Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture" (Impact Paper Award)

Roy T. Fielding
Adobe
USA
fielding@gbiv.com

Richard N. Taylor
Institute for Software Research
University of California, Irvine
USA
taylor@uci.edu

Justin R. Erenkrantz
Bloomberg
USA
justin@erenkrantz.com

Michael M. Gorlick
Institute for Software Research
University of California, Irvine
USA
mgorlick@acm.org

Jim Whitehead
Dept. of Computational Media
University of California, Santa Cruz
USA
ejw@ucsc.edu

Rohit Khare
Google
USA
rohit@khare.org

Peyman Oreizy
Dynamic Variable LLC
USA
peyman@oreizy.com

ABSTRACT

Seventeen years after its initial publication at ICSE 2000, the Representational State Transfer (REST) architectural style continues to hold significance as both a guide for understanding how the World Wide Web is designed to work and an example of how principled design, through the application of architectural styles, can impact the development and understanding of large-scale software architecture. However, REST has also become an industry buzzword: frequently abused to suit a particular argument, confused with the general notion of using HTTP, and denigrated for not being more like a programming methodology or implementation framework.

In this paper, we chart the history, evolution, and shortcomings of REST, as well as several related architectural styles that it inspired, from the perspective of a chain of doctoral dissertations produced by the University of California's Institute for Software Research at UC Irvine. These successive theses share a common theme: extending the insights of REST to new domains and, in their own way, exploring the boundary of software engineering as it applies to decentralized software architectures and architectural design. We conclude with discussion of the circumstances, environment, and organizational characteristics that gave rise to this body of work.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3121282

CCS CONCEPTS

• **Software and its engineering → Software architectures**; • **Information systems → RESTful web services**; • **Networks** → *Application layer protocols*;

KEYWORDS

REST, Representational State Transfer, WebDAV, ARRESTED, CREST, COAST

ACM Reference format:
Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. 2017. Reflections on the REST Architectural Style and "Principled Design of the Modern Web Architecture" (Impact Paper Award). In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3121282

1 A BRIEF HISTORY OF THE WEB, REST, AND ITS FORMULATIONS

The Web's initial architecture, as conceived by Berners-Lee in 1989 and implemented from late 1990-91, consisted of federated client/server components bound together by common protocols: a human-readable addressing system (URI), a simple mark-up language for hypertext (HTML/1.0), and a trivial protocol for transferring a hypertext document over TCP/IP (HTTP/0.9)[5].

Although rudimentary, the early Web's low entry barrier and use of existing Internet protocols were enough to demonstrate that a wide variety of information systems could be combined under a common hypertext interface.

By 1993, the Web had piqued the interest of computer science departments as well. NCSA introduced a new browser, Mosaic, that

was user-friendly and easy to install. HTTP was extended to carry an email-based message format, supporting non-HTML documents (e.g., images) and metadata, and a variety of new methods were proposed.

In 1993, the number of public Web servers grew at an exponential rate, doubling every three months, and continued at that torrid pace for over three years. Between the growth of commercial interest in the Web and the rate at which extensions were being introduced, success was tearing the Web's development community apart [2].

Roy Fielding became involved in the Web Project while doing research on distributed information services in 1993. He developed and published open source tools for Web maintenance, including `wwwstat` (logfile analytics) and `MOMspider` (a maintenance robot) [14], and created an open development project for `libwww-perl` (a Web client library written in the Perl language) out of MOMspider's internals.

After speaking about MOMspider at the First International WWW Conference, Fielding contributed to the standardization of HTML/2.0 (at one point reorganizing the entire specification to improve progress) and resolved an issue blocking Web addresses by authoring a separate standard for relative URLs [15]. When it came time to standardize HTTP, he wrote the charter for the IETF working group and became editor of the HTTP/1.x specifications with Henrik Frystyk Nielsen of CERN/W3C.

REST was born as a byproduct of the collaboration between Fielding and Nielsen while working on the HTTP specifications, pruning HTTP/1.0 to the essential bits and evaluating various ideas (their own and others') for a future HTTP/1.1. Fielding developed a model for ideal Web application behavior, initially called the *HTTP object model*, as a test case/oracle for understanding how changes to the protocol might impact the best applications on the Web. The "best," in this case, did not mean which applications were popular among users, but rather which ones resulted in a better Web: e.g., resilient to adverse network conditions, evolvable over time, and having the effect of increasing the Web itself (by encouraging the creation and identification of resources for reuse by others).

By 1995, many of the free software projects that had made the Web successful were gradually fading away. The NCSA httpd (web server) appeared to be abandoned, so Fielding joined a group of seven other webmasters in founding the Apache HTTP Server Project [17], an open development project dedicated to preserving a Web based on open standards.

The Apache server was redesigned to support a processing model and API for independent extensibility, allowing the core group to focus on platform features (like HTTP) while the extended community built new features on top of the modular API. In less than a year it had become the most popular server software for the Web. It played an important role in the standardization of HTTP/1.1, since IETF standards are based on a tradition of *rough consensus and running code*. Apache received the ACM Software System Award in 1999 for its contributions to the Web, its innovative architecture, and the pioneering way in which it was developed as a collaborative open source project.

It was only after HTTP/1.1 was finally published [18], in 1997, that Fielding began research on how to describe the HTTP object model in his doctoral dissertation. Unfortunately, "object model" was the wrong term. After talking to a few of his colleagues, it quickly became clear that the model was actually an architectural style—an abstraction across many specific application architectures [43]—and its use as a test oracle for HTTP was the same as evaluating whether a proposed change was an architectural mismatch for that style, and thus a potential problem for the best Web applications. Fielding changed the model's name to Representational State Transfer (REST) and set to work on its description as an architectural style.

The first version of what eventually became "Principled Design of the Modern Web Architecture" was submitted to FSE99. It was rejected, with reviewer comments including "Over all, the originality of the paper is quite low. There is only little to learn from it." and "- the web is old technolgoy [sic] now. - lots of jargon make the paper difficult to understand. ... - I can't find a novel lessons [sic] for software engineers in this paper."

Not dissuaded, the authors revised the paper and submitted it to ICSE 2000, held in Limerick, Ireland, in June of 2000 [20]. Fielding defended and published his dissertation in September [16] and, in 2002, a journal version of "Principled Design" appeared in ACM Transactions on Internet Technology [21], substantially enhancing the ICSE version and incorporating material from the Ph.D. dissertation.

Today "REST" and "RESTful architecture" are widely used terms, and sometimes even used appropriately. REST's influence can still be seen in the current standards for HTTP/1.1 [19] and URI [3]. Fielding's dissertation has been cited over 6,000 times, according to Google Scholar; the ICSE/TOIT paper, over 2,000 times. Over the past decade, O'Reilly & Associates alone has published 30 books with "REST" in the title; Amazon has 100 more. Crunchbase lists 2,000 startups with an "API" in their descriptions; about 50 specifically highlight "REST APIs".

This paper explores REST in a little detail, then proceeds to discuss common misunderstandings about the work and perceived shortcomings. A majority of the paper, however, is devoted to surveying what the REST work has inspired, and how that work advanced in new directions. Thus, this paper includes not only the original authors, but many other graduates of the same degree program, under the same advisor (Taylor), who have worked as colleagues in the exploration of software architecture and architectural styles for decentralized systems. The final section is devoted to the meta-issues of this research, namely what funding and organizational characteristics enabled the REST-related work to flourish.

2 JUST EXACTLY WHAT IS REST?

In spite of the formal publications, there has been a surprising amount of discussion focused on what REST is, and is not. The Wikipedia article on Representational State Transfer [57] has over 4,000 edits, reflecting growth as well as controversies. A decade ago, overzealous students of the style were even dubbed RESTafarians, while extensive debate raged between REST and so-called "Web Services" based on object-oriented RPC styles ("WS-*"). More recently, a series of seven annual international workshops have been held on "RESTful Design.", and just five years ago, "RESTful web services" was inducted into the 2012 ACM Computing Classification System (CCS).

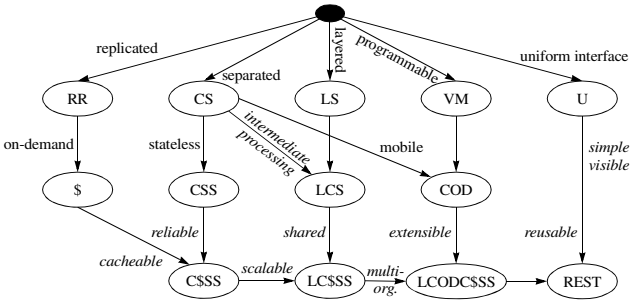

Figure 1: Derivation by style, constraints, and *properties*. [21]

In some cases the confusion stems from willful disregard of the substance and nuances of REST; in other cases it is the result of misunderstandings. In the discussion below we sketch the evolution of the definition of REST. In many respects the evolution of REST resembles how mathematical theories become more carefully and artfully articulated over time.

2.1 Formulation in Dissertation (2000)

Fielding's dissertation [16] is the original and most widely cited description of REST.

As an architectural *style* for network-based applications, its definition is presented in the dissertation incrementally, as an accumulation of design constraints that derive from nine pre-existing architectural styles and five additional constraints unique to the Web. Figure 1 shows the style derivation graph for REST and highlights the associated constraints and induced *properties*. Each style induces specific architectural properties, some positive and some negative (a.k.a., trade-offs). Some of the styles are implied by the Web's requirements; others were chosen for their beneficial properties, or to counteract the trade-offs of another style. The detailed discussion of this derivation can be found on pages 76-86 of the dissertation.

REST's five uniform interface constraints, as detailed in [16] are as follows:

- All important resources are identified by one resource identifier mechanism (induces simple, visible, and reusable);
- Access methods have the same semantics for all resources (induces visible, scalable, and available by enabling application of layered system, cacheable, and shared caches styles);
- Resources are manipulated through the exchange of representations (induces simple, visible, reusable, cacheable, and evolvable via information hiding);
- Representations are exchanged via self-descriptive messages (induces visible, scalable, and available by enabling application of layered system, cacheable, and shared caches styles, and evolvable via extensible communication); and,
- Hypertext as the engine of application state (induces simple, visible, reusable, and cacheable through data-oriented integration, evolvable via loose coupling, and adaptable though late binding of application transitions).

The ICSE 2000 and the TOIT papers used a similar formulation, albeit much more tersely in the ICSE paper.

2.2 Alternative Formulation at FSE (2007)

In 2007, three of us undertook an effort to more succinctly characterize REST, and articulate it in a manner less susceptible to misunderstanding by the software engineering community (as opposed to the network protocols community). This investigation was also the result of our experience as developers struggling to build web applications conforming to the REST style. We discovered 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.

This led to a statement of six key constraints that we first articulated in [13]. This formulation was later used as the basis for the style's presentation in the Taylor, Medvidovic, and Dashofy textbook on software architecture [51]. That book also includes a discussion of the derivation of REST from simpler styles, albeit with some differences from Fielding's original derivation graph shown in Figure 1.

2.3 Discussion

To this day differences exist among the authors regarding what is the "right" or "best" definition of REST. To some extent this is due to terminological preferences, based on the target audience. To some extent it is a matter of level of detail; the presentation in [13] has the virtue of succinctness; it also has the same vice.

REST is not an architecture, but rather an architectural style. It is a set of constraints that, when adhered to, will induce a set of properties; most of those properties are believed to be beneficial for decentralized, network-based applications, while others are the negative trade-offs that can result from any design choice (any constraint implies that a designer's space of choices is reduced). REST does not directly constrain the Web's architecture. Rather, an application developer may choose to constrain an architecture in accordance with the REST style. There is no way to force adherence to the REST constraints, though some poorly considered applications might not work well without them.

3 LESSONS FROM EARLY EXPERIENCE

3.1 Session Management

One early use of the Web was to support e-commerce. In this context the use of so-called "shopping carts" arose, wherein an end user would incrementally add indicators of merchandise to a list for subsequent purchase. Where and how this list should be maintained in the Web was unclear to various developers (client? server?) , and a range of solutions was developed. While some of these were RESTful, others were not. Lack of attention to this use case opened the door to popular but unfortunate solutions, such as cookies. This topic is but an instance of dealing with sessions, full treatment of which is outside the scope of this paper. Note, however, that session management can be an attempt to approximate concurrency, a challenge addressed by WebDAV, and is discussed in §4.2 below.

3.2 Namespaces, Resources, and Representations

We explored several systems — a web-based mail archiver (`mod_mbox`) and a version control system (Subversion) — of which we had been

involved in the design and implementation [11, 13]. One of the critical lessons demonstrated was the importance of the structure of the namespace (URL) in REST transactions and the value of decoupling resources from representations. As an architectural style, REST alone was neither sufficiently expressive nor definitive to guide the implementation. `mod_mbox` required two additional constraints beyond those dictated by REST: dynamic representations of the original messages and the definition of a consistent namespace.

However, these constraints depended upon an understanding of the content itself — a generic approach was inefficient. As `mod_mbox` was at its core a mail archiver, we could leverage the properties of the mail messages themselves to improve the modeling of the presented namespaces. To achieve this consistent namespace, `mod_mbox` relied upon the message's metadata (in this case, the `Message-ID` MIME header). On arrival into the archive, only a metadata entry is created for a message M. Consequently, if the metadata index was ever recreated, the URLs of the resources (messages) remain constant — guaranteeing the long-term persistence of links.

Instead of creating HTML representations as messages arrive, `mod_mbox` defers that transformation until a request for a specific message is received. Only later, when message M was fetched from the archive by a user-agent, was the HTML representation of M generated (with the help of $M's$ metadata entry). This sharp distinction between the resource and its representation minimized the up-front computational costs of the archive — allowing `mod_mbox` to gracefully handle more traffic than other contemporary systems.

3.3 Interplay with Application Architectures

The saga of Subversion speaks 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 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 (including one of the authors), failing to appreciate the performance penalty, did not implement pipelining, and fetched resources serially. Unsurprisingly, the network performance turned out to be unacceptable. The critical alteration was the use of independent data streams ("buckets") 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.

This, in turn, eliminated the overhead of XML encoding and permitted the reintroduction of simple caching intermediaries.

This suggests that the benefits of REST may be difficult to realize unless the individual web participants align their internal architectures to accommodate both asynchronous communications and concurrent computations.

Then-emerging web development techniques, such as AJAX and "mashups," suggested a pivotal 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 serverside. 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. When viewed from the perspective of the browser, at an abstract level the innovation of AJAX is the transfer, from server to client, of a computation whose execution is deferred client-side.

With these examples in mind, we reexamined REST, reformulating and expanding the core REST principles and constraints to accommodate the recent evolution of the web in CREST, §4.5.

4 WHAT REST HAS INSPIRED, AND WHERE IT HAS LED

Several generations of doctoral researchers extended the insights of REST to explore novel architectural styles that support properties required by complementary innovations beyond the classic Web model.

4.1 Web-based Development of Complex Information Products

Our interest in developing and extending REST and the modern web architecture was strongly motivated by a desire to have this infrastructure support large scale software engineering efforts. This agenda was outlined in a 1998 *Communications of the ACM* article which presented requirements and a technical agenda that would support the development of complex information artifacts via the web [22]. Key elements of the technical agenda were support for first-class hypermedia links, a scalable notification architecture, and support for remote collaborative authoring and versioning (WebDAV—see the following section). Our goal was to provide support for these services within the web infrastructure, consistent with the REST architectural style.

This approach was guided by many shared technical assumptions, many of them implicit. In the late 1990s, client-side JavaScript and manipulation of the HTML document object model (DOM) were not especially performant, and there were significant differences in JavaScript library capabilities within browsers. Due to this, we assumed that many complex document types would be supported either via large desktop applications, or specific plugins within the browser. The applications mostly did not support real-time interactive authoring (multiple collaborators in the same document at the same time), and it seemed unlikely they would soon add this support. HTML-based web pages were assumed to have limited capacity for supporting editing, via HTML forms. Notification services would require a distributed architecture in order to achieve Internet scale, such as the SIENNA distributed notification service [7].

Bi-directional links were initially supported within HTTP via LINK and UNLINK methods; they were not widely adopted, and were later removed [4]. In a different approach, WebDAV supported links via metadata properties defined on resources [25]. This approach was also infrequently adopted. Links as interoperable first

class computational agents were never supported in a standards-based way. Why so little love for bi-directional and first-class links? First, browsers never provided UI support for these links, in part because they don't fit the embedded link style of HTML. Where would these links appear in a browser window? Most applications of first-class, bi-directional links tend to fall within a single server's zone of control. Hence, first-class links embedded in HTML documents can be implemented in a rich, application-specific way as a traditional database-backed web application. For example, GitHub provides a richly hyperlinked version control and issue tracking environment implemented as a web application. Link shortener and link redirection services such as Bit.ly and PURL (persistent URL) tackle another use case for first-class links, the ability to change an endpoint without breaking the link.

With our focus on supporting software engineers, we insufficiently appreciated the importance of web site metrics and ad tracking as drivers of the web infrastructure. The ability to reliably track web browsing sessions and produce a range of metrics about these sessions drove the need for notification services. Tracking of browsing sessions takes place via JavaScript code which fires off notification messages on page transitions and other noteworthy events in a session, such as a click to purchase an item (this is the approach used by Google Analytics). Due to the value of this information, there is strong financial motivation to build large scale notification services within a single organization—no decentralization is necessary. These notification services, which began as efforts to track browsing sessions, have morphed into flexible services able to send a broad range of notification types and supported by an emerging standard [54].

The emergence of AJAX began a shift towards web applications with increasing amounts of client-side computational capacity and responsibility for maintaining interactive graphical interfaces. AJAX shone a bright light on the need for improved JavaScript performance, which led to rapid improvement in computational speed and cross-browser library consistency. By 2005-6, the technical capacity of the web browser had improved to the point where simplified word processor and spreadsheet capability could be provided in a client-side JavaScript applications. Initially launched by Google Docs and Sheets, other vendors (Zoho Office, Microsoft Office 365, Cacoo, Overleaf, etc.) have followed. These applications support real-time multi-person collaborative editing via the use of operational transforms [49], which requires development of the editor from the ground up to support this feature. This combination seemed highly unlikely to us in the late 1990s: dramatic improvements in JavaScript and the massive engineering effort required to re-engineer editors from the ground up to support real-time collaboration. As a result, our approach to collaborative authoring was focused on desktop-applications and plugins, and our approach to concurrency control was focused on whole-document locking to support turn-taking collaboration using these legacy applications.

In retrospect, our analysis of requirements was on-point—today's web does in fact have first-class links, notification services, and collaborative authoring which support the development of large scale information artifacts. However, we were off on the specifics of how this might play out. In the case of first-class links, perhaps we should have known better. Even in 1998 there was ample evidence of the limited interest in supporting this feature, since no browsers provided support, and deployment of LINK/UNLINK was rare. But, with notifications and real-time collaborative authoring, it appears the main determinant was economics. One can imagine an alternate path not taken where web site analytics were provided only via server-side services and not via a centralized service, or companies were unwilling to make significant investments in real-time collaborative editing, which until the mid-2000s had been a niche feature with uncertain future.

4.2 WebDAV

Web Distributed Authoring and Versioning (WebDAV) is a series of extensions to HTTP to provide remote authoring and version control for web resources (initial specification: [25], revision: [10]). The core WebDAV specification provides services for writing to resources, reading/writing metadata (properties) on these resources, and a lock-based concurrency control model.

WebDAV highlights a core assumption of REST, that the vast majority of information flow is from server to client in the form of resource representations. This core assumption informs many aspects of REST, including sending representations of state, caching, and stateless interactions between client and server. WebDAV had to circumvent these aspects of REST in order to achieve its remote authoring goals.

Consider the challenge of writing to a web resource. In REST, what is transmitted across the wire is a *representation* of the raw source of a resource consistent with some standard data type, not the raw resource itself. However, the author of a resource wishes to modify this raw resource directly. The raw source could be quite different from its representation, such as when a web page is created as the output of a program (e.g., a PHP script) running on the server, where the representation is in HTML, and the raw source is source code. WebDAV initially proposed creating a link in the resource metadata that potentially points to a separate location where the raw source could be directly modified and potentially read. This was not widely implemented, and was subsequently removed. This effectively limits WebDAV remote authoring to situations where there is a nearly direct correspondence between raw source and on-the-wire representation.

RESTful caches do not interact well with writing to a resource's raw source. Since a cache is only responsible for maintaining a copy of a resource representation, it has no knowledge of its raw source, or ability to modify it. Consequently, the raw source must be directly modified on the original server that generated the initial, cached resource representation, thereby requiring an authoring client to bypass all caches. Even if an authoring client were to try writing the raw source via a series of cooperating caches, there is no guarantee the raw source would be the same as the representation, and hence caches would not be able to proactively update their cache state with what is being written. Unlike, say, a memory cache which can update on read or write, RESTful caches can update only on reads.

The REST constraint of maintaining stateless interactions between client and server placed strains on the design of WebDAV. One example is WebDAV's lock-based concurrency control. A typical approach to resource locking involves creating a session, then establishing the lock within the session. When the session ends, the

lock disappears. Since sessions are inherently stateful, an alternate approach was required. WebDAV locks create globally unique identifiers called lock tokens which are used in subsequent requests to identify the lock. In this way protocol requests are kept stateless by using an identifier to refer to the lock's persistent state on a server.

Overall, the same RESTful architectural constraints which strongly contributed to the success of the modern web architecture also increased the difficulty of creating an interoperable authoring protocol using HTTP. The process of creating WebDAV did lead to a deep understanding of the design space of hypertext versioning systems, captured in Whitehead's dissertation [55] and in [56].

4.3 Dynamic Software Architectures

The Web's architecture is continually changing as clients, servers, proxies, and gateways join and leave the system. These components are themselves continually changing to provide new capabilities, such as adding new resources in the form of novel websites and web services, supporting new representations for resources (e.g., novel image and video formats), integrating novel hardware devices, adding novel features to user agents (web browsers), etc. The scale, diversity, and rapidity with which these changes occur make it impossible to capture even a snapshot of the Web's current architecture.

This malleability did not emerge by accident. It is a direct consequence of the constraints imposed by the REST architectural style. Specifically,

- URLs provide an anarchic, decoupled namespace with no central authority and each Web server may support whatever URLs it chooses and assign them whatever meaning it deems appropriate. This freedom to introduce URLs and resources is partially responsible for the outpouring of innovative Web applications and services.
- Representations for resources may evolve in an ad hoc manner with no central authority since components can negotiate with one another to pick a mutually suitable representation. The metadata enables intermediaries and the receiver to inspect and determine how to process a resource.
- Context-free interaction demands that all state be externalized. A request (to a server, for example) must carry whatever state is necessary for that server to be able to process it, without recourse to any prior history of interaction.
- A small set of well-defined methods keeps a low barrier for introducing new processing components.
- Idempotent operations and the presence of intermediaries support scalability.

Encouraged by the Web's malleability and the principal role that software architecture and architectural style played in helping to realize it, we wondered if a similar approach applied to the *internal* architecture of a single component (a single program or application) would engender a similar degree of malleability.

The ability to change an application's behavior *during* runtime is an increasingly important capability, both to support continuous operation of critical systems and to support a good user experience. Our approach ([40] and later refined in [39, 42]) centered on deploying an application with an explicit model of its own architecture (in terms of components and connectors, the interconnections between those components and connectors, and their mapping to implementation modules) plus a reusable runtime infrastructure that used the implementation mapping to (1) maintain consistency between the application's model and implementation and to (2) prevent changes that would violate the application's architectural constraints. We applied our approach to several proof-of-concept applications built in the C2 style [50], a layered, event-based style where components communicate exclusively by passing events thru active, first-class connectors. Our applications exhibited a surprisingly powerful and flexible degree of runtime adaptability.

After further experience and reflection, we broadened our attention to consider approaches at different levels of abstraction, and devised a framework to help us evaluate, compare, and combine these techniques. Our framework differentiates techniques based upon the *system model* they operate on (e.g., microprocessor instructions executed by a CPU, bytecode executed by a Java Virtual Machine, an architecture model mapped to its implementation modules) and how they confront four aspects of runtime change:

- *Behavior* concerns how the behavioral specification of the system is changed. E.g., are changes restricted to the recombination of existing behaviors or can novel behaviors be introduced, how are changes represented, deployed, and verified?
- *Asynchrony* concerns how a change is applied over time. E.g., is the system's execution suspended during changes or does it continue to execute, potentially in some limited capacity?
- *State* concerns how the system's state is changed, whether in memory, on disk, or in a separate subsystem, such as a database. E.g., is all state changed in unison or lazily as accessed; is the system's execution suspended while changes are made?
- *Execution context* concerns how the state of the machine interpreting the behavioral specification is changed. E.g., reordering a function's bytecode may not be possible while the interpreter's stack holds a reference to the function.

We refer to our framework as BASE [41, 52] and used it to characterize several popular architectural styles, including REST, C2, CREST (described in §4.5), MapReduce, Pipe-and-Filter, Event Notifications, and others. Evident in this analysis are several common leverage points used to achieve adaptability, namely:

LP1 making the parts that are subject to change identifiable, discrete and manipulable;
LP2 providing mechanisms for controlling interactions between the parts subject to change; and,
LP3 providing techniques for managing state

Returning to the REST style, we can readily identify its use of these leverage points to achieve runtime adaptability.

LP1 · clients, servers, proxies and gateways are discrete entities communicating via generic interfaces, allowing them to change
· messages are targeted at conceptual resources, allowing the realization of the resources to change
LP2 · components communicate by passing a representation of the resource, allowing the raw representation of the resource to change

· the format used for a resource's representation is late-bound, allowing the format to change or depend on the capability of the recipient or the characteristics of the request
· the generic resource interface hides implementation details, allowing communication mechanisms to change
· metadata accompanies the resource's representation, allowing caches and gateways to intervene
LP3 · each request contains all of the information necessary for a connector to understand it, allowing connectors to change, or choose to process requests serially or in parallel, or choose whether or not to intermediate.

4.4 Decentralized Consensus: ARRESTED

What were once hyped as "peer to peer" (P2P) systems, or now promised as "dApps" (decentralized apps on the blockchain), can't be characterized effectively within Client/Server architectural styles. The rise of instant messaging services, mobile push notifications, and social networking was reflected in the research themes of annual workshops at Irvine from 1998-2000 on event notification, namespaces, and decentralized organizations [30]. As we explored these complementary innovations around the Web, we concluded that real-time, Internet-scale event notification — group messaging that can be initiated by any party, at any time — highlighted three limitations of REST's request-response model:

- *One-shot:* Every request can only generate a single response. If that response message is an error (or lost), there is no recovery protocol.
- *One-to-one:* Every request proceeds from one client to one server. Instead of routing to a group at once, a chain of proxies passes it to each.
- *One-way:* Every request must be initiated by a client, and every response must be generated immediately, precluding servers from sending asynchronous notifications.

While REST presumes centralized resources (within a decentralized Web), we extended it to induce properties required by distributed and decentralized resources, such as Group Consensus and Simultaneous Agreement. The ARRESTED style [33] used four new building blocks: events, routes, locks, and estimates for corresponding constraints on Asynchrony, Routing, Delegation, and Estimation. Compared to the most common alternative, polling (REST+P), these styles offered tighter bounds on latency, supported larger groups, and degraded more gracefully when networks or services fail.

Together, these additional constraints can meet the BASE requirements[1] of decentralized systems: to only presume *Best-effort* network messaging; to *Approximate* the current value of remote resources; to be *Self-centered* in deciding whether to trust other agencies' opinions; and *Efficient* when using network bandwidth.

We identified two fundamental factors limiting the feasibility of consensus: latency and agency. Figure 2 maps a family of new styles against the 'now horizon' (components that can refer to the value of a variable 'right now') and agency boundaries (components that can trust each other). In other words, the now horizon separates consensus-based styles from consensus-free ones; and the agency boundary separates master/slave styles from peer-to-peer ones.

[1]Yes, an entirely different BASE than in the previous section...

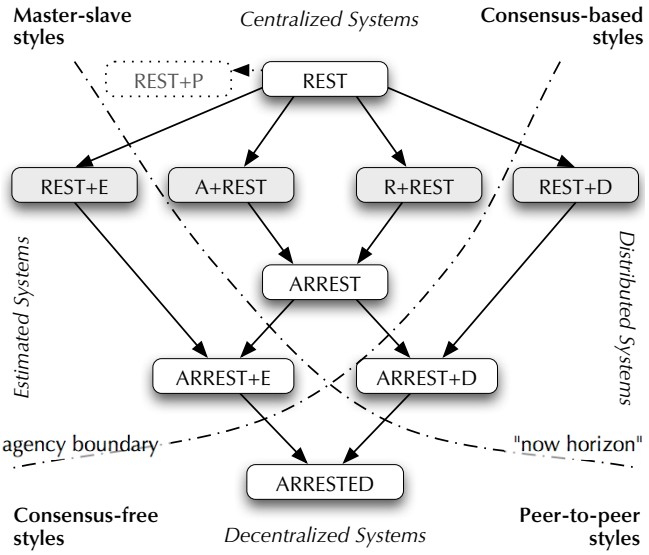

Figure 2: Diagram summarizing four new architectural styles, derived from four capabilities added to REST [32].

4.4.1 Assessments. Since then, Web applications have added several complementary features for real-time and group communication, such as WebRTC, Websockets, Webhooks, and HTTP/2 streaming. New use cases for push messaging to mobile apps and the Internet of Things (IoT) continue to proliferate. Internet-scale event notification services are available for content distribution (fast.ly, a CDN with near-real-time global invalidation), service integration (Apache Kafka, Amazon Kinesis, and Google Cloud Pub/Sub), and lightweight reactive programming platforms (Amazon Lambda, IFTTT, and AI 'assistants').

Nonetheless, centralized systems still dominate the Web. Amazingly, even planetary-scale databases like Spanner [8] became practical with the introduction of a bounded-error TrueTime service: a practical rebuke to the imaginary GlobalClock that ARRESTED approximates. Commercially, centralized networks also dominate two-sided markets, from auctions to advertising to payments. The promise of federated social networking across agency boundaries remains just that, in the face of addictive 'news feeds' based on machine learning techniques that are only feasible over centralized clickstreams (though Federated Learning [6] could change that).

4.4.2 Disruptions. Of the algorithms ARRESTED advocated, consistent hashing is now commonplace in NoSQL databases such as Cassandra [34]; and Merkle hash trees are still a practical way to create trust between suspicious agencies. However, most applications are within organizations — BitTorrent remains the most prominent use of Distributed Hash Tables (DHTs) *across* agency boundaries [58] with little evidence of success as public, shared infrastructure [44].

The authors did not anticipate anything like Bitcoin or the blockchain [37], perhaps the signature achievement of 'decentralization' over the past decade. An explosion of new systems are exploring the space of new possibilities for the technology beyond its origins as a cryptocurrency protocol [35]. We expect entirely new

architectural styles to emerge around what the Ethereum [59] calls "dApps" [9] that run directly on community-contributed computing resources.

4.4.3 Recentralization. Ultimately, our goal was to identify styles that could build *software that works the way society works*. The "way society works" has turned out to be far more *re*centralized than decentralized, though. When REST was being framed, it seemed inconceivable that two billion people would all agree to use one website (Facebook); or that "search engines" would index the entire public Web; or that advertising networks that match marketers to content publishers or app developers would be embedded across large swaths of the Web. When a decentralized alternative for source-control took off (git), "society" still adopted a centralized repository of incredible scale (Github). To the degree that RESTful designs have enabled the entire Software-as-a-Service (SaaS) industry to disrupt how custom software is developed and deployed, that has led to abundant choices *between* competitors — but almost no choices for sticking with last week's version, because SaaS subscription models continuously upgrade all their tenants on a centralized basis.

4.5 Computation Exchange: CREST

REST addresses Internet-scale hypermedia, but our FSE 2007 paper [13] was the gateway to a different vision of the web, one where Internet-scale computation exchange rather than content exchange, dominates web activity. To provide developers concrete guidance in the implementation and deployment of computational exchange, we offered 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.*
- 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.*
- CP3 *All computations are context-free.*
- CP4 *Only a few primitive operations are always available, but additional per-resource operations are also encouraged.*
- CP5 *The presence of intermediaries is promoted.*

The evolution of REST to CREST began in 2006–2007 when Erenkrantz and Gorlick, like Fielding before them, turned to the web as a living laboratory [12]. As described earlier by Erenkrantz (Section 3.2) the macro-level constraints of REST had seeped down into application architectures, a confirmation of the prior work of Oreizy [38]. Moreover, study of other decentralized systems revealed constraints (or alternatively, principles of construction) that bore more than a passing resemblance to the context-free state transfers of REST. This strongly suggested that REST was but one member of a family of architectural styles whose instantiations had been hiding in plain sight all along and that variations in, or deviations from, REST were not necessarily flaws or shortcomings but merely examples of natural and useful, domain-specific variations.

Another web development also attracted their attention. Web mashups, introduced by Paul Rademacher in April, 2005[2] spread like wildfire. Though unexpected (at least to Erenkrantz and Gorlick; Fielding may have thought otherwise) mashups offered a fresh perspective on REST intermediaries. To their eyes, mashups mirrored continuations (a well-known construction in the formal semantics of programming languages [53] and a control mechanism in several programming languages such as Scheme and SML) in the sense that the client-side scripts and the "redirection" URLs they contained represented, from the perspective of the mashup host M, the "rest of the computation" (that is, a continuation) that M itself might have performed had it not been constrained by network latency and scaling.

Two other examples, from entirely different domains, also informed their view. The first was the work of David Halls [28] who explored the role of mobile code (implemented as the network transfer of continuations from one remote Scheme interpreter to another) in the construction of distributed systems. Hall's example of a web server and web client that exchanged continuations (embedded in HTTP requests and responses) elegantly solved three problems that REST failed to address: session management, cookie injection, and the inconsistent behavior of the browser back button in the presence of a session-specific cookie.

The second example came from Alan Shieh [45, 46] who, following the design pattern of Aura and Nikander [1], reconstructed TCP as a stateless-protocol (named Trickles) in which the initiator of the TCP connection and the listener that responded to connection initiation exchange "network continuations" that encapsulate all of the requisite TCP session state. The burden of maintaining session state is thereby transferred to the initiator as every transmission from initiator to listener is accompanied by the session state generated by the listener in the prior round trip. These REST-like constructions for TCP confer like advantages: substantial reductions in server-side state, trivial connection restart, and connection mobility in which the network locations of the Trickles endpoints can be shifted without loss of connection state.

From this backdrop arose the idiom of *computation exchange*, in which peers interact by exchanging and evaluating live computations (state plus code) in a REST-like framework. Computational REST (CREST) for computation exchange was hammered out in three intensive days of whiteboard discussions among Erenkrantz, Gorlick, and Girish Suryanarayana in Spring 2006.

The work on CREST, detailed in [13], and Erenkrantz's doctoral thesis [11], set benchmarks for the analysis of REST-like systems. This presentation eased understanding, promoted cross-comparison among related styles, and encouraged reasoned analysis of the degree to which a system is RESTful.

Further, the reduction of REST to a terse constraint set laid bare the several independent axes of variation of REST, thereby allowing us to describe, with improved precision, the benefits that the constraints, both individually and in combination, conferred upon conforming architectures — the fundamental defining characteristic of an architectural style. Here we drew upon the prior work of Oreizy discussed in §4.3 in which *dynamic* architectures are expected to define what is and what is not dynamic; on this hinges the distinctions among the members of a family of dynamic architectures.

In retrospect, the formulation of CREST perhaps leaves too much unsaid, but nonetheless CREST took REST-like systems in a new direction by emphasizing the primacy of computation over content and relegating content to a side-effect of computation. In this way

[2] See http://www.housingmaps.com/ and https://forums.craigslist.org/?ID=26638141

CREST reflects the idiom of computation exchange: it elevates computations to first-class representations of a resource and designates context-free state exchange (including computational state reified as closures, continuation, or binding environments) as the sole form of information exchange among clients and servers. As detailed in [11, 13] CREST resolved several outstanding puzzles in the evoluton of the web including web mashups, session management, the (misplaced) role of cookies in client/server interactions, and the rationale for time-dependent resources such as weather forecasts or time-series responses like a stock ticker.

4.6 Computation Exchange with Security: COAST

Many reviewers of CREST observed that exchanging and evaluating computations (mobile code) among peers appears patently unsafe, leaving peers open to service theft or denial of service attacks, and easy prey for hostile takeovers where the peer is used as a launchpad for attacks against other peers in the network. COmputAtional State Transfer (COAST) also pursues the idiom of computation exchange but directly addresses these concerns, this time cast in an architectural style where security and peer safety are first-order concerns [26].

Under COAST the exchange of live computations (state + code) is the principal form of interaction among peers. All COAST exchanges rely on communication by introduction, meaning that a peer x can communicate with a peer y only if peer x holds a Capability URL (CURL) for y. CURLs are cryptographic structures; they are tamper-proof and cannot be guessed or counterfeited. Live computations received by peers via CURLs are evaluated in the context of execution sites, flexible sandboxes that confine the functional and communication capability of visiting computations. These four fundamentals: communication by introduction, live computations, execution sites and CURLs, are sufficient to protect against many common security threats including unwanted intrusion, resource theft, or gross abuse of capability. These same four concepts also account for a considerable degree of adaptation and flexibility. More broadly, the COAST architectural style embeds computation exchange in the object-capability model of security [36]; both computation exchange and object-capability contribute in equal measure to security and adaptation.

For any form of computation exchange there are two fundamental issues: communication and confinement; that is, how independent computations contact one another and exchange information and how their executions are confined to prevent damage to their hosts or other computations. The COAST style defines four rules: one each for services, execution, messaging, and interpretation. *Services* specifies the form and content of communications: asynchronous messaging of live computations comprising primitive values, closures, continuations, and binding environments and implicitly the meaning of service: computation-specific interpretation of mobile closures, continuations, and binding environments. *Execution* defines execution sites as a basic mechanism for functional and resource confinement. *Messaging* regulates how communication capability is allocated among computations. In particular, there is no ambient communication capability; without CURLs a computation is mute and without egress points (a capability that confers the right to read (extract) messages from a unidirectional communication channel) a computation is deaf. Finally, *interpretation* specifies that message interpretation is not only receiver-dependent but also delivery-dependent; both the CURL denoting the ingress point (a capability that confers the right to write (inject) messages into a unidirectional communication channel) of the message and the consequent transmission trajectory of a message can influence the interpretation. Within the four corners of the style rules peers exchange and evaluate live computations, thereby receiving and transmitting messages that contain primitive values, closures, continuations, and binding environments.

Specifically, the COAST rules are:

- **Services:** All services are computations whose only interactions are the asynchronous messaging of primitive values, closures, continuations, and binding environments.
- **Execution:** All computations execute within the confines of some execution site $\langle E, B\rangle$ where E is an execution engine and B a binding environment.
- **Messaging:** Computation x can transmit a message to a computation y only if there exists a unidirectional communication channel t such that x holds a CURL u denoting an ingress point of t and y holds an egress point of t.
- **Interpretation:** The interpretation of a message delivered to computation y via CURL u is y- and u-dependent.

Early results from COAST are encouraging, including:

- Secure remote evaluation [47, 48] of computations and secure remote spawning of computations are natural consequences of the COAST style.
- Live update modifies the code, structure and data values of a running system in place without halting the system or interfering with it [29], including three distinct forms of *secure* live update with hot backup, that is, transparent service recovery in the event that the update fails [24].
- A novel form of system-level monitoring, *capability accounting*, that can be used for forensic analysis, penetration detection, early warning of attack, and testing, both functional and security-centric [23].
- Remote evaluation allows service providers to pare their service offerings to the bare minimum and shift the burden of rapidly evolving and refining service APIs from the provider to the clients. Using a web bookmark service as a test case we demonstrated a minimalist API (containing only three simple service primitives) that is extended per-client by client-generated live computations delivered provider-side for remote evaluation.
- Dynamic rearrangement of computations among hosts for the sake of performance, latency, or security [27].

5 REFLECTIONS ON THE RESEARCH ENVIRONMENT AND PROCESS

The development of the Web, REST, and the derivative technologies discussed above have clearly had an enormous impact. To be sure, REST and the other technologies did not emerge solely from the seven authors on this paper. Indeed, a very large number of individuals contributed to the numerous IETF standards and to the software systems that realized those standards. That said, UC

Irvine's Institute for Software Research (and its predecessor, IRUS) has played a prominent role in these developments, as it was the home institution for the work of this paper's authors.

The point of bringing up this old history is not to tout accomplishments or burnish medals. Rather it offers a chance to reflect on the *milieu* of software engineering research: how it is funded, conducted, evaluated, published, and transitioned.

The tale of REST, the Web, and the HTTP/1.1 protocol is certainly at odds with much current software engineering research practice. The work on these topics at ISR spanned more than a decade. In the early years of the work it was difficult to explain to funding agencies why the Web was a "big deal" and why they would later be glad to tout it as one of their signature accomplishments. The University of California, Irvine had a hard time understanding why one of ISR's Ph.D. students was taking close to a decade to finish his degree. Wasn't that "slowness" indication of "inadequate progress towards the degree"? And how was this "open source" thing actually going to produce production-grade software?

In hindsight it is easy to see that we made the right decisions — at the time it was a bit of a struggle to tell the tale well. The point to emphasize, though, is that the accomplishments required a relentless determination to make advances that had depth, integrity, quality, and value. REST did not result from a summer research project that produced a one-off solution that no one will ever actually use. Developing REST and HTTP/1.1 required tenacity and a dedication to quality. It required building substantial software of lasting value. Would that all software engineering research held to the same standards and values.

What was the environment at UCI-ISR that enabled such contributions to emerge? Fundamentally it was one in which students were given the authority to pursue topics that they found exciting and thought were potential game-changers. That authority was accompanied by funding that enabled them to travel — sometimes extensively — in support of standardization efforts. That funding was beyond what is typically available from the NSF and similar agencies. Rather, it was DARPA that provided the key funding, especially in the initial years of the project. (And DARPA, to its credit, did not demand quarterly proof of relevancy of the REST research to the top-level goals of the funding project; they let us run too.)

The authority to "run with it" was accompanied, to be sure, with responsibility. That responsibility took several somewhat atypical forms. First, the work, for many years, was conducted as part of the multi-institution Arcadia project [31]. The practical consequence of that was that every 3 or 4 months the students were obliged to present progress on their work to a small, vociferous, and sometimes cantankerous audience of other Arcadia researchers. More than once did students experience a rocky reception. In retrospect, however, most would agree that such frank commentary was essential in refining their work in important ways. Second, also a consequence of being part of Arcadia, there was a strong push for students to produce substantial software based upon their research, where that software would be appropriate for trial application in industrial contexts. Third, the students were responsible for writing and presenting their work in the appropriate academic forums, as well as to standards bodies, such as the IETF.

Is this kind of research generally possible in today's funding and publication climate? Will universities tolerate this kind of process? The answer to the latter seems pretty much, "no." With many universities demanding that students graduate in 5 years (or 4 years, in some countries), there is little chance of such projects being undertaken as a Ph.D effort. This type of work seems destined only for post-docs. But funding and publications? Impact must be brought to the fore. The ICSE 2000 paper had no surveys, no statistical analyses, and essentially no evaluation section. It merely stated:

> "The REST architectural style has been validated through six years of development of the HTTP/1.0 and HTTP/1.1 standards, elaboration of the URI and relative URL standards, and successful deployment of several dozen independently developed, commercial-grade software systems within the modern Web architecture."

Particularly ironic is retrospective consideration of the original reviews of the first FSE submission. They basically said that there is no value to be had in reflecting on a design, post facto, nor in clarifying, or assessing how the design principles worked out in one (important) instance of practice. To the contrary, there should be more of this.

ACKNOWLEDGMENTS

The support of DARPA over the critical years of this project was essential to its success. Our sincere appreciation especially to Bill Scherlis and the late John Salasin. Likewise our appreciation to the National Science Foundation for their years of support. The support of ISR's corporate sponsors was also critical, and is gratefully acknowledged.

Numerous people contributed to the Web, of course, though the REST community owes a particular debt to Tim Berners-Lee, Henrik Frystyk Nielsen, Dan Connolly, Dave Raggett, and Larry Masinter. Advocacy for REST within industry has almost entirely been the work of others, especially Mark Baker, Paul Prescod, Mike Amundsen, Leonard Richardson, Sam Ruby, and the late Aaron Swartz. Our work has benefited from interactions with several more generations of students and colleagues, at UCI and beyond, for whom we are grateful to have collaborated with, including Mark Ackerman, Ken Anderson, Greg Bolcer, Eric Dashofy, Nenad Medvidovic, Kari Nies, Jie Ren, Jason Robbins, David Rosenblum, and Girish Suryanarayana.

REFERENCES

[1] T. Aura and P. Niklander. 1997. Stateless Connections. In *Proceedings of the First International Conference on Information and Communication Security (Lecture Notes In Computer Science)*, Y. Han, T. Okamoto, and S. Qing (Eds.), Vol. 1334. Springer-Verlag, 87–97.

[2] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. 1994. The World-Wide Web. *Commun. ACM* 37, 8 (Aug. 1994), 76–82. https://doi.org/10.1145/179606.179671

[3] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. 2005. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. (Jan. 2005). https://doi.org/10.17487/RFC3986

[4] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. 1996. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945. (May 1996). https://doi.org/10.17487/RFC1945

[5] Tim Berners-Lee and Jean-Francois Groff. 1992. WWW. *SIGBIO Newsl.* 12, 3 (Sept. 1992), 37–40. https://doi.org/10.1145/147126.147133

[6] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2016. Practical Secure Aggregation for Federated Learning on User-Held Data. In *NIPS Workshop on Private Multi-Party Machine Learning*. https://research.google.com/pubs/pub45808.html

[7] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems* 19, 3 (Aug. 2001), 332–383.

[8] James C. Corbett and Jeffrey Dean et. al. 2012. Spanner: Google's Globally-distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

[9] Chris Dixon. 2017. Crypto Tokens: A Breakthrough in Open Network Design. (June 2017). https://medium.com/@cdixon/e600975be2ef

[10] L. Dusseault. 2007. *HTTP Extensions for Web Distributed Authoring and Versioning (WEBDAV)*. Request for Comments 4918. Internet Engineering Task Force.

[11] Justin R. Erenkrantz. 2009. *Computational REST: A New Model for Decentralized, Internet-Scale Applications*. Ph.D. Dissertation. University of California, Irvine, Irvine, California, USA.

[12] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. 2006. *Harmonizing Architectural Dissonance in REST-based Architectures*. Technical Report UCI-ISR-06-18. Institute for Software Research, University of California, Irvine.

[13] Justin R. Erenkrantz, Michael M. Gorlick, Girish Suryanarayana, and Richard N. Taylor. 2007. From Representations to Computations: The Evolution of Web Architectures. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE'07)*. 255–264.

[14] Roy T. Fielding. 1994. Maintaining distributed hypertext infostructures: Welcome to MOMspider's Web. *Computer Networks and ISDN Systems* 27, 2 (1994), 193 – 204. https://doi.org/10.1016/0169-7552(94)90133-3 Selected Papers of the First World-Wide Web Conference.

[15] Roy T. Fielding. 1995. Relative Uniform Resource Locators. RFC 1808. (June 1995). https://doi.org/10.17487/RFC1808

[16] Roy T. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine, California, USA. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[17] Roy T. Fielding and Gail Kaiser. 1997. The Apache HTTP Server Project. *IEEE Internet Computing* 1, 4 (July 1997), 88–90. https://doi.org/10.1109/4236.612229

[18] Roy T. Fielding, Henrik Frystyk Nielsen, Jeffrey Mogul, Jim Gettys, and Tim Berners-Lee. 1997. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068. (Jan. 1997). https://doi.org/10.17487/RFC2068

[19] Roy T. Fielding and Julian Reschke. 2014. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231. (June 2014). https://doi.org/10.17487/RFC7231

[20] Roy T. Fielding and Richard N. Taylor. 2000. Principled Design of the Modern Web Architecture. In *Proceedings of the 22nd International Conference on Software Engineering*. IEEE, Limerick, Ireland, 407–416.

[21] Roy T. Fielding and Richard N. Taylor. 2002. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology* 2, 2 (May 2002), 115–150.

[22] Roy T. Fielding, E. James Whitehead, Jr., Kenneth M. Anderson, Gregory A. Bolcer, Peyman Oreizy, and Richard N. Taylor. 1998. Web-Based Development of Complex Information Products. *Commun. ACM* 41, 8 (August 1998), 84–92.

[23] Matias Giorgio and Richard N. Taylor. 2015. *Accountability Through Architecture for Decentralized Systems: A Preliminary Assessment*. Technical Report UCI-ISR-15-2. Institute for Software Research, University of California, Irvine.

[24] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2013. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York City, New York, USA, 279–292.

[25] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. 1999. *HTTP Extensions for Distributed Authoring – WEBDAV*. Request for Comments 2518. Internet Engineering Task Force.

[26] Michael Martin Gorlick. 2016. *Computational State Transfer: An Architectural Style for Decentralized Systems*. Ph.D. Dissertation. University of California, Irvine, Irvine, California, USA. Available as Technical Report UCI-ISR-16-3.

[27] Michael M. Gorlick, Kyle Strasser, and Richard N. Taylor. 2012. COAST: An Architectural Style for Decentralized On-Demand Tailored Services. In *Proceedings of 2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA'12)*. 71–80.

[28] David Alan Halls. 1997. *Applying Mobile Code to Distributed Systems*. Ph.D. Dissertation. University of Cambridge, Cambridge, UK.

[29] Michael Hicks. 2001. *Dynamic Software Updating*. Ph.D. Dissertation. Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, USA.

[30] Irvine Research Unit in Software (IRUS). 1998-2000. The Workshop on Internet-scale Technology. (1998-2000). http://isr.uci.edu/events/twist/

[31] R. Kadia. 1992. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SDE 5)*. ACM, New York, NY, USA, 169–180. https://doi.org/10.1145/142868.143768

[32] Rohit Khare. 2003. *Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems*. Ph.D. Dissertation. University of California, Irvine, California, USA. http://www.ics.uci.edu/~rohit/Khare-Thesis-FINAL.pdf

[33] Rohit Khare and Richard N. Taylor. 2004. Extending the REpresentational State Transfer Architectural Style for Decentralized Systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society, Edinburgh, Scotland, UK, 428–437. http://www.ics.uci.edu/~rohit/ARRESTED-ICSE.pdf

[34] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf

[35] David Mazieres. 2015. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation* (2015). https://www.stellar.org/papers/stellar-consensus-protocol.pdf

[36] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[37] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008). https://bitcoin.org/bitcoin.pdf

[38] Peyman Oreizy. 2000. *Open architecture software: a flexible approach to decentralized software evolution*. Ph.D. Dissertation. University of California, Irvine, Irvine, California, USA.

[39] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, and David Rosenblum. 1999. An Architecture-based Approach to Self-Adaptive Software. *IEEE Intelligent Systems* 14, 3 (May-June 1999), 54–62.

[40] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 1998. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*. 177–186.

[41] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. 2008. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of 30th International Conference on Software Engineering (ICSE Companion 2008)*. ACM, 899–910.

[42] Peyman Oreizy and Richard N. Taylor. 1998. On the role of software architectures in runtime system reconfiguration. *IEE Proceedings-Software* 145, 5 (1998), 137–145.

[43] Dewayne E. Perry and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* 17, 4 (Oct. 1992), 40–52. https://doi.org/10.1145/141874.141884

[44] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. 2005. OpenDHT: A Public DHT Service and Its Uses. *SIGCOMM Comput. Commun. Rev.* 35, 4 (Aug. 2005), 73–84.

[45] Alan Shieh, Andrew C. Myers, and Emin G. Sirer. 2005. Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI'05)*, Vol. 2. USENIX Association, 175–188.

[46] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. 2008. A Stateless Approach to Connection-Oriented Protocols. *ACM Transactions on Computer Systems* 26, 3 (September 2008), 8:1–8:50.

[47] James W. Stamos and David K. Gifford. 1990. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering* 16, 7 (July 1990), 710–722.

[48] James W. Stamos and David K. Gifford. 1990. Remote Evaluation. *ACM Transactions on Programming Languages and Systems* 12, 4 (October 1990), 537–564.

[49] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Trans. Comput.-Hum. Interact.* 5, 1 (March 1998), 63–108.

[50] Richard N. Taylor, Nenad Medvidovic, et al. 1996. A Component- and Message-Based Architectural Style for GUI Software. *Transactions on Software Engineering* (June 1996), 390–406.

[51] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. 2010. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons.

[52] Richard N. Taylor, Nenad Medvidovic, and Peyman Oreizy. 2009. Architectural Styles for Runtime Software Adaptation. In *Proceedings of the Eighth Joint Working IEEE/IFIP Conference on Software Architecture and Third European Conference on Software Architecture*. IEEE Computer Society, 171–180.

[53] R.D. Tennant. 1976. The Denotational Semantics of Programming Languages. *Commun. ACM* 19, 8 (August 1976), 437–453.

[54] M. Thomson, E. Damaggio, and B Raymor. 2016. *Generic Event Delivery Using HTTP Push*. Request for Comments 8030. Internet Engineering Task Force.

[55] Emmet James Whitehead, Jr. 2000. *An Analysis of the Hypertext Versioning Domain*. Ph.D. Dissertation. Univ. of California, Irvine, Irvine, California, USA.

[56] Emmet James Whitehead, Jr. and Yaron Goland. 2004. The WebDAV Property Design. *Software, Practice and Experience* 34 (2004), 135–161.

[57] Wikipedia. 2017. Representational state transfer — Wikipedia, The Free Encyclopedia. (2017). https://en.wikipedia.org/wiki/Representational_state_transfer

[58] Scott Wolchok and J Alex Halderman. 2010. Crawling BitTorrent DHTs for Fun and Profit.. In *Fourth USENIX Workshop on Offensive Technologies (WOOT10)*. http://static.usenix.org/events/woot10/tech/full_papers/Wolchok.pdf

[59] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014). http://yellowpaper.io/