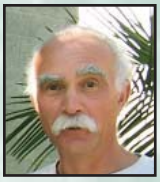
ISR
Institute for Software Research
University of California, Irvine

CREST: A new model for Decentralized, Internet-Scale Applications

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



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

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

September 2009

ISR Technical Report # UCI-ISR-09-4

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

CREST: A new model for Decentralized, Internet-Scale Applications

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

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
jerenkra@ics.uci.edu, mgorlick@acm.org, taylor@ics.uci.edu

ABSTRACT

CREST is a new architectural style for highly dynamic distributed applications. CREST (Computational REST) is a generalization of the REST architectural style that shaped the scaleable WWW. In CREST, URLs denote loci of computations and the representations exchanged are expressions that may be as simple as a string literal or as rich as full continuations or closures. CREST eliminates the client-server distinction of the WWW in favor of an economy of dynamic, individualistic peers. Hence CREST supports a computational exchange web where delivered content is a "side-effect" of such exchange. CREST emerged from long-term study of Web applications, examining the ways in which dynamism has started to appear, and the ways in which developers have struggled to apply REST principles in increasingly demanding applications. The paper presents the five key CREST axioms and discusses the issues that application designers must address. A framework supporting implementation of CREST applications is described and illustrated through discussion of a demo application, a dynamic news feed processor. The framework is fully backwards compatible with the existing Web infrastructure.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Representational State Transfer, mobile code, web services

1. INTRODUCTION

In our prior work [7], we highlighted several example web applications to illustrate a dramatic shift in emphasis and approach---the emerging dominance of computation over content. More specifically, that transition can be parsed with respect to seven distinct and enduring categories: *names*, *services*, *time*, *state*, *computation*, *transparency*, and *latency*. For example, our experiences with mod_mbox demonstrated that *names*, here URLs denoting resources, may represent more both static content and dynamic representations generated on-the-fly to suit a particular domain (in this case, web-based access to very large email archives). Our experience in constructing web-based access for Subversion illustrated that decoupling and segmenting *state* (by strictly separating communication and representation transformations internally within the architecture of a user-agent) can minimize *latency*, reduce network traffic and increase the efficacy of caching (in this case, far fewer HTTP requests were required for the same operations). Here we deployed protocol-level optimizations that both respected the standard behaviors of HTTP and did not frustrate the expectations of intermediaries (such as caching proxies).

Continuing the examples, fine-grained internet-scale *services* are a noble goal as they promote composition and innovation by both vendors and consumers. However, SOAP-based Web Services, crippled by well-known implementation deficiencies (such as a reliance on RPC, a lack of idempotency, and the improper intermingling of metadata and data in SOAP messages), are fundamentally incapable of realizing the promise of fine-grained, composable services without egregiously violating the REST axioms [8] that permitted the web to scale. SOAP-based Web Services, if broadly adopted, would rollback the scalability enabled by HTTP/1.1, itself designed explicitly with REST in mind. On the other hand, RESTful Web Services, for lack of crisp and clear design guidance, have yet to reach their full potential. In particular, services whose semantics are not closely tied to content delivery (such as shared document editing) offer maddeningly inconsistent and incomplete interfaces.

AJAX and mashups, two comparatively recent web innovations, illustrate the power of *computation* in the guise of mobile code-on-demand, as a mechanism for framing responses as interactive computations (AJAX) or for "synthetic redirection" and service composition (mashups). No longer must static content be transported from an origin server to a user-agent---we now transfer incomplete representations accompanied by domain-specific computations applied client-side to reify the content on-the-fly. The modern browser is now a capable execution environment where XML-based documents combined with interpreted JavaScript produce sophisticated client-side applications which are low latency, visually rich, and highly interactive. Additionally, AJAX-based mashups play the role of computational intermediary (proxy) in an AJAX-centric environment.

Drawing from the observations summarized above, we formu-

lated a small set of principles for Computational REST (CREST), a computation-centric successor to the REST architectural style, that characterized our vision of a future computational web. Over the past two years, we have refined those principles and constructed a series of prototypes of a reference infrastructure. We recently stood up the first complete CREST peers and to better understand the design and engineering consequences of the CREST architectural style, implemented a demonstration application that illustrates the surprising power, scope and flexibility of a computational web. In the following sections, we set out the technical underpinnings of our vision, describe in detail the principles underlying the design and implementation of the CREST peers, examine the issues that applications must address to exploit the CREST infrastructure to good advantage and illustrate these points with a discussion of our demonstration applications.

We proceed by first reviewing the basic concepts of continuations and closures, then proceed to discuss the broad idea of computational exchange and mobile code. This is followed in Section 4 with a description of how these concepts are specifically used in CREST. Details of CREST, including its axioms and design considerations, follow in subsequent sections.

2. CONTINUATIONS AND CLOSURES

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

In these programming languages which support continuations, closures are the typical mechanism by which continuations are defined. Figure 1 illustrates a closure in JavaScript. The function `closure_example` does not directly draw an alert box, but instead returns a function which draws an alert box. When the closure is evaluated, the user will see a pop-up box as depicted in Figure 2. More precisely, a closure is a function with zero or more free variables such that the extent of those variables is at least as long as the lifespan of the closure itself. If the scope of the free variables encompasses only the closure then those variables are private to the function (they can not be accessed elsewhere from other program code) and persist over multiple invocations of the function (a value established for a variable in one invocation is available in the next invocation). Consequently, closures retain state (thereby sacrificing referential transparency) and may be used to implement state encapsulation, representation, and manipulation–the necessary properties for computational exchange powered by continuations.

3. COMPUTATIONAL EXCHANGE

We borrow liberally from a rich body of prior work on mobile code and continuations to articulate our view of computational exchange. An excellent survey and taxonomy of mobile code systems may be found in [10] and, in particular, there are several

```
function closure_example(n) {
   var alert_text = 'Hello ' + n + '!';
   var alert_closure =
      function() { alert(alert_text); }
   return alert_closure;
}
var v = closure_example('John');
v();
```

Figure 1. An example of a closure in JavaScript

Figure 2. The resulting window produced by JavaScript closure in Figure 1

examples of mobile code implementations based on Scheme. Halls' `Tubes` explores the role of "higher-order state saving" (that is, continuations) in distributed systems [12]. Using Scheme as the base language for mobile code, Tubes provides a small set of primitive operations for transmitting and receiving mobile code among Tubes sites. Tubes automatically rewrites Scheme programs in continuation-passing style to produce an implementation-independent representation of continuations acceptable to any Scheme interpreter or compiler. Halls demonstrates the utility of continuations in implementing mobile distributed objects, stateless servers, active web content, event-driven location awareness, and location-aware multimedia.

Mobile objects are a weaker form of computation mobility. Scheme has been used to support mobile objects with `Dreme` in pursuit of distributed applications with little concern for process or network boundaries [9]. There, extensions to Scheme include object mobility, network-addressable objects, object mutability, network-wide garbage collection, and concurrency.

Continuations have an important role to play in many forms of web interactions and services. For example, Queinnec demonstrates that server-side continuations are an elegant mechanism to capture and transparently restart the state of ongoing evolving web interactions [21].

Matthews et al. extend this work, offering a set of automated transformations based on continuation-passing style, lambda lifting, and defunctionalization that serialize the server-side continuation and embed it in the web page returned to the client [16]. When the client responds to the web page the (serialized) continuation is returned to the server with the server resuming execution of the continuation. This is an example of computational exchange (from server to client and back again) that preserves context-free interaction and allows the server to scale by remaining largely stateless.

4. A COMPUTATIONAL EXCHANGE WEB

Scheme is the language of choice for CREST, in particular, Scheme is the language of computational exchange and Scheme

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

Figure 3. Example CREST program (Scheme)

```
{
   if (wordcount) {
      return wordcount(GET(
         "http://www.example.com/"));
   }
}
```

Figure 4. Example CREST program (JavaScript)

expressions, closures, continuations, and binding environments are both the requests and responses exchanged over the web. (Subsequently, we will also use simpler expressions in CREST exchanges; this is omitted from discussion here for parsimony.) Raising mobile code to the level of a primitive among web peers and embracing continuations as a principal mechanism of state exchange permits a fresh and novel restatement of all forms of web services, including serving traditional web content, and suggests the construction of new services for which no web equivalent now exists.

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

With CREST, there are two fundamental mechanisms of operation: `remote` and `spawn`. A `remote` computation evaluates the given program, closure, or continuation and if there is a result returns the resulting expression (which could be a new program, closure, or continuation) back to the original requestor. The other mechanism of operation is `spawn`, which is intended for installing longer-running custom computations. This mechanism allows a peer to install a new service and receive a new URL in response which permits communication with the newly installed service. This new URL can then be shared and messages can be delivered to the new service with the response wholly under the control of the newly installed computation.

To help illustrate the semantics of `remote`, we provide an example program (expression; rendered in the concrete syntax of Scheme in Figure 3 and JavaScript in Figure 4) issued by a client *C* to an URL *u* of origin server *S*. This program tests the execution environment of *S* for a function `word-count` (service discovery) and if the function (service) is available, fetches the HTML representation of the home page of `www.example.com`, counts the number of words in that representation (service composition), and returns that value to *C*. As shown by this example, in a web of computational exchange, the role of SOAP can be reduced to a triviality, service discovery is a natural side-effect of execution, and service composition reduces to program (expression) composition.

```
(define (word-count-service)
 (accept
  ((reply message-id ('GET url))
   (where (or (symbol? url) (string? url)))
   (! reply message-id (word-count (GET url)))
   (word-count-service))

  (_ (word-count-service)))) ; Ignore other message forms.
```

Figure 5. Example CREST spawn service (Scheme)

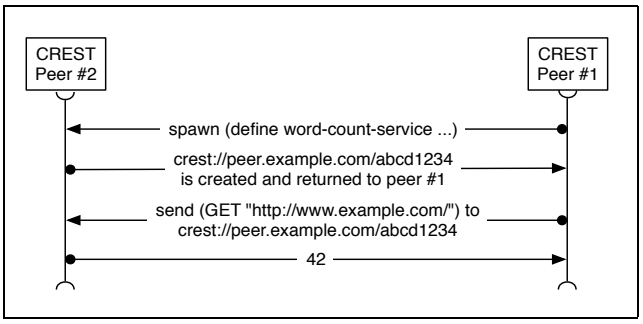


Figure 6. Messages exchanged in CREST

An example of a `spawn`-centric word-count service is provided in Figure 5. In this example, the overall computational view remains the same, but with a crucial distinction–a new resource (represented by an URL) now exists, where no such resource existed before, that responds to word-count requests. In other words, CREST computations synthesize new services out of old by exposing URLs as computational resources rather than content resources. An example of the sequence of computations and messages that are exchanged to install and use this service are presented in Figure 6.

5. CREST AXIOMS

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

CA1. A resource is a locus of computations, named by an URL.
CA2. The representation of a computation is an expression plus metadata to describe the expression.
CA3. All computations are context-free.
CA4. Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.
CA5. The presence of intermediaries is promoted.

5.1 A resource is a locus of computations, named by an URL. (CA1)

Any computation that can be named can be a resource: word processing or image manipulation, a temporal service (e.g., "the predicted weather in Cape Town over the next four days"), a

generated collection of other resources, a simulation of an object, and so on. Compared with REST, this axiom is not entirely inconsistent with the original REST axioms - many REST resources are indeed computation-centric (especially those presented under the guise of "RESTful Web Services"). However, CREST, by explicitly emphasizing computation over information, makes it far clearer that these are active resources intended to be discoverable and composable.

As discussed earlier, when a computational request arrives at an URL, two different modes of operation are possible: `remote` or `spawn`. If the closure is a `remote`, once the closure is received, the computation is realized within the context and configuration of the specific URL. In this way, the URL merely denotes "the place" where the received computation will be evaluated. Until the closure is received, the specified URL can be viewed as quiescent as there is no computation running but the URL merely presents the possibility of computation. If the evaluation of the received closure is successful, then any value (if there is one) returned by the closure on its termination will be sent back to the original requestor. In this way, `remote` is the equivalent in CREST of HTTP/1.1's GET request. With a `remote` closure, there is no provision for communicating with the closure after it is exchanged. No outside party, including the client who originally submitted the `remote` request, can communicate in any way with the `remote` computation once evaluation begins.

In contrast, if the closure received indicates a `spawn`, the locus of computation is revealed in a limited way in that a unique URL is then *created* to serve as a mailbox for that `spawn`ed computation. For example, Figure 7 denotes one potential formulation for a mailbox URL whereby the hex string represents a universally unique identifier. In response to the `spawn` request, the requestor is returned this URL immediately. This particular URL may be used by the original requestor or provided to another node for its own use. With that mailbox URL in hand, any client may now send arbitrary messages to the `spawn`ed closure via the mailbox. The executing closure which was provided with the initial `spawn` will read, interpret, and potentially respond to those messages.

5.2 The representation of a computation is an expression plus metadata to describe the expression. (CA2)

Since the focus of CREST is computational exchange, it is only natural that the representations exchanged among nodes are amenable to evaluation. In this axiom, we follow Abelson and Sussman's definition of expression: "primitive expressions, which represent the simplest entities the [programming] language is concerned with" [1]. While these exchanges may be simplistic in form (such as a literal representing static content or binary data), we expect that more complex expressions will be exchanged - such as closures, continuations, and binding environments. As discussed earlier, closures and continuations are particularly well-suited for computational exchange as they are powerful programming language constructions for state encapsulation and transfer. The exchange of binding environments (which associate variable names with their functional definitions or values) is yet another mechanism for transferring complex compositional computations.

CREST can also leverage its distinctive view of computation in order to produce more precise and useful representations. REST is nominally silent on the forms of exchanged representations but, to drive the application, implicitly requires the exchanged representations to be a form of hypermedia. In contrast, CREST relies upon computational expressions and the exchange of such to drive the application. To produce the appropriate representation, CREST employs explicit, active computations where REST relies upon a repertoire of interpreted declarative forms, such as the declared MIME types enumerated by a User-Agent. CREST achieves far greater precision when negotiating a representation, for example, not only can a particular format be specified (such as JPEG) but also specific resolutions (thumbnails versus full images). This model of content negotiation can simply not be achieved in REST in a straightforward manner. Exploiting computation directly in the negotiations reduces its complexity and eliminates the complex parsing and decomposition of representations thereby improving encapsulation, isolation, and composibility.

5.3 All computations are context-free. (CA3)

Like REST, CREST applications are not without state, but the style requires that each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it. Prior representations can be used to help facilitate the transfer of state between computations; for example, a continuation (representation) provided earlier by a resource can be used to resume a computation at a later time merely by presenting that continuation. Again, REST has a similar restriction - however, the mechanism for interactions in a context-free manner was under-specified and offered little guidance for application developers. By utilizing computational semantics, continuations provide a straightforward and expansive mechanism for achieving context-free computations.

5.4 Only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged. (CA4)

In HTTP/1.1 - the best known protocol instantation of REST - the available operations (methods) of the protocol are documented in the relevant standards documentation (RFC 2616). With HTTP/1.1, the server may support additional methods that are not described in the standards, but there is no discovery mechanism available to interrogate the server about which methods are supported on a particular resource. In CREST, since the node offers base programming language semantics, such discovery mechanisms are intrinsic and always available.

CREST nodes may define additional operations at a resource-level - that is, there may be operations (functions or methods) defined locally by the server which are pre-installed and are optimized for the server's specific environment. While these operations are exposed via CREST's computational model, these operations may actually be implemented in languages that are not directly amenable to CREST's computational exchange model (such as C, Java, or even raw machine code). For example, these locally defined functions may be front-ends to a proprietary database of credit scores, airline routings, or storehouse inventories. These mechanisms allow a particular provider to expose optimized or value-added resources to other CREST peers.

A critical feature lacking in the HTTP/1.1 protocol is that of two-way extensibility at run-time - new methods (such as those supported by extensions like WebDAV) can only be implemented on

http://www.example.com/mailbox/60d19902-aac4-4840-aea2-d65ca975e564

Figure 7. Example SPAWN mailbox URL

the server, but, other than implicit agreement or trial and error (handling the complete absence of a particular method), dynamic protocol adapation is not feasible with HTTP/1.1. However, with a CREST-governed protocol - which relies upon providing a computational platform (in our examples, in the rendered form of Scheme) - protocol enhancements are merely a form of providing additional computations (such as new functions) on top of the existing computation foundation. Therefore, with CREST, if a specific method is not available, the participant can then submit the code to the resource which interprets that method exactly as the participant desires.

In other words, participant *A* can send a representation *p* to URL *u* hosted by participant *B* for interpretation. These *p* are interpreted in the context of operations defined by *u*'s specific binding environment or by new definitions provided by *A*. The outcome of the interpretation will be a new representation—be it a program, a continuation, or a binding environment (which itself may contain programs, continuations, or other binding environments). To reiterate, a common set of primitives (such as the base semantics of the computational substrate, such as the Scheme primitives) are expected to be exposed for all CREST resources, but each *u*'s binding environment may define additional resource-specific operations and these environments can be further altered dynamically.

5.5 The presence of intermediaries is promoted. (CA5)

Filtering or redirection intermediaries may exploit both the metadata and the computations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the end user-agent and the origin server. The clear precursor of the full power of this axiom in a computational exchange web is in AJAX-based mashups, more fully explained in our prior work [7]. A derivative of our earlier word count example is given in Figure 8. In this example, CREST peer #2 has configured itself to use `babelfish.example.com` to translate all outgoing requests into Portuguese. CREST peer #1's code has not changed, but CREST peer #2's new configuration alters the computation yielding a different result..

6. THE PRACTICE OF CREST: ISSUES AND RESOLUTIONS

As the REST experience demonstrates, it is insufficient to merely enumerate a set of architectural principles; concrete design guidance is required as well. To this end, we explore some of the consequences of the CREST axioms, cautioning that the discussion here is neither exhaustive nor definitive. Nonetheless, it draws heavily upon both our experiences as implementors of web services and web clients and the lessons of the analyses of prior systems [8].

6.1 Names

CREST names specific computations (CA1) and exchanges their expressions between nodes (CA2).

To achieve this exchange, one appealing solution is to physically embed the expressions directly in the URL. Returning to our earlier word count example, the `remote` request can be constructed as depicted in Figure 9. By reusing the existing URL specification and embedding the computation directly in the URL, CREST provides a mechanism for embedding computations inside HTML content - allowing an existing user agent (such as Firefox) to interact successfully (and without its direct knowledge) with a CREST peer. This explicitly permits CREST to be incrementally deployed on top of the current Web infrastructure without requiring wholesale alterations or adoption of new technology. To be more precise about this reuse of the URL format, if *a* is the ASCII text of a expression *e* sent by client *c* to URL *P*://*S*/*u*0/…/*um*–1/ of origin server *S* under scheme *P* then the URL used by *c* is *P*://*S*/*u*0/…/*um*–1/*a*/.

To be clear, CREST URLs are not intended for human consumption, as they are the base mechanisms of computational exchange among CREST nodes; human-readable namings may be provided as needed by higher layers. Among computational nodes, the length of the URL or its encoding is irrelevant and ample computational and network resources are readily available among modern nodes to assemble, transmit, and consume URLs that are tens of megabytes long. In effect, the URL *u*=*P*://*S*/*u*0/…/*um*–1/ is the root of an infinite virtual namespace of all possible finite expressions that may be evaluated by the interpreter denoted by *u*. Finally *u*', a moderately compact and host-independent representation of a computational exchange, may be recorded and archived for reuse at a later point in time (CA3). One possible compact representation of our word count example is provided in Figure 10.

6.2 Services

A single service may be exposed through a variety of URLs which offer multiple perspectives on the same computation (CA1). Each URL may offer a different binding environment or support complementary supervisory functionality such as debugging or management (CA4). Different binding environments (altering which functions are available) may be offered at different URLs which represent alternate access mechanisms to the same underlying computation. We envision that a service provider could offer a tier of interfaces to a single service. For instance, a computation that performs image processing could be exposed as a service. In this hypothetical service offering, a free service interface is exposed which allows scaling up to a fixed dimensions (up to 1024x768) and conversions into only a small set of image formats (only JPEG and GIF). In addition to this free service, another interface could be exposed for a fee (protected via suitable access controls) which places no restrictions on the dimensions of the resized image and offers a wider range of support for various image formats (such as adding RAW or TIFF formats).

In addition, alternative URLs can be used to perform supervisory tasks for a particular service. For long running custom computations, as is the intention for `spawn`, an outside party might desire more insight into the progress and state of the computation. The outside party may also wish to suspend or cancel the computation. In this case, a unique "supervisory" URL `d` can be generated by the CREST interpreter in addition to the `spawn`ed computation's mailbox. Clients can then direct specific `remote` closures to `d` in order to access special debugging and introspection functions. For example, the supervisory environment can provide current stack traces, reports of memory usage, and functions to monitor commu-

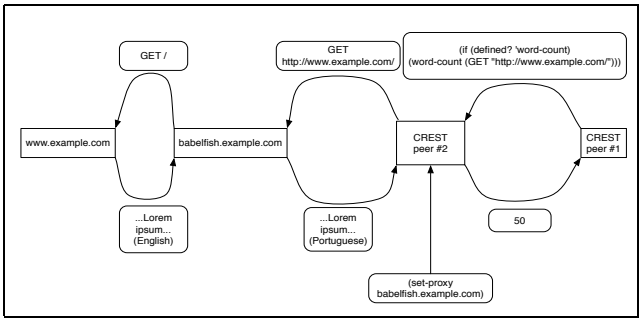


Figure 8. Word count example with an intermediary

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

Figure 9. CREST URL example (expanded)

```
crest://server.example.com/word-count/www.yahoo.com/
```

Figure 10. CREST URL example (condensed)

nications originating from the computation or messages arriving at the mailbox. If the environment chooses, it can also expose mechanisms to suspend or kill the computation. A `remote` closure delivered to this supervisory URL could then combine these debugging primatives to produce a snapshot of the `spawn`ed computation's health. Or, if the supervisory environment supports it, a new `spawn` closure can be delivered to `d` which will automatically kill the computation if the original closure exceeds specific parameters.

6.3 Time

The nature and specifics of the locus of computation may also vary over time. For example, functions may be added to or removed from the binding environment over time or their semantics may change (CA4). One potential reason for this variation is that a service provider wishes to optimize the cost of providing the service depending upon the present computational load. Therefore, in a service representing a complex mathematical function, a provider can offer a more precise version of a function that uses more CPU time during off-peak hours. However, during peak hours when overall computational cycles are scarce, a less precise variant of the function can be deployed which uses less CPU time. Additionally, the interpreter may change as well for the sake of bug fixes, performance enhancements, or security-specific improvements.

Functions in the binding environment may also return different values for the same inputs over time. For example, a random number generator function must vary its output in successive calls in order to be useful. Yet, there is nothing to prevent a locus from being stateful if it so desires. A URL representing a page counter computation that increments on each `remote` invocation would be stateful. It is important to understand that the locus, in addition to everything else, may be stateful and that state, as well as everything else, is permitted to change over time.

6.4 State

It is vital to note that many distinct computations may be underway simultaneously within the same resource. A single client may issue multiple `remote`s or `spawn`s to the same URL and many distinct clients may do the same simultaneously. While a computational locus may choose be stateful (and thus permit indirect interactions between different computations), it is important to also support stateless computations whereby these parallel computations do not have any influence or effect on any other instance within the same computational namespace (CA3). With stateless computational loci, independent parallelization of the evaluation is easily available and straightforward.

In order to address scalability concerns with stateful services (such as a database), specific consistency mechanisms must be introduced to try to regain parallelization. The coarsest-grained consistency mechanism would be to only allow one evaluation of the stateful service at a time as protected by a mutex (akin to the Java `synchronized` keyword). However, as discussed in further detail in [8], the web as a whole tends to require optimizing for substantially higher read volumes than write volumes. Therefore, it is possible to introduce weak consistency models where writes are delayed or processed independently in order to permit highly parallelizable read operations [3, 5, 6, 14].

6.5 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. In contrast, CREST relies upon potentially autonomous computations to exchange and maintain state (CA2, CA3). Given the compositional semantics offered with CREST, we expect that it will be common for one resource to refer to another either directly or indirectly. As a consequence, there may be a rich set of stateful relationships among a set of distinct URLs (CA1). This state will be captured within the continuations that are exchanged between services. A service provider will be able to give its consumers 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 inside of the continuation. As long as the consumer is interested in persisting the stateful relationship, it needs to merely retain the continuation (embedded in an URL) to let the provider resume the state of the service at a later date.

6.6 Transparency

With the computational mechanisms transparently exposed in an URL, a computation can be inspected, routed, and cached. In this way, intermediaries and proxies are strongly embraced by CREST (CA5). This allows a service to scale up as needed by sharing network resources; a single origin server may now be dynamically reconstituted as a cooperative of origin servers (peers) and intermediaries (CA3, CA4). A proxy can be interjected into a request path to record all of the computations exchanged between parties. Additionally, an intermediary can conduct intelligent routing based on just the URL and status line information and, since the computations are also transparent, they can be altered by the intermediary, for example amending computations with wrappers for the sake of debugging.

More importantly, CREST also permits the caching of computations by legacy web caching modules. Since all of the computation is specified directly in the URL, a cache can trivially compare the URL against all the cached prior interactions that it has cached. If

the response is cachable (as dictated by site policy or by the original service provider) and is still fresh according to the HTTP expiration parameters, then that prior interaction can be returned to the requestor without contacting the original service provider. This feature permits the graceful introduction of caching into the CREST environment.

6.7 Migration and latency

Given the dominance of computation and computational transfer CREST applications can successfully minimize network and computational latency by migrating computations. Applications may stitch many computations from multiple service providers into a comprehensive whole, as no one origin server may be capable of supplying all of the functional capability that the client requires.

The physical characteristics of the underlying network connection can have a substantial effect on latency. Recent investigations into cloud computing services suggest that the performance will be subject to and dominated by variations caused by network latency [20]. As a remedy, CREST encourages computation migration, that is, moving the computation closer to the data store, to reduce latency (CA2). For many classes of computations, such as filtering or summation (map/reduce), the reduction is dramatic.

CREST communications are fully asynchronous, though an exemplary peer respects the request/response ordering constraints when communicating with an HTTP/1.1-compliant weak peer (see Section 7). All 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 servers must be receptive to previously generated continuations long after they were first created (on timespans of seconds to months) and restart them seamlessly. All CREST peers employ timers to bound waiting for a response or the completion of a computation. CREST nodes 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 dates."

7. CREST FRAMEWORK & EXPERIENCE

To both facilitate the adoption of CREST and to explore the implications and consequences of the style we have constructed a CREST framework that allows us to build applications in this style. Our framework has two classes of peers: *exemplary* peers and *weak* peers. As depicted in Figure 12, exemplary peers are standalone servers that have a rich set of computational services available. These exemplary peers utilize Scheme as the language in which the computations are written. In order to assist and expose third-party libraries, our Scheme implementation is written on top of Java [18] - so any Java frameworks are accessible from the exemplary peer. To foster interoperability with the existing Web, these exemplary peers can act as a HTTP/1.1 server (to expose the computations running on that peer to browsers) as well as an HTTP/1.1 client (to permit the local computations on that peer to fetch resources on a HTTP/1.1 server or another remote peer). On a modern Intel-class laptop with minimal performance tuning, our exemplary peers can serve dynamic computations in excess of 200 requests per second. In contrast to exemplary peers, weak peers are confined to the restrictions of a modern Web browser and rely upon JavaScript as the fundamental computational foundation. For ease of development and portability for our weak peers, our example applications use the Dojo JavaScript framework. Supported browsers include Mozilla Firefox, Safari, Google Chrome, and Internet Explorer. Mobile devices such as an Apple iPhone and Google Android phone are also supported as weak peers (through their built-in browser applications.)

7.1 Example Application: Feed Reader

Our example application is a is a highly dynamic, reconfigurable feed reader that consumes RSS or Atom feeds. For purposes of comparison, Google Reader or Bloglines may be considered as a starting reference point - however, our application has a much stronger computational and compositional aspect than either system offers today. In addition to merely displaying a feed, one such novel functionality present is that a user can dynamically link the feeds to a tag cloud to display the most popular words in the feed. A screenshot of the running application is presented in Figure 11.

In our example, there are two separate classes of computations that are occuring: the *widget* computations running on the exemplary peers, and the *artist* computations running on the weak peers. An overview of the run-time architecture is provided in Figure 13. There are eight different widgets: a manager (which allows a user, via a weak peer, to create and link widgets), an URL selector, an RSS reader, tag clouds, sparklines, a calendar, a Google News reader, and a QR code (a 2D barcode format). Via a manager widget, these widgets can be linked together - such as the URL selector linking to the RSS reader, indicating that the reader should fetch its feed from a specific URL. Each one of these widget computations may have an associated artist computation which is responsible for visually rendering the state of the widget in a weak peer (such as a browser). In our example, the artists and widgets communicate via exchanging JSON over HTTP - more sophisticated computational exchanges (such as full closure and continuations) are commonly employed among exemplary peers. There does not have to be a one-to-one relationship between artist and widgets - in our example, a manager widget has two separate artists - one which lets the user add new widgets and another artist (the mirror) which visually depicts the entire state of the application using boxes, arrows, and color.

Using our CREST framework, all eight of our widgets total under 450 lines of Scheme code - the largest widget is the feed reader widget computation which is approximately 115 lines of code. All of our artists are written on top of the Dojo JavaScript framework and comprise approximately 1,000 lines of JavaScript and HTML.

7.2 Design Considerations: Feed Reader

Using the considerations presented in Section 6 as our guide, we now discuss their impact and how they manifest themselves in our demostration example.

Names. Each instantiation of a widget computation has its own unique per-instance URL. Through this URL, artist computations running in the confines of a week peer or another widget computation on the same or different peer can deliver a message to a specific widget computation. These messages can either fetch the current state of the computation via a GET request (such as to retrieve the current tag cloud data) or update the state of the computation via a POST request (in order to deliver a new set of words to the tag cloud). Within the weak peer, artist computations have

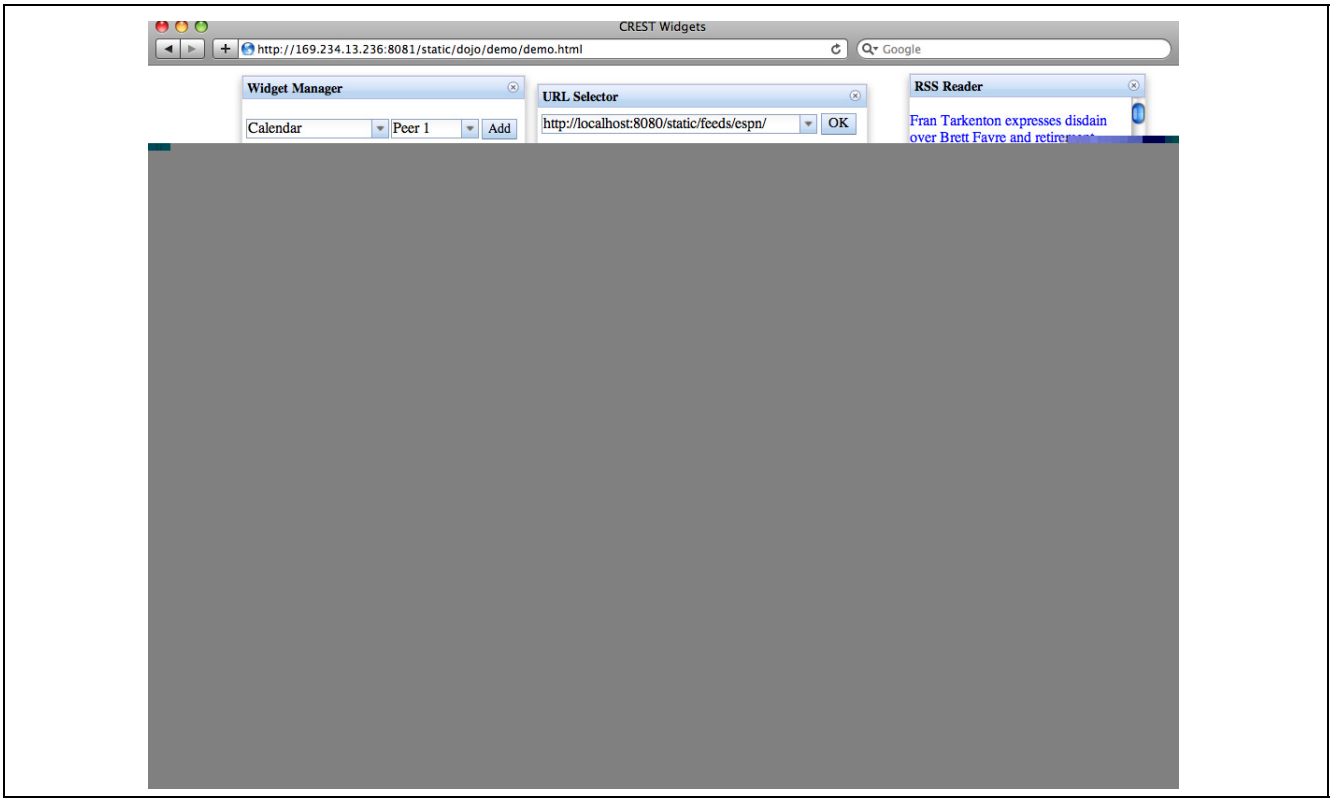


Figure 11. Screenshot of CREST-based Feed Reader application

unique identifiers within the local DOM tree of the browser. However, these weak peers are restricted to only communicating with exemplary peers and can not expose any services or directly interact with other weak peers.

Services. Both the widget and artist computations in our example application are independently managed and run. Each widget computation locally defines what services it provides - such as the URL widget component permitting the storing and retrieval of the current value (in this case, a feed URL). In contrast, the widget manager computation offers the ability to spawn new widgets as well as maintaining information about already instantiated widgets.

It should also be noted that multiple artist computations may offer different perspectives on the same service. In our example, both the mirror and manager artist computations provide alternative views of the current state of the system (one as a list of existing widget computations, the other as a graphical representation of the existing computations).

Time. Over time, the computations offered by the example application will vary. The computations are not defined, created, or even initialized until they are explicitly activated by the manager widget. Additionally, there is a set of relationships between widget computations (links) that are created and destroyed over time. In our example, this set of relationship determines the data flow from one widget computation to another widget computation.

State. In our example application, the widget computations have the ability to maintain a local per-instance state. Our feed reader widget computation will retrieve and parse the designated feed on a periodic basis as defined by a clock-tick. The feed reader widget computation will then store the parsed feed as a local stateful value. By maintaining the state (essentially caching the parsed feed), the feed reader widget can easily scale since it must only return the cached representation rather than constantly retrieving and parsing the feed in response to a given request. In contrast to the widget computations, our artist computations are all stateless. The artist computations are configured to periodically poll their affiliated widget computation for its state and then renders that state accordingly.

Computation.

The artist computation's responsibility is to (visually) render the state of its affiliated widget computation into a specific form. The widget computations executing on the exemplary peer have no restrictions on what they can compute modulo security implications as discussed in Section 8. Through the use of links between widget computations, dynamic composability of services is supported. For instance, the Google News widget can be dynamically linked to the calendar and tag cloud widget computations in order to search for a given keyword (from the tag cloud) at a given date in the past (from the calendar).

Transparency. The demonstration supports varying degrees of computational exchange, namely "shallow" copy versus "deep" copy. In a shallow copy, all weak peers share among themselves a single collection 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

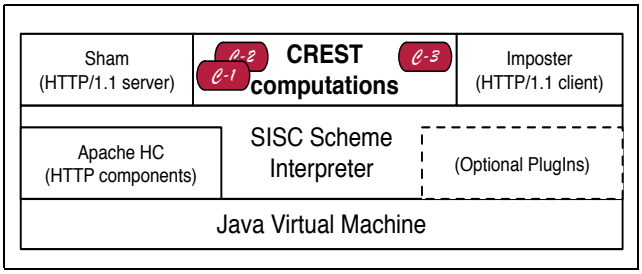


Figure 12. Exemplary CREST Peer

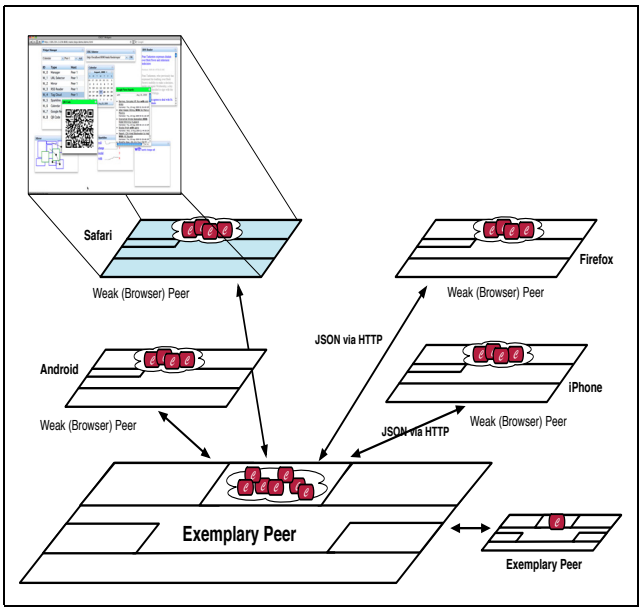


Figure 13. Feed Reader Architecture

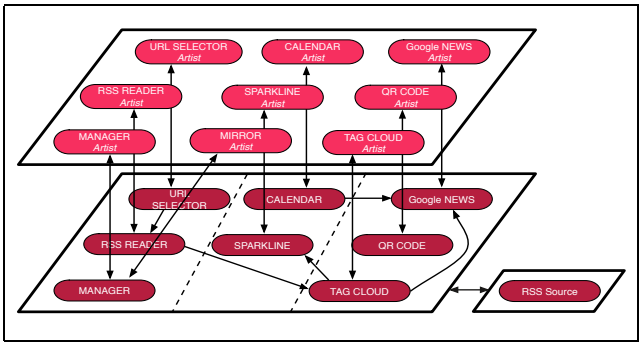


Figure 14. Feed Reader Computation List

updates for the same widget computations. Under deep copy, a new weak peer creates a fresh, forked collection of widget computations whose computational states are a continuation, captured at the instant of the join, of the parent collection. This weak peer will now observe, from that point forward, an independent, evolving state. In this way, the deep copy creates a new version of the feed reader application at the time the continuation is created.

Interestingly, due to the accelerated timeframes of our example application (feeds were updated every three seconds), we had to introduce cache-busting parameters in our artist computations on the Google Android phones because of the browser's aggressive caching behavior.

Migration and latency. Since all relationships between computations are identified by a URL, these computations may either be remote or local. In this way, widget computations can be migrated with abandon as 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.

With regards to latency, the division between artist computations (which draw the local display) and the widget computation (which maintains the state) allows for minimal transference of data between nodes. Upon creation of the widget computation, the relevant artist computation is transferred and instantiated on the weak peer. Therefore, by providing the artist computation as an output of the widget computation, the widget computation has complete control over what formats the artist computation require and can hence use any optimized format that it desires.

8. ADDITIONAL RELATED WORK

Web Services impose a service perspective on large-scale web systems however, its flaws are numerous and fundamental including inappropriate service granularity [22], an unsuitable invocation mechanism [24], and intermingling of data and metadata causing high latency [4]. Google Wave [15] employs a classic client/server architecture for which the medium of exchange is XML documents and state deltas encoded as operational transforms. CREST resembles a web-scale actor model [2] and like [11] exploits a functional language as the implmenentation medium for its actor-like semantics. The demonstration sketched above resembles Yahoo Pipes [26] or Marmite [25] however, unlike these systems, all significant computation is distributed outside the browser in CREST peers and within the browser the only computation is for the sake of rendering.

Security is a significant issue for mobile code systems and several distinct mechanisms are relevant for CREST. Strong authentication is a vital starting point for trust, resource allocation, and session management and for which we will employ self-certifying URLs [13]. Mechanisms for resource restriction, such as memory caps and processor and network throttling, are well-known, however, environment sculpting [23] may be used to restrict access to dangerous functions by visiting computations and is the functional analog of the capability restriction in Caja [17]. In addition, since an exemplary peer executes atop the Java Virtual Machine all of the security and safety mechanisms of the JVM may be brought to bear. Finally, CREST exemplary peers will employ byte code verifiers for remote and spawn computations and various forms of law-governed interaction [19] to monitor and constrain the behavior of collections of computations executing internet-wide on multiple peers.

9. SUMMARY

This paper represents the culmination of a prolonged study of modern Web applications and how dynamism in applications both has changed the Web during the past decade and how it can be further exploited. Our earlier paper at ESEC/FSE 2007 detailed the first part of this study: analyzing existing systems and providing an initial CREST model for rationalizing their behavior. This paper has tackled the second part, showing how dynamism, in the form

of computations, can be made the central aspect of distributed Web applications. The refined CREST axioms presented here, accompanied by design guidance for the application of those principles, provides the foundation for a dramatic change in the Web. The demonstration application described in the paper indicates how these broad yet specific principles can be supported in practice. The implementation framework upon which the sample application was built is generic, is fully backwards compatible with the existing Web infrastructure, and will be the basis for our next phase of investigation, in which we anticipate examining how this computational web can be used to solve problems in other areas, ranging from the smart energy grid to situational awareness to high-speed streaming video applications. In addition to these application-based studies, we anticipate investigations into numerous support areas, including development and testing techniques and tools, implementation of the security elements, and provision of services for computation search and composition.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Numbers 0438996 and 0820222. Thanks to Yongjie Zheng and Alegria Baquero for their assistance in creating the CREST framework.

11. REFERENCES

[1] Abelson, H., Sussman, G.J., and Sussman, J. Structure and Interpretation of Computer Programs. Second ed. 683 pgs., MIT Press, 1996.

[2] Baker, H. and Hewitt, C. Laws for Communicating Parallel Processes. MIT Artificial Intelligence Laboratory Working Papers, Report WP-134A, May 10, 1977. <http://hdl.handle.net/1721.1/41962>.

[3] Chang, F., Dean, J., Ghemawat, et al. Bigtable: A Distributed Storage System for Structured Data. In Proceedings of the OSDI'06: Seventh Symposium on Operating System Design and Implementation. Seattle, WA, November, 2006. <http://labs.google.com/papers/bigtable-osdi06.pdf>.

[4] Davis, D. and Parashar, M. Latency Performance of SOAP Implementations. In Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. p. 407-12, Berlin, Germany, May 21-24, 2002.

[5] DeCandia, G., Hastorun, et al. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review. 41(6), p. 205-220, 2007.

[6] Dubois, M., Scheurich, C., and Briggs, F. Memory access buffering in multiprocesses. In Proceedings of the 13th Annual International Symposium on Computer Architecture. p. 434-442, Tokyo, Japan, 1986.

[7] Erenkrantz, J.R., Gorlick, M., Suryanarayana, G., and Taylor, R.N. From Representations to Computations: The Evolution of Web Architectures. In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering (ESEC/FSE). p. 255-264, Dubrovnik, Croatia, Sept 3-7, 2007.

[8] Erenkrantz, J.R. Computational REST: A New Model for Decentralized, Internet-Scale Applications. PhD Thesis. Information and Computer Science, University of California, Irvine, 2009.

[9] Fuchs, M. Dreme: for Life in the Net. PhD Thesis. New York University, 1995.

[10] Fuggetta, A., Picco, G.P., and Vigna, G. Understanding Code Mobility. IEEE Transactions on Software Engineering. 24(5), p. 342-361, May, 1998.

[11] Ghosh, D. and Vinoski, S. Scala and Lift - Functional Recipes for the Web. IEEE Internet Computing. 13(3), p. 88-92, 2009.

[12] Halls, D.A. Applying Mobile Code to Distributed Systems. Ph.D. Thesis. University of Cambridge, 1997.

[13] Kaminsky, M. and Banks, E. SFS-HTTP: Securing the Web with Self-Certifying URLs. MIT Laboratory for Computer Science, 1999. <http://pdos.csail.mit.edu/~kaminsky/sfs-http.ps>.

[14] Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM. 21(7), p. 558-565, July, 1978.

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

[16] Matthews, J., Findler, R.B., Graunke, P., Krishnamurthi, S., and Felleisen, M. Automatically Restructuring Programs for the Web. Automated Software Engineering. 11(4), p. 337--364, 2004.

[17] Miller, M.S., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized JavaScript. <http//google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf>, PDF, June 7, 2008.

[18] Miller, S.G. SISC: A complete Scheme interpreter in Java. Technical Report, 2003. <http://sisc.sourceforge.net/sisc.pdf>.

[19] Minsky, N.H. and Ungureanu, V. Law-governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. ACM Transactions on Software Engineering Methodology. 9(3), p. 273-305, July, 2000.

[20] Palankar, M.R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. Amazon S3 for science grids: a viable solution? In Proceedings of the 2008 International Workshop on Data-aware Distributed Computing. p. 55-64, Boston, MA, 2008.

[21] Queinnec, C. The Influence of Browser on Evaluators or, Continuations to Program Web Servers. In Proceedings of the International Conference on Functional Programming. Montreal, Canada, 2000.

[22] Stamos, J.W. and Gifford, D.K. Remote Evaluation. ACM Transactions on Programming Languages & Systems. 12(4), p. 537-564, 1990.

[23] Vyzovitis, D. and Lippman, A. MAST: A Dynamic Language for Programmable Networks. MIT Media Laboratory, Report, May, 2002.

[24] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S.C. A Note on Distributed Computing. Sun Microsystems, Report TR-94-29, November, 1994. <http://research.sun.com/techrep/1994/abstract-29.html>.

[25] Wong, J. and Hong, J.I. Making Mashups with Marmite: Towards End-user Programming for the Web. In Proceedings of the SIGCHI Conference on Human Factors In Computing Systems. p. 1435-1444, San Jose, California, USA, April 28-May 3, 2007.

[26] Yahoo Inc. Pipes. <http://pipes.yahoo.com/pipes/>, 2009.