability beyond its immediate outgoing connections.

Routing is a classic distributed computation as packet forwarding is computed hop-by-hop by the transit routers. *The robustness and scaling of internet routing is a direct consequence of the REST principles as routing implements an abstract namespace, IP addressing; explicit and repeated state transfer, the header of the IP packet in hand; and generic operations, specifically best-effort forwarding from an inbound port to an outbound port*. Routing, like the web, must also minimize latency. This goal dictates that the micro-architecture of a router be highly concurrent to prevent congestion and reduce packet loss. Router architects go to great lengths, using specialized processors and complex multi-threaded implementations, to achieve a high degree of concurrency and stage routing as pipelines of transformations to reduce core complexity.

4.2. Network flow management

Routing may also be significantly improved without abandoning the REST model. As now constituted, network endpoints are independently responsible for congestion control. Therefore, network-wide flow management that does not increase complexity or degrade performance and robustness is highly desirable. SCORE is one example that requires no distributed, dynamic, replicated and consistent flow state network-wide since all the requisite, per-packet, flow state is transported within each individual packet [18]. SCORE also introduces us to the idea of network continuations - the representation of the execution state of a distributed computation (flow management). Since each packet carries the entire state information, SCORE's dynamic flow management algorithm is context-free as the routers need not maintain any per-flow state. *SCORE conserves our REST-based view of routing by augmenting each packet with a network continuation thereby carrying forward the REST properties of independence and scaling. SCORE also preserves the concurrent micro-architecture of routers since flow computations may be applied independently of other router computations.*

4.3. Network transport

Stateful, connection-oriented protocols, such as TCP, may be recast as context-free by replacing endpoint state with state transfer. Aura and Nikander [2, 3] first observed that a stateful network connection, where both endpoints retained state that determined the progress and rate of an exchange, could be converted to a context-free connection, wherein only one endpoint maintained the state required by the connection.[1] They suggested that servers employing such connections would exhibit far more graceful behavior under heavy connection loads, since the amount of state required server-side was reduced to a small constant and was invariant with respect to the number of ongoing connections. This insight was revisited in Trickles [17], a reconstruction of the TCP network stack using the techniques of Aura and Nikander - demonstrating throughput comparable to a stateful TCP stack and exactly the scaling performance predicted earlier.

The REST principles apply in full to this reconstruction of TCP: an abstract namespace, positions in a byte stream; the repeated explicit transfer of state, the TCP control block; and a small set of generic operations, segment transmission, positive acknowledgement, and selective retransmission. Trickles recast the TCP control block as a network continuation which transformed a TCP endpoint into a scalable context-free server where server and network latency is minimized by concurrency, multiple simultaneous transactions between endpoints is permitted (independence of transactions), and a set of highly constrained generic operations is provided (minimal core functions).

4.4. Network sessions

The adoption of REST disciplines at the lower transport layer induces new behavior at the session layer immediately above. As seen with Trickles, transforming a stateful connection to a context-free one has intriguing architectural consequences [17]. Stateful connections bind the connection to specific endpoints, while context-free connections allow endpoint migration. With a context-free connection, a server replica may service any packet of a context-free connection transparently - either to failover or offload work (load balance). Conversely, since clients hold the state required to resume a connection, client-side connection delegation is also feasible, as the state can be provided to an intermediary acting on its behalf. This allows clients to nominate upstream intermediaries.

Viewed in this way, the presence of explicit state exchange in REST has profound implications on the scalability of both the macro- and micro-architectures. It also suggests a regime of long-lived connections that see only intermittent, infrequent use but whose connection state may be traded or shared among a collection of trusted partners.

1. The authors used the term stateless, but we believe context-free is a better description as one side still always maintains state. In this situation, the client presents the state to the server every time.

4.5. Network applications

A new class of applications has recently emerged on the Web that extend the notion of a REST interface in interesting ways. One such example is Google Maps [11], which employs an application model known as *AJAX* [10], 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 silent, the interpretation environment for delivered content. This silence was deliberate, as content interpretation and presentation is highly content- and application-specific. Browsers have long provisioned for *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 context.

Dynamically downloading the code to the browser moves the computational locus away from the server. Instead of performing computations solely on the server, some of the computations (perhaps for presentation logic) can now be executed locally. By reducing the computational latency such that these events happen locally, 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 implementations serve to reduce server-side computational load. Thus AJAX, respecting all of the constrains of REST, expands our notion of resource.

5. Computational REST architectural style

Driven by the insights from the previous sections and summarized in Table 1, we formulated three additional macro constraints for REST to better explain the phenomena we identified. These macro constraints, in turn, lead to a larger body of micro constraints, acting on the level of a single active element (server, cache, client, etc.). The relationships between the macro constraints on one side and the micro constraints on the other appear to be both rich and complex.

Table 1. Insight and macro constraint summary

Insight	*New constraint*
Delay content transformation needed to amortize the cost of computation	Execution dynamism
Expose content-specific namespaces to permit resource layering & extensibility	Computational namespaces
Support the transition of the computational locus away from the origin to result in more efficient allocations	Computational namespaces, locus of computation
Support long-lived connections that can be traded among partners	Execution dynamism

5.1. Macro-architectural guidelines

These new macro guidelines apply to multiple active elements simultaneously. We examine each macro guideline in turn, computational namespaces, the locus of computation, and execution dynamism. Each expresses an aspect of computational transfer - the REST exchange of computations as resources.

5.1.1. Computational Namespaces. The pervasive practice of exposing content-specific namespaces suggests that *computations be expressed as resources*. This constraint is far more specific than the minimal guidelines REST provides, but the specificity preserves metadata and data boundaries. Consider a user requesting that a list of items available for purchase be sorted in descending order of price. Often, this computation is executed at the origin server and an appropriate representation returned. By expressing the sort criteria in the namespace, a smart intermediary or browser can take the original list (possibly cached) and perform the sort closer to the requestor, thereby improving scaling and decreasing latency.

5.1.2. Locus of computation. The desire to transition the computational locus away from the origin server toward the client for the sake of reducing latency or improving interactive response leads to a more general form of computational exchange. This exchange requires the transfer of a continuation, a closure or context, and a code base - though in some cases the latter two elements may be elided. A continuation is always required, because otherwise the client node in the exchange cannot determine where to (re)start the computation. In many cases, the continuation is nothing more elaborate than a call to the equivalent of a C main()—the sole entry point of the computation.

In AJAX applications, that single call is specified as an XHTML onload event that is triggered when the server-supplied representation is fully loaded into the client. For example, the trivial XHTML fragment

```
<body onload="main()"></body>
```

informs the AJAX client to initiate the client-side computation by calling the Javascript function main() once the full page is loaded. In a more general AJAX form, a closure and continuation are constructed server-side as a single Javascript object O whose attributes and methods comprise the closure and continuation respectively. O is transmitted to the client as Javascript source appearing in the

```
<head> ... </head>
```

element of the page. Irrespective of the details of the representation, the computation, as a triumvirate of continuation, closure, and code base, is expressed as a web resource.

Just as the REST model induces web intermediaries, shifting the locus of computation induces computational intermediaries where the ability to generate continuations allows computation to be shifted from one node to another.[2] A service may, over time, move computation outward from servers, to intermediaries, or to clients as it evolves. Existing intermediaries, such as web caches, can be used as is since the movement of computation can be expressed in terms and representations that are backward-compatible with modern REST practices, protocols, and representations.

5.1.3. Execution dynamism. The drive to delay content transformation and support long-lived connections that may be traded among computational partners leads to execution dynamism where computation within a node may vary over time in form, content, and semantics. Since the node is not bound to any one single {continuation, closure, code base} triple, that triple may change - in part or in its entirety - over the execution history of the node. We sketch a few brief examples to illustrate. Let C denote a particular triple {continuation, closure, code base}. It is easy to imagine a node supporting two or more such "environments" simultaneously; in fact, Javascript engines embedded within web browsers do just that. C_i and C_j, $(i \neq j)$, each derived from separate, distinct, and otherwise unrelated origin servers, may participate in any number of cooperative computing relationships, for example, as producer/consumer in a larger in-node pipeline, as co-routines, or as active states in a large intra-node state machine. Here the distinct C_i remains unchanged but the relationships may change over time.

2. It is important to observe that neither the REST nor CREST styles are tied to HTTP, XHTML, Javascript or any other common web technology and (C)REST-based implementations are possible using altogether different protocols, representations, and programming languages.

Alternatively, individual C_i may come and go over time as a computation unfolds within a node.

Finer-grain dynamism within a single C is also feasible. One notion is that the closure of C may expand over time as follows: if a closure fails to resolve a reference r, then the closure returns to a sequence of origin servers, in order of decreasing preference, to obtain content to resolve r, thereafter caching the content in the expanded closure. Alternatively, the content in the closure may contain expiration dates and any reference r past an expiration date requires that the content be refreshed. A similar variation may be applied to the code base. Infrequently used functions, appearing only as stubs in the code, are resolved, much in the manner described for references, at the time of call. For the purposes of adaptation or upgrade a node may download a new environment C' and substitute it, on-the-fly, for C.

5.2. Micro-architectural guidelines

We now enumerate design guidance on the structure, components, and interactions within a single REST element and briefly indicate their relationships to the macro-architectural guidelines introduced above.

5.2.1. Migrate computations. The new macro-architectural constraints of computational namespaces and execution dynamism indicate that the computational components within a micro-architecture may need to freely migrate across the macro-architecture. This would allow a service to scale up as needed by deploying more physical resources; instead of operating as a single origin server, a micro-architecture may now dynamically be broken down into a combination of intermediaries and origin servers. Suitable supporting frameworks can facilitate this transition by allowing the computations to operate within the context of any REST node (server, intermediary, user agent, etc.).

5.2.2. Constrain transformations. Computational namespaces and execution dynamism 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.

5.2.3. Minimize core functions. Once we introduce the concept of execution dynamism and a locus of computation, we get a glimpse as to how REST-based architectures are expected to evolve. New functionality can be introduced on another node or within the current micro-architecture. When new functionality is introduced at the micro-architectural level (such as a new protocol or encryption method), the architecture must be able to accommodate these tasks. Hence, by reducing the immutable core of the micro-architecture to the bare minimum (such as reading and writing bytes), the new functionality can be dynamically introduced.

Table 2. Macro/micro constraint relationships

Macro constraints	*Micro constraints*
Computational namespaces, execution dynamism	Migrate computations
Computational namespaces, execution dynamism	Constrain transformations
Execution dynamism, locus of computation	Minimize core functions
Locus of computation	Mitigate latency
Execution dynamism	Provide multiple interfaces

5.2.4. Mitigate latency. Adding a notion of a locus of computation at the macro-architecture hints that micro-architectures should adopt a strategy for minimizing the impact of network and computational latency on other nodes. If these interactions are synchronous, any latency in the macro-architecture will be reflected in the micro-architecture; for example, a user agent becomes unresponsive while it fetches a resource, or an intermediary node stalls while composing multiple representations from upstream sources. Therefore, the micro-architectures should have potent mechanisms for reducing the impact of latency. While threading is one viable strategy, it is prone to race conditions and deadlocks. A framework can instead leverage a single network thread responding to asynchronous network events. These events notify the micro-architecture as to when data can be received or sent without waiting. In all cases, the framework requires timers to bound waiting for a response.

5.2.5. Provide multiple interfaces. To support execution dynamism and facilitate a reasonable learning curve, there should be ample assistance to ease writing code controlling the micro-architecture. 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 allowable control and expressiveness.

Table 3. CREST Architectural Style Overview

Guidelines	*Rationale*	*Exemplar*
mod_mbox: *Mail archiver system*, **Serf**: *Subversion using Serf framework*, **Mashups**: *AJAX-based site combinations*, **Multimedia**: *Dynamic transcoding of content*		
Macro-architecture guidelines		
Computational Namespaces	Desired computations in CREST must be explicitly identified in the request for a resource	New representation formats can be exposed ad hoc (**mod_mbox**)
Locus of computation	Late-binding of identification as to where the computation will occur allow for scalability and customization	Allow dynamic reconfigurations for data streams (**Multimedia**)
Execution dynamism	The macro-architecture can optionally configure itself on-the-fly by distributing code	Downloading of JavaScript into a web browser context (**Mashups**)
Micro-architecture guidelines		
Migrate computations	Computations should freely move around between CREST micro-architectures to permit scalability and customization	Moving presentation logic from server to browser via AJAX (**mod_mbox**)
Constrain transformations	Alterations of data streams should happen explicitly instead of implicitly	Individual transformations are atomically in a single bucket (**Serf**)
Minimize core	Allow CREST frameworks to support different protocols	All protocol logic in buckets (**Serf**)
Mitigate latency	Reduce the impact of combining resources from multiple upstream sources or those which have expensive operations	Asynchrony and non-blocking network operations (**Serf**)
Provide multiple APIs	Facilitate a learning curve for developers adopting CREST	Coarse and fine-grained APIs (**Serf**)

5.3. CREST Architectural Style

Broadly speaking, we can gather these constraints together to codify a new architectural style that expands the concept of state transfer to include *computational transfer*. Just as REST requires transparent state exchange, our new style, Computational REST (CREST), further requires the transparent exchange of computation. This evolutionary expansion of REST is easier to appreciate if one regards state exchange from the perspective of a programming language continuation - the representation of the execution state of a program such that the computation may be suspended (rendered as a continuation) and resumed at a later point in time by reconstituting the continuation as an active execution state. Thus, in the CREST model, the client is no longer merely a presentation agent - it is an execution environment supporting the computational goal of the CREST exchange. A summary of the CREST style and benefits is presented in Table 3.

6. Evaluation of CREST

We adopt a three-dimensional approach to evaluating the CREST design guidelines. We first examine whether CREST can adequately explain the dissonances observed in existing systems that were situated in a REST-based environment. Towards this end, we revisit the motivating examples from Section 3, mod_mbox and Subversion. Next, using the example of web mashups, we investigate how CREST guidelines help elucidate emerging architectures that could not be satisfactorily explained by REST alone. Finally, we hypothesize how CREST can guide the composition of future architectures, including new forms of web computations.

6.1. Explaining mod_mbox and Subversion

As previously introduced in Section 3.1, mod_mbox took specific advantage of exposing a computational namespace and the use of dynamic representations. Using CREST as our guide, we now understand that the choices at a micro-architectural level had a noticeable impact at the macro-architectural level. 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 macro-architectural feature of namespaces from decisions made at micro-architectural level. 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. Adding AJAX supported the further shifting of the computational locus away from the original server.

As we discussed in Section 3.2, Subversion initially suffered from network latency issues. With the explanatory powers of CREST, we can view, at the macro-architectural level, 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 micro-architecture, we could repair the macro-architectural deficiencies. A new client was deployed which 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 re-deployed. Hence, with the CREST constraints, we can express the problem, its deficiencies, and the ultimate rectification.

6.2. Emerging architectures: Mashups

Mashups, an emerging form of REST-based service, are characterized by the participation of at least three distinct entities:

- *A web site, the mashup host* h
- *One or more web sites* $\omega_1, \omega_2 \ldots \omega_n$, $\omega_i \neq h$ *, the content providers*
- *An execution environment* ε *, usually a client web browser*

The mashup host h generates an "active" page P for the execution environment ε . P contains references (URLs) to resources maintained by content providers $\omega_1, \omega_2 \ldots \omega_n$ and a set of client-side scripts in Javascript. The client-side scripts, executing in ε , interpret user actions and manage access to, and presentation of, the combined representations served by $\omega_1, \omega_2 \ldots \omega_n$.

Google Maps-driven mashups are especially popular; examples include plotting the location of stories from the Associated Press RSS feed [20], and Goggles, a flight simulator [4]. One highly original mashup contains photographic collages whose individual "pixels" are minute images of books and video tapes sold by Amazon.com [16].

From the CREST perspective, mashups are nothing more than a triple {continuation, closure, code base} where the closure makes reference to resources on multiple web sites $\omega_1, \omega_2 \ldots \omega_n$. Note that from the perspective of CREST, there is no requirement that web browsers be the execution environment for the mashup.

6.3. Guiding future architectures

6.3.1. Derived Mashups. 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* are themselves mashups - with the combination execution happening on an intermediary 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. 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.

6.3.2. Distributed Computing. CREST also speaks to the domain of large-scale, loosely-cooperative, distributed computation (such as seen with SETI@home [1] or Folding@home [14]). CREST suggests that, for some distributed computing applications, work assignments be issued as purely REST exchanges where computational results are returned to the origin server (or a designated agent server) as a REST-based HTTP-like POST or PUT. Since the client application code is (re)issued with each work request, a client application update is trivial as it requires only a single centralized update at the origin server. Immediately thereafter, each participating client is updated at the next CREST-based work request/assignment exchange.

6.3.3. Multimedia. CREST can also be applied to multimedia exchanges. *Flows* are content or state transfers that extend over significant periods of time between two endpoints. STREeaming stAte Kinematics (STREAK) extends REST to establish, route, and redirect real-time flows originating from video cameras, sensor fields, stock market tickers, and the like [12]. CREST and STREAK can be combined to create enterprise-scale, distributed, adaptive computations. At the transport layer, STREAK is the mechanism by which the topology of directed graphs of dataflow computations may be rearranged at execution time in a REST-based manner. CREST provides mechanisms and guidelines for starting, pausing, resuming, and halting the active nodes within the dataflow graph.

For example, CREST-compliant transcoders may be dynamically inserted into video flows, via the mechanisms of STREAK, to provide client-device dependent transcoding services that are sensitive to client limitations such as screen size, connection bandwidth and quality, processing capacity, or power reserves. Each transcoder is a CREST-based computation that functions as the transcoding equivalent of a mashup, accepting one or more video streams as inputs (the analog of a mashup's content providers) and generating one or more refined or derived streams that may be delivered to an end device or serve as feedstocks for other CREST-based nodes downstream.

The division of labor, sketched above, between two REST-based styles, STREAK for topological (connective) adaptation and CREST for computational adaptation, suggests that other architectural styles, such as dataflow, may be reinterpreted and refactored in keeping with REST-based principles to create styles that combine the best attributes of the base style with the scaling and robustness of REST.

7. Summary

Our goal in this paper was to provide an understanding of the relationship between the macro- and micro-architectures in REST-based systems. Using REST as our guide, we have drawn numerous insights from a diverse collection of systems that either were web-based and failed to express the anticipated benefits of REST or shared the concepts of REST but found in a domain other than the web. REST by itself was not expressive enough at the micro-architectural level to fully capture why these systems succeeded or failed. In response to these sets of insights, we have introduced an additional set of constraints to REST, creating a new style called Computational REST (CREST).

The underlying insight that guides the set of CREST constraints is that we must include *computational transfer* in addition to representational state transfer to properly capture the necessary architectural decisions. In this paper, we looked at numerous systems to examine how CREST either better explains their set of architectural decisions or guides creation of new systems. And, we believe that CREST can predict new systems that we have not yet seen but are natural consequences of the CREST constraints.

8. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Numbers CNS-0438996 and CCF-0524033.

9. References

[1] Anderson, D.P., Cobb, J., et al. SETI@home: an experiment in public-resource computing. *Communications of the ACM*. 45(11), p. 56-61, November, 2002.

[2] Aura, T. and Nikander, P. *Stateless Connections*. Helsinki University of Technlogy, Report Research Report A46, May, 1997.

[3] Aura, T. and Nikander, P. Stateless Connections. In *Proc. of the First International Conference on Information and Communications Security*. 1334, p. 87-97, Springer-Verlag. London, November 11-14, 1997.

[4] Caswell-Daniels, M. *Googles Flight Sim*. <http://www.isoma.net/games/goggles.html>, HTML, 2006.

[5] CollabNet. *Subversion*. <http://subversion.tigris.org/>, HTML, 2003.

[6] CollabNet. *Eyebrowse*. <http://eyebrowse.tigris.org/>, HTML, 2006.

[7] Erenkrantz, J.R. *Architectural Styles of Extensible REST-based Applications*. Institute for Software Research, Report UCI-ISR-06-12, August, 2006.

[8] Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. In *Proc. of the 22nd International Conference on Software Engineering*. p. 407-416, Limerick, Ireland, June, 2000.

[9] Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*. 2(2), p. 115-150, May, 2002.

[10] Garrett, J.J. *Ajax: A New Approach to Web applications*. <http://www.adaptivepath.com/publications/essays/archives/000385.php>, HTML, 2005.

[11] Google Inc. *Google Maps*. <http://maps.google.com/>, HTML, 2006.

[12] Gorlick, M. *Streaming State Kinematics and Flow Engineering*. UCI Institute for Software Research, Technical Report UCI-ISR-06-3, March, 2006.

[13] Hood, E. *MHonArc*. <http://www.mhonarc.org/>, HTML, 2004.

[14] Larson, S.M., Snow, C.D., et al. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. In *Computational Genomics*. Horizon Press, 2002.

[15] Nottingham, M. Understanding Web Services Attachments. *BEA Dev2Dev*. May 24, 2004.

[16] Shanahan, F. *Zollage*. <http://www.francisshanahan.com/collage/bezos/bezos_b.aspx>, HTML, 2005.

[17] Shieh, A., Myers, A.C., et al. Trickles: A Stateless Network Stack for Improved Scalability, Resilence, and Flexibility. In *Proc. of the Symposium on Networked Systems Design and Implementation*. Boston, MA, May, 2005.

[18] Stoica, I. *Stateless Core: A Scalable Approach for Quality of Service in the Internet*. PhD. Thesis. Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[19] The Apache Software Foundation. *mod_mbox*. <http://httpd.apache.org/mod_mbox/>, HTML, 2006.

[20] Young, M. *AP News + Google Maps*. <http://www.81nassau.com/apnews/>, HTML, 2006.