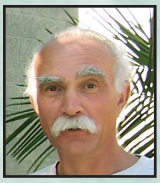
ISR Institute for Software Research
University of California, Irvine

The Infrastructure of a Computational Web



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

Justin R. Erenkrantz
justin@erenkrantz.com

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

May 2010

ISR Technical Report # UCI-ISR-10-3

Institute for Software Research
ICS2 221
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

The Infrastructure of a Computational Web

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

Justin R. Erenkrantz
justin@erenkrantz.com

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

ABSTRACT

We suspect that the diversity of web behaviors and applications are largely emergent, arising not from any single technical principle or mechanism, but instead resulting from the interaction of multiple primitive mechanisms. Hypothesizing that modifications of web primitives will yield entirely new web structures, behaviors, and applications we present the *computational web*, where computation subsumes content and computation exchange subsumes content exchange—a web described by the architectural style Computational REST (CREST). Based on trial implementations, we describe the infrastructure of a computational web comprised of CREST peers; offer the results of a novel test application, a dynamic, collaborative feed reader; and present an unexpected architectural structure, fractalware, an emergent structure of self-similar, recursive CREST peers.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Computational REST, CREST, REpresentational State Transfer, REST, computational web, computation exchange, distributed applications, decentralized applications

1. INTRODUCTION

No one, including its early developers, foresaw the astonishing diversity of web behaviors and applications now accepted as commonplace. We suspect that these behaviors and applications have been largely emergent, arising not from any single innate technical principle or mechanism, but as the outcome of the interaction of multiple primitive mechanisms. The modern web rests on a technical triad of URLs, metadata, and protocol. URLs name information resources; metadata is the means by which we distinguish among the multiple content representations of an information resource; and a single protocol, HTTP, defines the exchanges among web origin servers and clients. This triad, the core of REpresentational State Transfer (REST), largely accounts for both the web's success and its limitations [10, 11]. *We hypothesize that systematic modifications of these primitives will yield entirely new web structures, behaviors and applications.*

Starting with this hypothesis, we present a profoundly different future web—one dominated by computation and in which content is a *side effect* of computation. To this end we postulate a small set of enabling technical principles, listed in Table 1. In a computational web URLs name computation resources, not information; meta-programming and reflection replace metadata; and an asynchronous meta-protocol customized by policy, rather than a synchronous fixed protocol, defines the exchanges among computational web peers. This alternative triad—computation resources, meta-programming, and meta-protocol—is the core of Computational REST (CREST) [10, 11], an architectural style for which computation subsumes content and the exchange of computations subsumes the exchange of content—the *computational web*. If our hypothesis of emergent behavior is correct, then a computation-centric web will yield a broad spectrum of behaviors and applications not seen in a content-centric web.

Our prior work [11] detailed the motivation for, and the evolution of, CREST as an architectural style but provided no technical foundations for a CREST infrastructure, no implementations, nor any experimental results. We address each of these here, describing the underpinnings of a CREST infrastructure, the characteristics of trial implementations of CREST peers, and the experimental results of constructing a test application.

In doing so we expose the first example of emergence in the computational web, *fractalware*, an architecture of self-similar, recursive CREST peers; an unexpected construction, as nothing in CREST [11] defines fractalware or even suggests that it exists. Further, the fluidity of service within a fractalware infrastructure (as every CREST computation is a service provider in miniature) belies the traditional notion of middleware altogether, since the notion of a layer in a peering collective is computation-centric and its fractal structure suggests that "middleware service" is wherever one happens to find it: layers are a matter of convenience, perspective, and scope, not static construction and restriction.

We begin our discussion of the infrastructure of the computational web and the emergence of fractalware by outlining the architecture of the CREST infrastructure from the perspective of CREST peers in Section 2. Section 3 turns to

Technical Report UCI-ISR-10-3, May 2010

Table 1: Comparative Elements of REST and CREST

Technical Element	Content-Centric Web (REST)	Computation-Centric Web (CREST)
URL	*Names content resource*	*Names computation resource*
Metadata	*Permits 1–n mapping of resource to representations*	*Meta-programming and computational reflection*
Representation	*Byte sequence + metadata*	*Computation as closure or continuation (virtual machine-specific encoding)*
Protocol	*HTTP/1.1*	*Asynchronous meta-protocol*
Members	*Clients and servers*	*Peers*
Interactions	*Servers and clients exchange content*	*Peers exchange computations and messages*
Service composition	*Large grain predefined by server*	*Arbitrary grain defined by visiting computation*
Security	*Policy enforcement on access using client address or password*	*Federated policy enforcement for access to data and selection of providers*

the foundations of CREST, describing in some detail the technical elements of peers and computation exchange. To illustrate the potential of a computational web we present in Section 4 some early experimental results, a collaborative feed reader. Section 5 offers preliminary observations on CREST practices based on our experience to date. Section 6 is forward looking and sketches the mechanisms for addressing security and trust within a CREST infrastructure. Section 7, organized in the style of an FAQ, presents related work. Finally, Section 8 reviews our results so far and outlines our near-term research goals.

2. ARCHITECTURE

CREST architectures are comprised of peers that play the roles of both client and server. We describe those roles here from an architectural perspective and reserve further technical details for Section 3. In particular, we omit the multilayered security mechanisms regulating the peers' actions, reserving that discussion for Section 6. This pedagogical artifice eases discussion of the technical foundations, avoids needless complications in the presentation of the demonstration application, and simplifies the assessment of our hypothesis.

CREST peers receive and transmit messages, closures, and continuations. A peer hosts a peer-specific URL-space just as a web origin server hosts an origin-specific URL-space. Just as a web request from a web browser is directed to a server-specific URL, each and every CREST message, closure, or continuation sent to a peer is directed to some peer-specific URL u. Just as the meaning or interpretation of a web browser request is URL-specific and defined by the origin server, the meaning or interpretation of a CREST message or computation is URL-specific and defined by the receiving peer. In other words, the interpretation of, and response to, a message m, closure λ, or continuation $\vec{\lambda}$ directed to a URL u of P is P- and u-specific.

In its role as server, a peer P responds to messages sent to P by other peers and offers computation resources for closures and continuations sent by other peers. There are two forms of computation resources, both named by URLs. The first form, an *execution locus*, is a resource to which peers may direct a closure or continuation for execution. Each such locus may be customized for a specific domain (say order fulfillment) and/or language (such as JavaScript, Python, or Scheme). A peer may present many such loci, each named by a distinct URL.

The second form of computation resource is a *subpeer*, an active thread executing a closure or continuation within the confines of an execution locus. The semantics and behavior of a subpeer Q are no different from that of the parent peer P, within whose execution locus Q resides. Q, like its parent P is named by an URL and, like its parent P may—modulo P-specific security and resource policy—exchange messages, closures, and continuations with peers, offer its own Q-specific execution loci, and consequently, support subpeers of its own.

In its role as client, a peer P transmits messages, closures, and continuations to other peers. A single message may yield zero, one, or many responses, each response a message, closure, or continuation. When P sends a closure λ or a continuation $\vec{\lambda}$ to a peer Q there are two possible responses: either P will receive a message containing a value, the outcome of executing λ ($\vec{\lambda}$) as a function within the context of a Q-specific execution locus, or P will receive a message containing a URL u naming a newborn subpeer, an active thread executing λ ($\vec{\lambda}$) within an execution locus under the authority of Q. The first response is a degenerate case of the second response. The symmetric exchange and evaluation of closures and continuations among peers is *computation exchange*.

The privileges granted to a visiting closure (continuation) are never greater than the privileges of the peer to which the closure (continuation) is directed. In particular, the privileges of a subpeer are granted by its parent peer, are never more expansive than the privileges of the parent, and may be far more restrictive. In addition, each peer—and the complete recursive hierarchy of subpeers beneath it—is backwards-compatible with the existing web as each root peer (a peer for which there is no parent; P, V, and W in Figure 1) supports an HTTP-compliant web server that maps legacy web requests to (sub)peer asynchronous messages and the (sub)peer asynchronous replies to legacy web responses. Thus, for a legacy web client (browser), interacting with any (sub)peer is indistinguishable from interacting with a legacy web server. The obverse also holds, as (sub)peers may themselves interact with legacy HTTP web servers in a manner indistinguishable from any legacy web

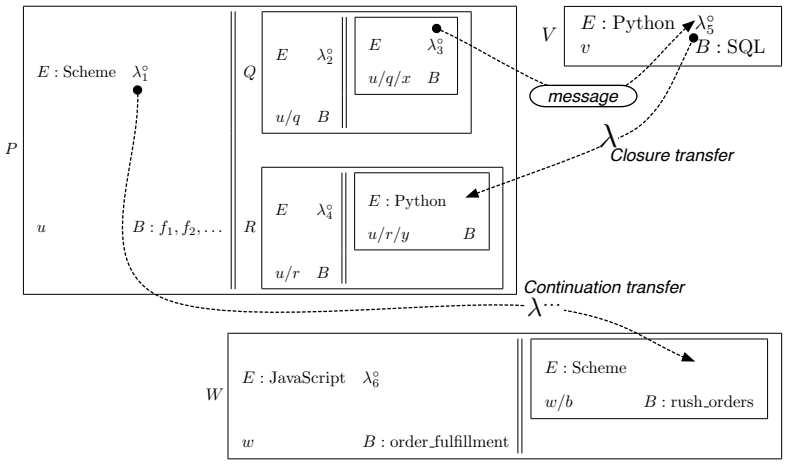


Figure 1: Interacting peers and subpeers.

client.

Computation exchange is deeply self-similar, exhibiting the same structure and semantics at many scales simultaneously. For example, under computation exchange a closure or continuation executing under the authority of a peer responds to messages using the same mechanisms as the peer itself, and service compositions of multiple peers are no different in structure or semantics than either the individual peers or the individual computation exchanges of which the whole is composed. We term this emergent property of the CREST architectural style *fractalware*.[1] As we shall see in Section 4 the same fractal element participates as a service component in many layers and many factorings simultaneously and simplifies simplifies forms of collaboration not commonly seen in a content-centric web.

3. TECHNICAL FOUNDATIONS

The technical elements of CREST draw from web architecture [6, 14, 39], formal models such as actors [4], pi-calculus [42], ambient calculus [7], and bigraphs [33], programming languages including Erlang [2], Scheme [9, 19] and JavaScript [31], protocols including BEEP [40], and HTTP [13] and internetwork architecture [8]. We present here details and rationale for peers, the CREST equivalent of web origin servers and clients; the naming and representation of computations; URL-specific binding environments, the mechanism by which peers specialize computation resources; the role of meta-programming as a substitute for metadata; and aspects of inter-peer messaging. Many of these elements play an important role in our test application described in Section 4, particularly, the allocation and exchange of service among multiple peers, URL-specific computational specialization, and agile inter-peer messaging.

3.1 Peers

A URL u of a peer P may name a resource $\langle B, E\rangle$ where B is a u-specific global binding environment and E a u-specific execution engine (see Figure 1). A resource of this form is an *execution locus*, diagrammed as $\boxed{\begin{array}{cc} E & \\ u & B \end{array}}$ where u is the URL of the locus, and B and E are its global binding environment and execution engine, respectively. A closure λ or continuation $\vec{\lambda}$ directed to URL u of peer P is executed under the authority of P in the context of global binding environment B by execution engine E (the representation of closures and continuations is sketched in Section 3.2).

A closure λ (continuation $\vec{\lambda}$) arriving at URL u of peer P comprising resource (execution locus) $\langle B, E\rangle$ may be *evaluated* or *spawned*. When *evaluated*, λ ($\vec{\lambda}$) is executed as a zero-argument function (a thunk) in a virgin thread T by engine E in the context of global binding environment B and the return value of the execution of λ ($\vec{\lambda}$) is transmitted to the peer (URL) named in the metadata accompanying λ ($\vec{\lambda}$). That return value may be a primitive (string, number, boolean, character), a computation (closure or continuation), a structure (list, vector, hash table, record), or an object. Naturally a computation, structure, or object may recursively contain any primitive, computation, structure, or object. In this case no other interaction is permitted [43].

When *spawned*, computation λ ($\vec{\lambda}$) is evaluated (as a thunk) in a virgin thread T by engine E in the context of global binding environment B. In this case, *unlike evaluation*, T is, by construction, a subpeer of P, named by a unique URL u_T and, like its parent, hosts a T-specific URL-space whose root is u_T. A subpeer is diagrammed as $\boxed{\begin{array}{cc} E & \lambda^\circ \\ u/\alpha & B \end{array}}$. As T is executing in the context of execution locus $\langle B, E\rangle$ under the authority of parent peer P, the capabilities of T are no more than those of P and may be significantly less. In general T may, to the extent permitted by the policy of parent peer P, receive and transmit messages, closures, and continuations and evaluate or spawn its own closures (continuations) or those sent to it by other peers.

Evaluation is merely a degenerate case of spawning and identical mechanisms protect the security and integrity of P in

[1]Thanks to John C. Knight, whose remark on the "fractal" structure of CREST peers inspired our nomenclature.

both cases. CREST peers, designed and constructed thusly are recursive, replicating, and self-similar at all levels. Nothing in the CREST architectural style dictates that peers be constructed so—it was revealed over the course of several generations of peer implementations. Fractalware is emergent, an unexpected consequence of the mechanisms of computation resources and computation exchange.

Peer P may confine a prospective T in any number of ways: limiting or modifying the contents of the URL-specific global binding environment B, restricting, modifying, or monitoring the actions of the URL-specific execution engine E, cordoning, throttling, or capping access to resources (for example, processor, memory, or bandwidth), or actively monitoring and restricting T's messaging or computation exchange with other peers. These restrictions are enforced recursively; it is impossible for any subpeer born under T to have any more rights than T itself. Thus T, under the guise of creating subpeers, can never expand its rights base and its parent P may, by recursive construction, restrict the rights of T and any of its descendant subpeers (including preventing T from creating any subpeers at all).

As shown in our test application (Section 4) subpeers are a powerful mechanism for service specialization, restriction, and replication.

3.2 Naming and Representing Computations

In the classic web, URLs name information (content) resources. In the world of CREST a URL names an active computation (a peer) or an execution locus offered by a peer. The computations exchanged among peers are represented as closures and continuations. Here a closure is a function f (whose representation may be source code, binary machine code, virtual byte codes, or some other intermediate form) plus the lexical scope binding environment of f (if any). A continuation is a snapshot of a (virtual) machine state including the stack, registers, and those portions of the heap (both data and instructions) recursively traceable from the stack or registers.[2] A machine may be physical hardware, a byte code interpreter, the interpreter of an abstract program representation, or a software emulation of physical hardware, to name only a few of the possibilities. As each execution locus $\langle B, E\rangle$ is both peer- and URL-specific, peers may offer multiple distinct global binding environments B and execution engines E to their fellow peers. For example, E could be a just-in-time compiler and virtual machine for JavaScript [18], a Java Virtual Machine [29], or a Scheme compiler/interpreter [37]. CREST is language-neutral and, to the extent that a language L permits closures and/or continuations, CREST peers may host L-specific execution loci. More generally, an execution locus may accept language L source code for compilation and execution on a peer-managed virtual machine E operating in a sandbox [22].

As URLs are both the names of active computations (peers and subpeers) and computation resources (execution loci), the same mechanisms, messages and computation exchanges, may be applied everywhere within a fractalware infrastructure.

[2]A serialized continuation (for computation exchange) can be voluminous however, there may be cases for which a partial (composable) continuation will suffice, with corresponding savings in volume.

3.3 URL-Specific Binding Environments

In functional programming languages the global binding environment defines those symbols not bound in lexical scope, for example, the symbol + in Scheme is bound to a base function that sums its parameters as numbers, (+ 3.14 7 22) returns 32.14. Scheme is exceptionally elegant in this regard, as all base language forms and primitives are defined as functional bindings in the global binding environment; however, the rough equivalent can be found in other languages, for example the modules of Python or the packages of Java. All of these languages and many others lend themselves to environment sculpting [31, 38] shaping the global binding environment by adding, modifying, or excluding bindings.

A binding environment B of $\langle B, E\rangle$ at URL u of peer P may *add* functions, say for image or audio transcoding, that are not defined in any other execution locus of P. Bindings in B may be *modified* to suit peer needs; for example, by restricting parameter types or implementing sophisticated per-function monitoring. Finally, *excluding* bindings from B enforces the principle of least privilege. For example, a peer prevents visiting computations from reading the local file system by expunging such functions from its URL-specific global binding environments B. Since the requisite functions are undefined, no visiting computation (and hence no subpeer) is granted the capability to read local files. Environment sculpting is illustrated in Figure 1, where execution loci at URLs v and w offer binding environments specialized to support SQL queries and order fulfillment respectively.

URL-specific binding environments generalize web services, free clients to compose novel services as they see fit, encourage service differentiation among providers, and ease service evolution. Fractalware allows every computation in a CREST infrastructure to be a service provider.

3.4 Meta-programming, Not Meta-data

Metadata in the content-centric web is the mechanism by which origin servers present one-to-many mappings of resources to representations and define the lifespan of those representations to web clients and intermediaries (caches and caching proxies). While metadata is explicitly available as a component of CREST asynchronous messaging in our implementation, it is secondary to the role of meta-programming and reflection within CREST. The global binding environment B of an execution locus $\langle B, E\rangle$ at a URL u may contain numerous reflective functions for enumerating the contents of the environment B, discovering function roles, types, and parameters, monitoring peer behavior, performance analysis, debugging, resource monitoring, and remote control, to name but a few.

Meta-programming is the means by which fractalware is reflective, whereby developers construct fractalware-based tools for the composition, orchestration, deployment, monitoring, analysis, and adaptation of fractalware-based applications. In other words, fractalware builds fractalware.

3.5 Protocols and Messaging

The serial in-order semantics of HTTP request/response reduces concurrency among and within web elements, increases delay and latency, and restricts the range of temporal behaviors among origin servers and web clients. Moreover, the imprecision of HTTP connection close [13] makes pipelining difficult—and transaction semantics impossible—to implement reliably. Transport parallelism between HTTP server and client requires multiple TCP connections, confounding congestion control, with each connection exacting the cost of TCP session startup and teardown.

To address these limitations our implementation of CREST messaging among peers is fully asynchronous and a message may contain any primitive, closure, continuation, structure or object and include arbitrary metadata (including closures or continuations) regarding the message as a whole or any message component. A peer, in reply to a message, may send zero, one, or many responses. While messages are delivered in order, a peer may reply to messages out of order, that is, the order of responses is independent of the order of message reception. Both classic request/response interactions and "pushed" notifications are degenerate cases of the full semantics.

Message interpretation is peer- and URL-specific and the same message m sent to URLs u and v may evoke two distinct responses, even if URLs u and v appear in the same peer P. The transmission of closures and continuations among peers is layered over a policy-driven asynchronous messaging meta-protocol [40] augmented with law-governed interaction [34]. Each connection is multiplexed into channels and each subpeer of a peer is assigned a distinct channel isolating the subpeer from any other subpeer sharing the connection while allowing message parallelism with full flow control. We intend that bandwidth throttling and law-governed interaction be performed per-channel, in other words, stateful peer-specific policy governs all subpeer messaging and computation exchanges.

Fractalware communication is fully asynchronous, allowing peers to adopt the temporal behavior best suited to their needs. In addition, fractalware adapts to legacy protocols, such as HTTP, as our experimental feed reader (Section 4 below) demonstrates.

4. FRACTALWARE AT WORK

Over the past three years we have constructed a series of exploratory CREST frameworks for experimenting with the structure and semantics of peers and constructing test applications. Their focus included URLs as names for computation, closure and continuation exchange among peers, environment sculpting, inter-peer messaging, and backwards compatibility with existing HTTP/1.1-compliant web servers and browsers.

The latest framework supports two classes of peers: exemplary peers and weak peers. Exemplary peers are standalone, support a rich set of computational services, and employ Scheme as the execution engine in their URL-specific computation resources. For interoperability with the legacy Web, an exemplary peer can act both as an HTTP/1.1 server (to expose the computations running on that peer to browsers) and as an HTTP/1.1 client (to allow computations executing on a peer to access any HTTP/1.1 server). On a modern Intel-class laptop with minimal performance tuning, our exemplary peers serve dynamic computations in excess of 200 requests per second. By contrast, weak peers execute within the confines of a Web browser and rely upon JavaScript as the execution engine. Mobile devices such as Apple iPhones and Google Android phones are supported as weak peers via their built-in browsers.

4.1 Feed Reader

Our example application, built upon our current CREST framework, is a highly dynamic, reconfigurable, collaborative reader for RSS or Atom feeds. For purposes of comparison, Google Reader[3] or Bloglines[4] are useful reference points.

However, our application demonstrates the power and utility of computation exchange as a mechanism for dynamic application (re)composition and real-time ad-hoc collaboration among multiple parties. It also highlights deep backwards interoperability with legacy web infrastructure.

In our example, two separate classes of cooperative computations are executing: widget computations (stateful producers) on the exemplary peers, and artist computations (stateless renderers) on the weak peers. There are eight different widgets: a manager (which allows a user, via a weak peer (browser), to create and link widgets), a URL selector, an RSS reader, tag clouds, sparklines, a calendar, a Google News reader, and a QR code. With a manager widget, these various widgets can be linked together; for example, linking the URL selector to the RSS reader to force the reader to fetch its feed from the URL given in the selector. Each widget computation is "tracked" by a corresponding (but independent) artist computation (executing in a weak peer) that visually represents the widget's state. A screen snapshot of the feed reader, from the perspective of a browser (weak peer) is shown in Figure 2.

With computation exchange all weak peers stay synchronized, where changes in the widget set, links between widgets, or widget positions made by one weak peer are reflected to all participating peers, both exemplary and weak. Here we have the computation equivalent of "shallow copy" where all peers participate in a single shared computation. Alternatively it is possible, using continuation exchange, to fork the computation at any point (the computation equivalent of "deep copy") in which one or more peers synchronize to a deep copy (the continuation) and share and manipulate a now *independent computation* whose state will, from that point forward, diverge along a separate path.

Using our CREST framework, the eight different widgets total less than 450 lines of Scheme code; the largest widget is the feed reader widget computation, approximately 115 lines of code. The artists, written on top of the Dojo JavaScript framework, comprise approximately 1,000 lines of JavaScript and HTML.

4.2 Observations and Lessons Learned

[3] `http://reader.google.com`
[4] `http://www.bloglines.com`

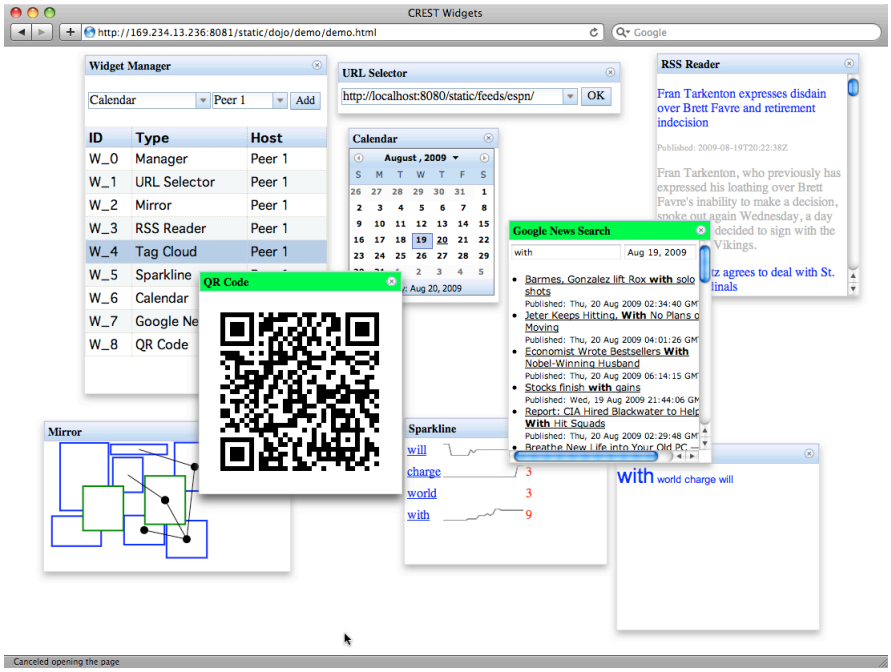


Figure 2: The feed reader from the perspective of a browser (weak peer).

URLs name computations. Each instantiation of a widget computation w on an exemplary peer has a unique per-instance URL. Through this URL, artist computations, executing in the confines of a week peer, and other widget computations on the same or different peer may direct messages to w. For example, messages either fetch the current state of the w computation or update the state of the w computation. Within a weak peer, artist computations have unique identifiers within the local DOM tree of the browser. However, these weak peers are restricted to communicating only with exemplary peers and cannot expose services to, or directly interact with, other weak peers.

Peers are fractalware. The individual widget computations w are subpeers with the same range of behaviors as the parent peer, but each subpeer offers distinct and specific services; in other words, each subpeer w interprets the messages it receives in the context of its URL- and computation-specific state. The self-similarity at all scales simplifies replication, collaboration, coordination, state sharing and state splitting. Independence at all scales eases the introduction of services such as new widgets and artists, or maintaining, for the sake of backwards compatibility, multiple versions of the same widget (service)—forms of adaptation that are critical for pliant service architectures.

Fractalware evolves. Services vary over time, reflecting need and subpeer relationships. The feed reader above stands up (or destroys) subpeers on demand in real-time in response to client requests. Computations, data flows (from subpeer to subpeer and from subpeer to artist), and state relationships are established, synchronized, and torn down to offer a consistent computational view to all peers, exemplary or weak.

Fractalware conserves state parsimoniously. Widget computations (subpeers) maintain local (per-instance) state. The subpeer acting as the base feed reader retrieves and parses the selected feed periodically, storing the parse tree (an XML document) as local state (essentially caching the feed). The feed reader (subpeer) easily scales since it returns, on request, only the cached representation rather than repeatedly retrieving and parsing the feed in response to each request and may be replicated courtesy of computation exchange. In contrast, the artists are stateless and periodically poll their respective widget computations for their state and render that state accordingly (HTTP prevents an exemplary peer from pushing state deltas directly to a browser, a weak peer). Fractalware is pliant with respect to state representation, preservation, and transfer.

Fractalware eases service and computation composition. New renderings of widget computations are trivial to introduce as artist and widget computations are loosely coupled by URLs and widgets (subpeers) do not discriminate among service (state) requests. As widget computations are themselves linked on-the-fly one to the other by URLs, dynamic service composition is trivial. For example, the Google News widget can be linked on demand to the calendar and tag cloud widget computations to search for a given keyword (from the tag cloud) at a given date in the past (from the calendar).

Fractalware promotes computational transparency. Fractalware supports a spectrum of computation exchange, from "shallow" copy to "deep" copy and all increments in between. For this example, in a shallow copy, all weak peers share among themselves a single set of widget computations (perhaps distributed among multiple exemplary peers) but have independent artist computations. In this case, all weak peers

see exactly the same state updates for the same widget computations. Using deep copy, a new weak peer would create a fresh, forked collection of widget computations (via computation exchange) whose computational states are a continuation, captured at the instant of the join, of the parent collection. This weak peer would now observe, from that point forward, an independent, evolving state. In essence the deep copy instantiates a new version of the feed reader application at the time the continuation is created. Further, there are intermediate points between shallow and deep copy where only a subset of the widget computations are forked, yielding semi-independent evolving states among multiple distinct instantiations of the application. This suggests new forms of collaboration where the degree of sharing is modulated by growing and pruning the fractalware on demand.

Fractalware promotes migration. As all exchanges among computations are identified by URLs, these computations may be either remote or local. In this way, widget computations can be migrated with abandon so long as they are accessible via an URL. By adding multiple exemplary peers, dynamic widget migration and load sharing (computational exchange) allows the sample application to scale seamlessly.

Fractalware eases latency. In the example application the division between the artist computation (which draws the local display) and the corresponding widget computation (which maintains the state) minimizes state transfer among exemplary and weak peers. When a widget computation (subpeer) is instantiated, the relevant artist computation is transmitted to and instantiated on the weak peer (computation exchange). As the artist computation is an output of the widget computation (per our motto "exchange computation, not content"), the widget computation exerts complete control over the state representations presented to the artist computation using representations optimized for the task at hand.

5. CREST BENEFITS

We explore here some of the consequences of the CREST architecture and technical foundations, drawing from our experiences as implementors of web services and web clients and the lessons of our analyses of prior systems [11].

5.1 Names

An URL u is, in essence, a capability, as a peer knowing u may transmit a message, closure, or continuation to it. For this reason, a URL u naming a subpeer contains a UUID [27] in its path to prevent peers from "guessing" u. URLs naming execution loci may be similarly protected and the binding environments of execution loci may contain functions that map "well known" paths to cryptographically encoded URLs. Thus URLs may be used as a form of encapsulation, which improves service modularity, and permits "private" services to offered alongside of public services.

Other useful information, protected by cryptographic signature(s), may be included in the path of a URL; for example, expiration dates to limit the period of access, declarations of connection rights, or other terms of use. In addition, messages, closures, and continuations may be embedded directly within a URL (we omit the details of the serialization and encoding) and transferred to another peer or archived for later use. This mechanism allows legacy web clients (such as browsers) to interact, with minor restrictions, with CREST peers and their execution loci using nothing but HTTP [10].

URLs play many roles in fractalware: as destination addresses for messages, closures, and continuations; as capabilities; as shrouds for encapsulation, as representations for computations; and as a mechanism for deep, backwards compatability with legacy HTTP web infrastructure.

5.2 Services

A single service may be exposed through multiple execution loci, each offering distinct perspectives on the same computation where the binding environments differ or support complementary supervisory functions for debugging or management. For example, a provider could offer tiers of execution loci to a single service based on a subscription fee, with higher fees commanding better performance or richer, more capable binding environments. Also, an alternative URL can name an execution locus for supervisory tasks tied to a particular service. For long-running custom computations, as is the intention for subpeers, an outside party may require information on the progress and state of the computation or wish to suspend or cancel the computation. In this case, a unique "supervisor" execution locus, named by a URL v can be generated by the parent peer in addition to the URL u naming the subpeer. Clients can then direct custom closures to v to access v-specific debugging, introspection, and control functions.

Fractalware allows services to vary among multiple dimensions such as functionality, performance, resources, control, feedback, or cost while still maintaining a high degree of commonality and uniform accessibility and interaction.

5.3 Time-Varying Behavior

The nature and specifics of an execution locus $\langle B, E\rangle$ may vary over time as functions may be added to, or removed from, binding environment B, or their semantics change. For example, a service provider, optimizing the cost of a service depending upon the present computational load, can offer a more precise version of a function that uses more CPU time during off-peak hours. Additionally, the execution engine E may change as well for the sake of bug fixes, performance enhancements, or security-specific improvements.

Time is a natural dimension of fractalware and temporal variations are accommodated by meta-programming and reflection.

5.4 State

Many distinct computations (subpeers) may be underway simultaneously within the confines of the same resource (execution locus). A single client may issue multiple evaluations or spawns to the same URL u and many distinct clients may do the same simultaneously. An execution locus may choose to be stateful, thus allowing indirect interactions between different computations, or stateless, where parallel computations (subpeers) have no influence or effect on any other instance within the same (or any other) execution locus. With stateless execution loci, independent parallel computation is straightforward. Scalability for stateful services (such as a

data base) demands consistency mechanisms for safety and to regain some measure of parallelism.

Fractalware offers a high degree of parallelism but also supports degrees of state coupling for synchronization and coordination ranging from state (memory) sharing within a single execution locus to state exchange by way of messaging.

5.5 Computation

REST relies upon an end-user's navigation of the hypermedia (through the links between documents) to maintain and drive the state of the overall application [39]. In contrast, CREST relies upon potentially autonomous computations to exchange and maintain state. Given fractalware, we expect that it will be common for one resource to refer to another either directly or indirectly; hence, there may be a rich set of stateful relationships among a set of distinct URLs. This state can be captured within the continuations exchanged between peers; in particular, a service provider could supply its consumers with a continuation that permits later resumption of the service. In this way, the provider does not have to remember anything about the consumer as all of the necessary information is embedded within the continuation and the consumer merely retains the continuation to let the provider resume the state of the service at a later point in time.

Computation exchange is a promising starting point for services devoted to restart, backup, replication, and transfer of function in decentralized systems.

5.6 Migration and Latency

Fractalware-based applications can minimize network and computational latency by migrating computations. Applications may stitch many computations from multiple service providers into a comprehensive whole, as no one peer may be capable of supplying all of the functional capability that a client requires. As overall performance may be subject to, and dominated by, variations in network latency, fractalware encourages computation migration, that is, moving the computation closer to the data source, to reduce latency. Its efficacy is this regard deserves further investigation.

CREST communications are fully asynchronous, though an exemplary peer respects the request/response ordering constraints when communicating with an HTTP-compliant weak peer. CREST peers, both exemplary and weak, must offer mechanisms for reducing or hiding the impact of latency; for example, by encouraging concurrent computation and event-driven reactions (such as the nondeterministic arrival of responses). Since those responses may be continuations, origin peers must be receptive to previously generated continuations long after they were first created (on timespans of seconds to months) and restart them seamlessly. CREST peers employ timers to bound waiting for a response or the completion of a computation and execution loci may employ timestamps and cryptographic signatures embedded within a continuation to reliably determine its age and may refuse to resume a continuation past its "expiration date."

Computation exchange lowers the barrier to migrating computations closer to their data sources. Mechanisms that reduce "friction" are often disruptive, forcing unexpected realignments. We speculate that migration, available as a fundamental mechanism, may lead to novel services and applications.

6. SECURITY AND TRUST

Security and trust [44] are critical to collaborative applications as all are at risk if the CREST infrastructure is insecure. We discuss here the provision of security at this foundational level. The discussion is largely speculative, as many of the mechanisms described are either unimplemented or the subject of ongoing experimentation.

6.1 Mobile Code

Computation exchange relies on the technology of mobile code. CREST is currently implemented as a Scheme interpreter [32] executing within a Java Virtual Machine (JVM). Computations are exchanged among CREST peers as closures or continuations—expressed in the high-level virtual machine code of the Scheme interpreter. This layered implementation provides considerable defense against malicious or erroneous mobile code, as the JVM executes within a host sandbox (such as a FreeBSD jail, Solaris Container, or a host-level hypervisor [35]), the Scheme interpreter executes within a JVM sandbox [30], and the Scheme interpreter itself is protected by CREST laws for verification of the (virtual) machine code delivered by arriving computations. No execution locus or subpeer has access to global state or any other execution locus.

Once a peer dispatches a computation to perform, CREST interactions use self-certifying URLs [25] to ensure mutual strong authentication of each execution locus. This authentication allows a CREST peer to offer tailored service to specific peers and deny that service to unapproved peers. Bytecode verifiers [28] can ensure that a computation is safe to execute. Peers offering computations can use environment sculpting [38] (a form of the principle of least privilege) to add, remove, and wrap functions in each URL-specific global binding environment to ensure minimal privileges and resource sandboxing [36, 41] to restrict and monitor resource consumption for each visiting peer.

6.2 Computation Placement

Under traditional authorization the service provider determines whether a request is authorized before accepting the request. Computational placement is the obverse—which peers are authorized or accredited to provide the services demanded by the calling party. There will be policy and authorization computations done on both sides—by the server evaluating the authorizations of the client—as well as client selection of the destination "vendor" execution locus for the computation exchange.

The trust properties required for distributed, decentralized applications vary from case to case; likely candidates include security, reliability, dependability, and fault-tolerance. The CREST defense-in-depth, plus law-based interactions, both within the execution context of a locus and among computations interacting at multiple loci, offer a promising foundation for secure and trustworthy interactions among elements at all levels.

To address issues in selecting the services to which a computation might migrate, we will utilize accreditation attributes associated with server environments [20]. Computations will be tagged with security and reliability requirements, which will be used as inputs to the process of selecting the computation environments to which sub-computations may migrate. Moreover, considerations of privacy and side-channel obfuscation can be included in these requirements. The process of dispatching a computation will retrieve certified (accredited) attribute certificates from the target peer, and validate requirements before dispatching the computation. Similar techniques can also be applied to provide type enforcement for cross-platform computation [3].

6.3 Federated Security Services

Administration of computations in the CREST architecture is federated; a computation exchange may cross agency borders and hence, distinct policy regimes. With a reliance on core security services such as identity management and authorization, security assertions from one part of the system must be used and understood everywhere. Some policy decisions in the CREST system will be based on identity (user, peer, or execution locus). We propose to adapt existing federated identity management technologies, such as Shibboleth[5], and federated ID notarization [21] to tie existing user credentials to our framework.

Additionally, the execution behavior of visitors will be actively managed by the URL-specific law [1], itself a CREST computation that oversees and regulates each and every significant action of the visitor including, but not limited to, virtual machine code verification, identity authentication, rights restriction and control, resource throttling and bounding, communications (such as message contents, format, destination, or point of origin), lifespan, and auditing. Since the enforcer of the URL-specific law maintains locus-specific state and has read access to the global state of all loci executing under the color of a specific authority (agency), it can enforce the law in a manner that is both context-sensitive and responsive to evolving conditions [1, 34]. Moreover, the base mechanisms for identity and those for establishing and enforcing CREST laws provide a sound foundation for higher-order policy and extensions.

6.4 CREST Contributions to Security

CREST computation exchanges may augment the policy enforcement mechanisms deployed in traditional systems. Closures supply a computation context similar to the security context often used to make access decisions; computation exchange appears better aligned with the desired flow of credentials than is provided through the security context management mechanisms in traditional systems.[6]

7. RELATED WORK

CREST draws upon a broad range of related work. Space constraints prohibit a comprehensive discussion; however, several questions arise repeatedly. To help place CREST in context we present related work as "frequently asked questions."

Isn't CREST Computational Exchange Just Mobile Code? While CREST shares many mechanisms with mobile code [16, 17, 23] it is fundamentally different in several respects. Each CREST resource is either an execution locus, a binding environment plus an execution engine, or a subpeer (an execution engine executing a closure [5, 12] or a continuation [24] transmitted to it by a CREST peer), both named by URLs. CREST peers may communicate via messages [2, 19] with any computation for which they hold an identifying URL. CREST unifies code mobility, distributed computation, and fine-grain, Internet-scale, interthread messaging with a single uniform, scalable naming scheme. Since each URL-named execution locus contains a URL-specific global namespace and execution engine, CREST resources may vary from one another by varying the execution locus affiliated with the URL. Again, a single mechanism allows a CREST peer to offer an array of services and service mixes, each organized in a tree of peer-specific URLs. This degree of flexibility and specificity allows CREST peers, via computation exchange from one URL-specific execution locus to another, to shape, construct, modify, and extend on-the-fly service offerings not only from peer to peer but from URL to URL.

Isn't CREST the Same as Google Wave? Google Wave [26] is a document-centric service that relies on dedicated servers and a class of sophisticated distributed algorithms, Operational Transforms, to maintain a consistent document view for a set of shared documents that are edited simultaneously by multiple participants. Google Wave does not implement computation exchange; its focus is restricted to multiple-user, shared document applications.

Hasn't the Cloud Already Solved This Problem? The cloud [8], and prior to it, the grid [15], are both major advances in harnessing remote computing resources to solve large (computation or data volume) problems. Our work is not restricted to such problems, although it would address them by integrating with existing cloud/grid infrastructures. Neither the cloud nor the grid are web architectures at all—there the web serves as little more than a convenient transport medium. In contrast, the CREST infrastructure presented here refashions web architecture as fractalware and offers more flexibility and finer-grain computation composition than either the cloud or the grid.

8. SUMMARY

The foundational elements of CREST, as applied here, give rise to a distinctly different web—a computational web—one whose differences suggest that web behavior is emergent, arising from variations within a confined design space. Our experiments to date, including the implementation of CREST peers and a test application, exhibit behaviors not found in the HTTP-based content-centric web. In addition, CREST peers themselves exhibit a novel architectural structure, fractalware, a recursive construction that offers attractive, but still unproven, formulations for collaboration, state sharing, and computational composition.

In the future we plan to pursue two parallel, but complementary, research paths. We will expand the set of test applications for fractalware and are considering several domains, including e-commerce, energy management, grid computing

[5] `http://shibboleth.internet2.edu`

[6] Thanks to Clifford Neuman for this observation.

data services, and multiplayer games. At the same time we will invest considerable effort in refining and extending the next-generation CREST peers, with particular emphasis on policy-based computation exchange and security suitable for large-scale, distributed, decentralized services. We anticipate that the applications will greatly inform the structure and semantics of the peers and that the peers, with their fractalware constructions, will suggest novel architectures in our target domains.

9. ACKNOWLEDGMENTS

Our thanks to Alegria Baquero and Yongjie Zheng for their assistance in implementing the feed reader application. We are indebted to Alegria Baquero and Kyle Strasser for their painstaking reviews of prior drafts of this paper. We also thank André van der Hoek, James Jones, and Neno Medvidovic for their comments on portions of this paper.

This work supported in part by the National Science Foundation under Grants CCF-0438996 and CCF-0820222.

10. REFERENCES

[1] X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *Proceedings of the European Symposium on Research in Computer Security*, pages 39–60, 2003.

[2] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[3] L. Badger, D. F. Sterne, D. L. Sherman, et al. Practical domain and type enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1995.

[4] H. Baker and C. Hewitt. Laws for communicating parallel processes. Working Papers WB-134A, MIT Artificial Intelligence Laboratory, May 10, 1977.

[5] A. Bawden and J. Rees. Syntactic closures. In *Proceedings of the ACM conference on LISP and functional programming*, pages 86–95, 1988.

[6] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). RFC 1738, December 1994.

[7] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science, Special Issue on Coordination*, 240(1):177–213, June 2000.

[8] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Pearson Education Inc., 2007.

[9] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, 4th edition, 2009.

[10] J. R. Erenkrantz. *Computational REST: A New Model for Decentralized, Internet-Scale Applications*. PhD thesis, University of California, Irvine, September 2009.

[11] J. R. Erenkrantz, M. M. Gorlick, G. Suryanarayana, and R. N. Taylor. From representations to computations: The evolution of web architectures. In *Symposium on the Foundations of Software Engineering*, pages 255–264, September 2007.

[12] M. Feeley and G. Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987.

[13] R. T. Fielding et al. Hypertext transfer protocol HTTP/1.1. RFC 2616, June 1999.

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

[15] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

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

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

[18] A. Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of Conference on Programming Languages, Design and Implementation*, June 2009.

[19] G. Germain, M. Feeley, and S. Monnier. Concurrency oriented programming in Termite Scheme. In *Proceedings of Scheme and Functional Programming Workshop*, pages 125–136, September 2006.

[20] M. T. Goodrich et al. Authenticated dictionaries for fresh attribute credentials. In *Proceedings of the Trust Management Conference*, pages 332–347, 2003.

[21] M. T. Goodrich, R. Tamassia, and D. D. Yao. Notarized federated id management and authentication. *J. Comput. Secur.*, 16(4):399–418, 2008.

[22] Google: The Chromium Projects. The Chromium Projects: Sandbox. `http://dev.chromium.org/developers/design-documents/sandbox`, March 2010.

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

[24] R. Hieb and R. K. Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, 1990.

[25] M. Kaminsky and E. Banks. SFS-HTTP: Securing the web with self-certifying URLs. Technical report, MIT Laboratory for Computer Science, 1999.

[26] S. Lassen and S. Thorogood. Google Wave Federation Architecture. `http://www.waveprotocol.org/whitepapers/google-wave-architecture`, 2009.

[27] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (UUID) URN namespace. RFC 4122, July 2005.

[28] X. Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, April 2003.

[29] T. Lindhold and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, 1999.

[30] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. Wiley, 2nd edition, 1999.

[31] M. S. Miller et al. Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`, June 2008.

[32] S. G. Miller. SISC: A complete Scheme interpreter in Java. `http://sisc.sourceforge.net/sisc.ps.gz`, 2002.

[33] R. Milner. *The Space and Motion of Communicating*

Agents. Cambridge University Press, 2009.

[34] N. H. Minsky and V. Ungureanu. Law-governed interaction: A coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering Methodology*, 9(3):273–305, July 2000.

[35] T. Mitchem, R. Lu, and R. O'Brien. Using kernel hypervisors to secure applications. In *Proceedings of the 13th Annual Computer Security Applications Conference*, 1997.

[36] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, 2002.

[37] A. Piérard and M. Feeley. Towards a portable and mobile Scheme interpreter. In *Proceedings of the Scheme and Functional Programming Workshop*, pages 59–68, September 2007.

[38] J. A. Rees. *A Security Kernel Based on the Lambda Calculus*. PhD thesis, Massachusetts Institute of Technology, 1996.

[39] B. L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2007.

[40] M. T. Rose. The blocks extensible exchange protocol core. RFC 3080, March 2001.

[41] A. D. Rubin and D. E. Geer, Jr. Mobile code security. *IEEE Internet Computing*, 2(6):30–34, 1998.

[42] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[43] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–564, 1990.

[44] G. Suryanarayana et al. An architectural approach for decentralized trust management. *IEEE Internet Computing*, 9(6):16–23, November/December 2005.