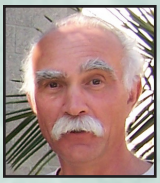
ISR
Institute for Software Research
University of California, Irvine

Communication and Capability URLs in COAST-based Decentralized Services



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

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

December 2012
ISR Technical Report # UCI-ISR-12-10

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

Communication and Capability URLs in COAST-based Decentralized Services

Michael M. Gorlick and Richard N. Taylor

Abstract Decentralized systems are systems-of-systems whose services are governed by two or more separate organizations under distinct spheres of authority. Coordinated evolution of the various elements of a decentralized system may be difficult, if not impossible, as individual organizations evolve their service offerings in response to organization- and service-specific pressures, including market demand, technology, competitive and cooperative interests, and funding. Consequently, decentralized services offer unique challenges for evolution and adaptation that reach well beyond any one single organizational boundary. However, client-driven service customization and tailoring is a powerful tool for meeting conflicting, independent client demands in an environment where disorderly and uneven service evolution predominates. Computational State Transfer (COAST) relies on capability security to minimize the risks of client-driven customization, for which fine-grain management of communication capability is critical. We introduce the Capability URL (CURL) as the unit of communication capability and show how two distinct mechanisms, communication capability and mobile code, can be combined to express and enforce constraints on the communications among decentralized computations.

1 Introduction

Decentralized software systems are distributed systems that span multiple, distinct spheres of authority—participants may unilaterally change their behaviors in ways that may or may not be compatible with the needs or goals of the other members. The web is a prime example; servers come and go, links are created and broken, and mashups are deployed that rely upon the APIs of other web sites. Integrated supply

Michael M. Gorlick
University of California, Irvine, California, USA, e-mail: mgorlick@acm.org

Richard N. Taylor
University of California, Irvine, California, USA, e-mail: taylor@ics.uci.edu

UCI Institute for Software Research Technical Report #UCI-ISR-12-11, December 2012

chains are another example; the designs of NASA's Curiosity rover and both the Boeing 787 and Airbus A380 commercial aircraft required the network-mediated collaboration of thousands of engineers in many dozens of companies. Decentralization appears in numerous domains including disaster response, coalition military command, commerce, finance, education, and scientific research. A decentralized system may be open or closed. In the former participation is loosely constrained if at all, while in the latter participation is governed by agreements (with varying degrees of formality, rigor, and enforcement) among the participants. The global web is an open system and anyone can participate, but participating in a business-to-business supply chain system demands negotiations and contracts. Joining or leaving the global web can be done on a whim while joining or leaving a supply-chain system is not undertaken lightly.

All decentralized systems are intrinsically dynamic: members join and leave, service relationships change, system implementations and deployments vary as will their rates of evolution and adaptation, and members adapt to the changing business, financial, or regulatory environment. Both open and closed decentralized systems raise concerns of security and trust and neither are immune to malicious behavior.

COmputAtional State Transfer (COAST), an architectural style based on the idiom of *computation exchange*, targets decentralized systems and their security issues. COAST has its roots in two earlier architectural styles, REST and CREST. The World Wide Web is one of the best known decentralized applications and REST (REpresentational State Transfer) is the architectural style [21] underlying the web's evolution, performance, and scaling. Code mobility was always part of the REST style (for example, Javascript embedded in HTML pages) with the nominal goal of fostering browser-side display of new media types or reducing application latency. In other words, computation mobility in REST was subservient to content transfer and focused largely on optimizing the transfer and interpretation of resource representations.

On one hand REST was a huge success, as adherence to the REST principles set the stage for the web's unparalleled expansion. However, REST has many shortcomings. From the outset there was insufficient support for differentiation, as the rapid adoption of cookies, in violation of REST precepts, demonstrated. The emergence of Ajax (mashups) and the exploitation of computation in the browser suggested a more prominent role for mobility—constructing and deploying customizations and application services [17]. At the same time inadequate security led to numerous breaches.

Inspired by REST, the evolution of web architecture, and the rapid introduction of Ajax and Web Services, we formulated CREST [14, 15, 17], an architectural style in which computations displaced content representations as the unit of exchange among hosts. In CREST, actively executing computations (as opposed to "resources" as abstracted black-boxes of information) were named by URLs and computations exchanged state representations reified as closures and continuations. Our trials of CREST, including a customizable, collaborative feed reader and analyzer [14, 16] and Firewatch [24], a system for wildfire detection and response, showed considerable promise for constructing highly dynamic systems. However, CREST needlessly

inherited many constraints from Web architecture and, like REST before it, failed to address security in any comprehensive manner.

COAST, the successor to CREST, is a style for which security is a dominant concern and whose mechanisms allow hosts to minimize the risk of executing visiting computations on behalf of clients. A detailed view of COAST accompanied by a demonstration application is given in [26]. Here our focus is communication security, whereby COAST hosts manage communications among computations and modulate access to critical services. However, before introducing the communication mechanisms we describe our domain of interest, decentralized SOAs, from the perspective of computation exchange and from there move to the COAST style itself. With that behind us we turn to our principal contribution, the details of Capability URLs, and present examples of their use.

2 Decentralized Systems via Computation Exchange

Decentralized systems whose constituent subsystems operate under distinct spans of authority must meet two conflicting goals: protecting valuable fixed assets (such as servers, databases, sensors, data streams, and algorithms) and meeting the evolving service demands of a diverse client population.

Computation exchange (the computational analogue of content exchange) is the bilateral exchange of computations among decentralized peers. In this regime, content delivery is a by-product of the evaluation of computations exchanged among peers. Computation exchange exploits existing core organizational functions, processes, and assets to create higher-level customized services, but imposes significant security obligations.

Computation exchange generalizes and subsumes a number of well-known styles for distributed computing, including remote procedure call [7, 40], remote evaluation [47, 49], REST [21], and service-oriented architectures [19]. From the perspective of computation exchange remote procedure call is an exchange containing a single function call, remote evaluation is an exchange containing an entire function body, REST is an exchange of a small set (`GET`, `PUT`, `POST`, `DELETE`, and so on) of single function calls accompanied by call-specific metadata, and service-oriented architectures are higher-order compositions of remote evaluation.

Computation exchange induces all of the risks associated with mobile code [22] including waste of fungible resources (processor cycles, memory, storage, or network bandwidth), denial of service via resource exhaustion, service hijacking for attacks elsewhere, accidental or deliberate misuse of service functions, or as a springboard for direct attacks against the service itself. A computation accepted from a trusted source may be erroneous or misapply a service function due to honest misunderstanding or ambiguity. Even a correct computation may expose previously-unknown bugs in critical functions, leading to inadvertent loss of service.

For decentralized systems authentication, secrecy, and integrity are necessary but insufficient for asset protection as there is no common defendable security perimeter

when function is integrated across the multiple, separate trust domains [56]. Here an attack on one authority threatens all. At best a breach may lead to failures in other trust domains. At worst a breached authority may undertake an "insider" attack against its confederates. With this in mind decentralization demands that security be everywhere always. Applications that cross authority boundaries inherently bring security risks; adaptations in such contexts only increase the peril; hinting that security should be a core *architectural* element.

3 The COAST Architectural Style

COmputAtional State Transfer (COAST) is an architectural style for decentralized and adaptive systems [26]. Its applications have origins in CREST and, before that, the REST architectural style. COAST targets decentralized applications where organizations offer execution hosts (called *islands*) whose base assets can include databases, sensors, devices, execution engines, domain-specific functions, or access to distinctive classes of users. In COAST, third-party organizations create their own custom-tailored versions of services (modulo the constraints imposed by the asset owner) by dispatching computations to asset-bearing islands. For instance, a monitor-and-alert function may be defined by one user to run periodically on an island offering access to a to a collection of environmental sensors. Mobile code both implements the computations in a COAST system and defines the messages exchanged among those computations. Decentralized security and guarding against untrusted or malicious mobile code are principal island concerns—the style mandates architectural elements that when used appropriately provide access, resource, functional, and communication security. In exchange for the complexity imposed by these security mechanisms, COAST allows the construction of on-demand tailored services and enables a wide range of dynamic adaptations in decentralized systems.

COAST security relies on the *Principle of Least Authority* (POLA) [45] and *capability-based security* [8]. POLA dictates that security is a product of the authority given to a principal (the functional power made available) and the rights given to the principal (the rights of use conferred with respect to that authority). At each point within a system a principal must be simultaneously confined with respect to both authority and rights. A *capability* is an unforgeable reference whose possession confers both authority and rights of access to a principal. COAST is one architectural style for computation exchange, just as "pipes and filters" is one of many architectural styles for data processing. COAST's constraints mandate where, when, and how authority and rights are conveyed.

The COAST style states:

- All services are computations whose sole means of interaction is the asynchronous messaging of closures (functions plus their lexical-scope bindings), continuations (snapshots of execution state [20]) and binding environments (maps of name/value pairs [29])

- All computations execute within the confines of some execution site $\langle E, B \rangle$ where E is an execution engine and B a binding environment
- All computations are named by Capability URLs (CURLs), an unforgeable, tamper-proof cryptographic structure that conveys the authority to communicate
- Computation x may deliver a *message* (closure, continuation, or binding environment) to computation y only if x holds a CURL u_y of y
- The interpretation of a *message* delivered to computation y via CURL u_y is u_y-dependent

For example, Alice operates a COAST-based high-performance image processing service. Her clients dispatch computations for processing, enhancing, and analyzing a wide variety of commercial, industrial and scientific imagery to her service. The execution sites in her server farms are managed by her own COAST computations whose CURLs denote site-specific processing varying across a spectrum of performance and functionality.

Bob, whose machine shop manufactures custom aviation and motorcycle racing components, is one of Alice's clients. His COAST-based automated visual inspection system dispatches quality-control computations containing high-resolution digital photographs of components to Alice's execution sites for final inspection. Alice's proprietary algorithms combined with Bob's customized closures for component- and use-specific analysis help Bob maintain a high level of quality.

Carol, another of Alice's clients, analyzes medical imagery for physicians and medical testing labs. The sheer volume of the imagery, along with strict medical privacy regulations, prevent Carol from shipping her closures, binding environments and imagery to an outside processor (as Bob does for his custom racing components), so Carol has licensed an image processing library from Alice that has been integrated into the execution sites of her own in-house COAST-based services.

Carol obtains analytical tools for her imagery from Dave, whose biotechnology company deploys computations for narrowly targeted tissue analyses to COAST sites. Carol dispatches service requests (as computations) to Dave's COAST services. Each of her requests prompts Dave's computations to generate a custom analysis (as a closure or continuation) optimized to meet her request-specific needs and constraints. Included in each of Carol's requests is a nondelegable, "use-once-only" CURL referencing one of her execution sites containing privacy-sensitive medical images.

Dave deploys his customized analysis to Carol's site via Carol's CURL. As Dave's analysis executes on Carol's COAST infrastructure her execution site prevents Dave's computation from accessing any other confidential imagery. Her COAST-based monitoring and auditing infrastructure tracks the execution of Dave's analysis from beginning to end, ensuring that it does not violate patient privacy regulations. The nondelegable, use-once-only CURL prevents Dave from sharing the CURL with any other COAST site (nondelegation), and, as it can never be used more than once, neither Dave nor any attacker that infiltrates Dave's infrastructure can ever send more than a single computation to Carol's request-specific, privacy-sensitive, execution site.

COAST offers two distinct forms of capability, functional capability—what a computation may do and communication capability—when, how, and with whom a computation may communicate. Functional capability is regulated by execution sites while communication capability is regulated by CURLs. These two mechanisms, execution sites and CURLs, can be combined in many different ways to elicit domain- and computation-specific security.

Execution Sites: Over its lifespan each COAST computation is confined to an *execution site* $\langle E, B \rangle$.The execution engine E may vary from one exectution site (and computation) to another: for example, a Scheme interpreter or a JavaScript just-in-time compiler. The execution engine defines the execution semantics of the computation and the machine-specific limits (e.g., resource caps) imposed upon the computation.

The binding environment B contains all of the functions and global variables offered to the computation at that execution site. Names unresolved within the lexical scope of c (the *free variables* of c) are resolved, at time of reference, within the binding environment B. If B fails to resolve the name the computation is terminated.

Both the execution engine and binding environment of an execution site $\langle E, B \rangle$ may vary independently and multiple sites may be offered within a single address space. E may enforce site-specific semantics: for example, limits on the consumption of resources such as processor cycles, memory, storage, or network bandwidth; rate-throttling of the same; logging; or adaptations for debugging. The contents of B may reflect both domain-specific semantics (for example, B contains functions for image processing) and limits on functional capability (B contains functions for access to a *subset* of the tables of a relational database).

Capability URLs: CURLs convey the ability to communicate between computations. A CURL u issued by a computation x is an unguessable, unforgeable, tamper-proof reference to x, as it contains cryptographic material identifying x and is signed by x's execution host. A CURL referencing x may be held by one or more other computations y. CURL u is a capability that designates the network address of computation x, contains arbitrary x-specific metadata (including closures), and grants to any computation y holding u the power to transmit messages to x. When y transmits a message m to x via CURL u both the message m and the CURL u are delivered together to x.

A computation x uses the CURLs it issues to constrain its interactions with other computations and to bound the services it offers. The rationale for constraining interaction in this way is based upon security concerns. A computation y, holding a CURL for x, can send arbitrary closures to x in the expectation that x will evaluate those closures in the context of some x-specific execution site $\langle E, B \rangle$. Therefore x must defensively minimize the functional capability that it exposes to visiting closures.

A computation can accumulate communication capability in the form of additional CURLs. For any computation, CURLs conveying additional communication capability are:

- Contained in the closure defining the computation
- Returned as values by functions invoked, or

- Embedded as values in the messages received.

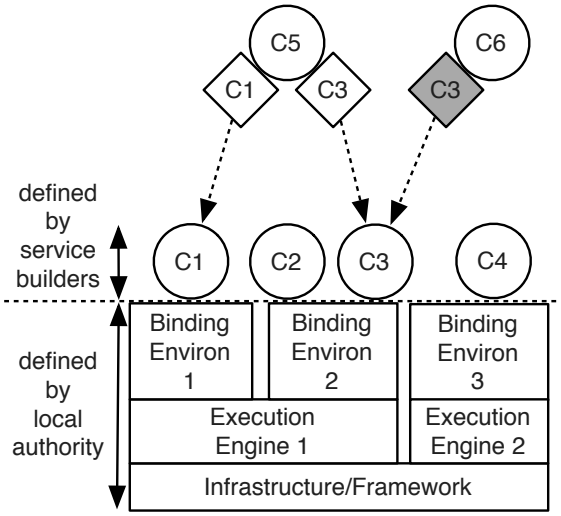


Fig. 1 The notional structure of a COAST execution host where a trusted code base allocates execution engines and binding environments to computations, whose implementations are sourced from a variety of other organizations. Computations (given as circles) C5 and C6 each hold a distinct CURL (given as diamonds) denoting different services offered by computation C3. By holding those CURLs C5 and C6 possess the right (denoted by dotted arrows) to dispatch mobile code as messages to C3 for execution.

Constructing COAST Applications: A COAST application is constructed from multiple services available at distinct, decentralized, execution sites, each of which offers location- and organization-specific primitives. Those services themselves may depend on customized collaborations with yet other services. Figure 1 illustrates the notional structure that COAST induces on execution hosts.

Computations are expressed in MOTILE, a single-assignment functional language with functional, persistent data structures [41] (all data structures are immutable). A COAST *island* is a single, uniform address space occupied by one or more computations. Computations residing on an island I issue one or more CURLs to the computations with whom they wish to communicate. A CURL *u for x* is a CURL generated by x. For the sake of security, communication among computations is "communication by introduction" meaning that computation x can't communicate with computation y unless it already holds or obtains (via function call or messaging) a CURL issued by y.

4 Capability URLs in Detail

Each CURL u denotes a specific computation x and contains a self-certifying network address [31], a path (a list of MOTILE values), and arbitrary metadata. To ensure the integrity of "introduction only" it must be effectively impossible to guess, forge, or alter a CURL. CURLs are a first-class, immutable capability in MOTILE; hence, within the confines of a legitimate island, it is impossible for a MOTILE computation

to forge a CURL or alter one surreptitiously. Every island *I* holds a public/private key pair and guarantees the integrity of the CURLs that its computations issue by signing each with its private key.

For the sake of safety and security islands must manage and limit access to both fungible resources (such as memory or bandwidth) and island-specific assets (such as sensors or databases). Restricting the lifespans of computations may help an island stave off resource exhaustion and limiting the total number of messages that a computation may receive or the rate at which they are delivered can limit access, improve performance, or reduce the severity of computation-specific denial of service attacks. These forms of resource security protect against malicious visiting computations intent on resource attacks or exploiting the island as a platform for attacks directed elsewhere.

With these primitive mechanisms at hand it is trivial to generate a "once only" CURL that is invalid after a single use. Finite CURL lifespans allow computations to offer time-limited services to their clients; for example, such CURLs can be used by a transaction coordinator to enforce time limits among the participants of a two-phase commit. An e-commerce service can combine lifespans with use counts to generate the CURL-equivalent of limited-offer coupons or gift cards, and rate limits are useful in "introductory" promotions in which the service may want to bound the rate of use by newcomers.

The CURLs generated by a computation *x* draw upon a tree of "resource accounts" whose root is the resource account granted to computation *x* "at birth" by the island *I* on which *x* resides. Each account has a finite lifespan and contains a "balance" comprising a use count and rate limit. The initial balance allocated to a new account is "withdrawn" from its parent account and the lifespan of the new account is never more than the lifespan of the parent account. Many accounts may derive from the same parent account and many CURLs may share a single resource account in common.

```
(let* ((path (list "question" "ultimate"))                          1
       (metadata                                                    2
        (list (cons "name"      "Arthur Dent")                      3
              (cons "residence" "Earth"))))                         4
       (I@ (curl/new (resource/root) path metadata)))               5
  (curl/send                                                        6
   Guide@                                                           7
   (list "SPAWN" (lambda () (curl/send I@ 42))))                    8
  (receive))                                                        9
```

Fig. 2 A simple MOTILE program. Island *I* sends a closure to island *J* for execution that does nothing but transmit the the number 42 back to island *I*.

A Simple MOTILE Program: Figure 2 is an example of a simple program in which a trivial closure is dispatched by island *I* to a remote island *Guide* for execution and a constant value is transmitted back to island *I*. Line 3 binds the variable `I@`

to a CURL for computation *x* on island *I*. The function `resource/root` always returns the root resource account of the calling computation; the MOTILE function `curl/new` (line 3) generates a CURL given a resource account from which the CURL draws its resources (use count, rate limit, and lifespan), a path (line 1), and metadata (line 2).

The function `curl/send` given in lines 4–6 transmits a spawn command (the second argument, line 6) to the computation denoted by the first argument, a CURL for island *Guide* bound to the variable `Guide@` (line 5; the details of how computation *x* acquired CURL `Guide@` are omitted for the sake of brevity and clarity). The closure, `(lambda ...)`, evaluated by an execution site of island *Guide*, immediately transmits the message `42` back to computation *x* on island *I* via CURL `Guide@`. The MOTILE function `receive`, called on line 7 of computation *x*, blocks until a message *m* for computation *x* arrives and returns that message *m* as its value.

```
(define (palindrome? s)                                                  1
  (define (traverse? s left right)                                       2
    (or                                                                  3
     (= left right)                                                      4
     (> left right)                                                      5
     (and                                                                6
      (eq? (string-ref s left) (string-ref s right))                     7
      (traverse? s (add1 left) (sub1 right)))))                          8
  (traverse? s 0 (sub1 (string-length s))))                              9
                                                                         10
(let* ((reply (promise/new 60.0))                                        11
       (reply/promise (car reply))                                       12
       (reply/curl    (cdr reply))                                       13
       (palindromes                                                      14
        (lambda ()                                                       15
          (curl/send                                                     16
           reply/curl (filter (words/get) palindrome?)))))               17
                                                                         18
  (curl/send J@ (list "SPAWN" palindromes))                              19
  (promise/wait reply/promise 60.0 #f))                                  20
```

Fig. 3 A computation on island *I* dispatches a closure to island *J* to obtain all of the palindromes in in a dictionary of words.

A Client-Defined Service: Figure 3 illustrates sending a closure from island *I* to extract all of the palindromes contained in a database of words maintained by island *J*. Since island *J* has no predefined function for detecting palindromes, the computation on island *I* defines (lines 1–9) a function `palindrome?` that accepts a string *s* and returns true (`#t`) if *s* is a palindrome and false (`#f`) otherwise. MOTILE uses promises to bridge the gap between functional programming and asynchronous messaging. A *promise* is a proxy object for a result that is initially unknown because the computation of its value has yet to be initiated or is incomplete. Line 11 creates a new promise with a lifespan of 60 seconds. In MOTILE a promise consists of two

elements: the promise object proper (`reply/promise` in line 12) and a single-use CURL, `reply/curl` in line 13, by which the result of the promise will be resolved by some computation .

Lines 14–17 define the closure, `palindromes`, that will be transmitted to island *J* for evaluation. `palindromes`, when executed by island *J*, first applies the *I*-defined predicate `palindrome?` as a filter to the contents of the word database and then sends the result of that filtering (a list, possibly empty, of the palindromic words in the database) to the island computation denoted by the CURL `reply/curl`. The functions `filter` and `words/get` are resolved in the binding environment of the closure's execution site on island *J*. Line 19 is the transmission by *I* to *J* of the request to evaluate the closure `palindromes`. The variable `J@` is a CURL for island *J* denoting the target execution site for the `palindromes` closure. Finally, at line 20, the computation on island *I* waits a maximum of 60 seconds for the spawned computation to complete and return its result. If for some reason the spawned computation is unable to complete its task in the time allotted the result of the promise will be the value `#f` (false) given in line 20.

The program of Figure 3 is a classic example of moving computation close to the data that it demands and illustrates an effect that is difficult to achieve in a RESTful system; island *J* may easily host a large dictionary of words but it's not likely to implement a service expressly designed for extracting palindromes. However, that omission is irrelevant in COAST-based systems since a client is free to compose client-specific higher-order services from the primitives found in the execution sites of island *J*. No such provision exists in RESTful services.

Provider-Issued Mobile Code in CURLs: A computation may embed closures as metadata in the CURLs that it issues and use those embedded closures as the interpreters of the messages that it receives. As the CURL is tamper-proof, the receiving computation (by definition the issuer of the CURL) may safely rely on any state and mobile code the CURL contains. When the computation first constructs and issues the CURL, it ensures that the CURL contains all of the static state (including arbitrary generated closures) that the computation will need in the future to serve the holder(s) of the CURL. In this manner computations, in addition to granting the capability to communicate, can enforce fine-grained constraints on the interpretation of messages. For example, a computation *x* may issue a CURL to *y* that allows *y*'s mobile code, when sent to *x*, to call only one particular function that *x* selects and makes available.

For instance, an e-commerce site wants to issue CURLs as coupons for a book sale where three popular books, identified by ISBN numbers $b1, b2, b3$, will be on sale for a month at 80% of the list price, but only on the even days of the month-long sale.

The construction of such a CURL is given in Figure 4. Lines 1–4 specify the derivation (via function `resource/new` at line 2) of a CURL-specific resource account, `sale`, from the root account (line 3) of the computation. At line 1–4 `sale` is granted a balance of three total uses, a rate limit of once every seventeen seconds, and a total lifespan of 30 days (`timespan/seconds` at line 3 takes days, hours, minutes, and seconds and converts that span of time to total seconds).

```
(let* ((sale                                                    1
        (resource/new                                           2
         (resource/root)                                        3
         3 (/ 1.0 17.0) (timespan/seconds 30 0 0 0)))           4
       (day/even?                                               5
        (lambda () (even? (date/day (date/now)))))              6
       (path (list "books" "sale"))                             7
       (metadata                                                8
        (list (cons "ISBNs"    (list b1 b2 b3))                 9
              (cons "gate"     day/even?)                       10
              (cons "discount" 0.80))))                         11
  (curl/new sale path metadata))                                12
```

Fig. 4 Generating a time-limited, day-specific sales coupon as a CURL.

`day/even?` at lines 5–6 is a provider-generated MOTILE predicate that returns true if the current day of the calendar month is an even integer and false otherwise. `date/now` and `date/day` are provider-side calendrical functions. `date/now` returns the current date as a structure and `date/day` extracts the day of the month (1–31) from that structure. Line 7 defines the path for the CURL to be generated and lines 8–11 define the metadata to be included in the CURL as key/value pairs: the ISBNs of the books on sale, the gate function defined by the provider to determine the validity of the "coupon" and the amount of the sale discount. Finally, line 12 generates and returns the desired CURL.

When the e-commerce site receives a purchase request message sent by way of a "coupon" CURL it passes the CURL and message on to the book sale computation only if the message arrived before the expiration date of the CURL. The book sale computation executes the gate function contained within the metadata of the CURL to determine if the coupon is valid. If so it allows the purchase to proceed; otherwise the request is rejected. As the book sale computation is ignorant of the details of the gate function included in the CURL metadata the e-commerce site provider can easily generate customized sale coupons, each with different gate functions.

Service Implementations in CURLs: Figure 5 illustrates how a CURL can carry a service implementation; here generating custom ranges of real random numbers. Lines 1-3 define a utility function `random/new` that returns a customized random number generator as a closure (line 3). The provider-side function `random` returns a real random number in the open range $[0,1)$. `service/custom` returns a customized CURL for a client requiring a random number service using bounds, `low` and `high`, specified by the client. The CURL is granted a use count of 100, a rate limit of 7.5Hz and a lifespan of 90 seconds (lines 6 and 10). The CURL metadata contains the custom random number generator (line 9). The CURL itself is the return value (line 10).

The server itself is just a skeleton that expects messages whose only content is a "reply to" CURL *r*. Recall that every MOTILE/island message is accompanied by the CURL *u* to which the message is directed. On receiving such a message the server

extracts the service implementation (as a closure f) from the metadata of CURL u, evaluates f, and transmits that result via CURL r.

```
(define (random/new low high)                                          1
  (let ((difference (- high low)))                                     2
    (lambda () (+ low (* (random) difference)))))                      3
                                                                       4
(define (service/custom low high)                                      5
  (let ((custom (resource/new (resource/root) 100 7.5 90.0))           6
        (path (list "random" "custom" low high))                       7
        (metadata                                                      8
         (list (cons "implementation" (random/new low high)))))        9
    (curl/new custom path metadata)))                                  10
```

Fig. 5 Generating a client-specific service as a CURL.

Non-Delegation and CURL Revocation: COASTCAST [26] is a COAST-based service for the distribution and manipulation of real-time High Definition (HD) video. Islands whose assets include HD cameras and execution sites containing primitives for managing cameras and encoding (compressing) video serve video streams to other islands with high-resolution monitors for displaying the video streams. Island assets may be less tangible, for example, islands with sufficient computing capacity and network bandwidth to relay high-bandwidth video streams to other less capable islands. In such applications camera islands may want to restrict direct access to cameras to a small set of trusted display or relay islands. In other words, if island I holds a CURL u granting it access to a particular camera of island J we would like to guarantee that only I may access the camera of J even if it hands CURL u on to island X for its use. This property, non-delegation, is enforced by embedding J-generated restrictions (as metadata) in the CURL u that J provides to I. As all islands are self-certifying, island J can determine authoritatively if a message m sent to it via CURL u was sent from island I or some other island X. Each CURL u may contain (as metadata) a predicate (a single-argument closure) that, given the address of an island, returns true if the island is permitted to use u and false otherwise.

Figure 6 illustrates the construction of a CURL by island J that limits delegation on the basis of processor load. Islands A, B, and C are each permitted access to HD camera 3 of island J with a resolution of 720p at a frame rate of 20 frames-per-second (indicated by the `path`, line 14). Combined, the three islands may access the camera a total of 10 times (line 1), at most twice per day (lines 2–3), over a period of 14 days (lines 4–5). Lines 6–7 define a derived resource account `camera` that enforces these use, rate, and lifespan constraints.

Lines 8–13 define the delegation predicate dictated by island J. The access of all three islands A, B, and C is determined by the current processor load on island J. A may access the camera only if the processor load is low (≤ 3.7), B may access the camera only if the processor load is moderate at worst (≤ 7.3), and C may access

```
(let* ((uses 10)                                                    1
       (rate                                                        2
        (/ 2.0 (timespan/seconds 1 0 0 0))) ; Twice per day.        3
       (lifespan                                                    4
        (timespan/seconds 14 0 0 0)) ; Fourteen days.               5
       (camera                                                      6
        (resource/new (resource/root) uses rate lifespan))          7
       (delegate                                                    8
        (lambda (x)                                                 9
          (or                                                       10
           (and (eq? x A) (<= (cpu/load) 3.7))                      11
           (and (eq? x B) (<= (cpu/load) 7.3))                      12
           (and (eq? x C) (<= (cpu/load) 10.1)))))                  13
       (path (list "camera" 3 "720p" 20))                           14
       (metadata (list (cons "delegate" delegate)))                 15
       (J@ (curl/new camera path metadata)))                        16
                                                                    17
  (curl/send A@ J@)                                                 18
  (curl/send B@ J@)                                                 19
  (curl/send C@ J@))                                                20
```

Fig. 6 Generate a CURL that limits delegation on the basis of processor load.

the camera only if the processor load is not excessively high (≤ 10.1). The CURL `J@` for access to camera 3 is generated at line 16 and is distributed to islands *A*, *B*, and *C* at lines 18–20.

When a closure f is sent to the execution site of camera 3 of island *J* via CURL `J@` island *J* applies the delegation predicate in the CURL metadata to the address of the transmitting island. If the predicate returns true then closure f is evaluated in the context of the execution site of camera 3; otherwise, closure f is rejected. Consequently, no other islands besides *A*, *B*, or *C* can access the camera and the access of these three is predicated on the current processor load. If another island *X* somehow acquires CURL `J@` it cannot be used productively by *X*.

Any CURL issued by an island may be revoked at any time by that island. The unit of revocation is the resource account *r* on which the CURL draws; for example, the CURL `J@` generated at line 16 of Figure 6 draws upon the resource account `camera` constructed at lines 6–7. If the resource account *r* upon which a CURL *u* draws is invalidated by its issuing island then any message transmission via *u* will be summarily rejected from that point forward. If multiple, distinct CURLS $u_1, \ldots, u_n$ draw upon *r* then the invalidation of *r* revokes all such CURLs u_i.

5 Motile/Island: A Reference Infrastructure

The COAST style imposes substantive constraints on how COAST-based applications must be built. Satisfying these constraints with a typical imperative programming

language is awkward so we have created an implementation platform for constructing and deploying COAST applications: MOTILE, a mobile code language whose semantics and implementation enforce key constraints on the use and migration of capability, and ISLAND, an infrastructure for MOTILE computations.

COAST Computations as Actors: Each COAST computation is implemented as an actor [3]. Each actor is an independent thread of computation that may transmit asynchronous messages to other actors, receive asynchronous messages from other actors, conduct private computations, and spawn new actors. All four actions are implemented (and perhaps selectively restricted) by functions in binding environments. Spawning is implemented as a specialized kind of message sending. The assumptions of the actor model, private computation and asynchronous messaging, match those of COAST, where private computation is conducted only in the context of a specific execution site. Actors are distinct from agents as, unlike agents [9], each actor is immobile (closures and continuations are mobile but not an actor). Also in many agent systems the identity of the agent is invariant as it moves from host to host, whereas spawning a closure or continuation results in a new and distinct actor.

Motile: MOTILE is a single-assignment, functional language for defining COAST computations. All MOTILE actors are named by one or more CURLs, a base data type in MOTILE. All MOTILE data structures are purely functional [41] (hence immutable). This choice reduces the semantic distinctions between messaging where sender and receiver share an address space and messaging where the sender and receiver occupy separate address spaces. Since all data structures (including messages) are immutable the data synchronization races common to shared-memory, imperative languages are not possible. By implication, shared-memory attacks where values are mutated after being shared with other actors are impossible.

Island: An *island* is a single, homogeneous address space occupied by one or more MOTILE actors. Islands implement the role of "execution host" discussed in Section 3. Each island is uniquely identified by a triple: the public key half of a public/private key pair, a DNS name, and an IP port number. All islands are self-certifying [35, 54] and all communication between islands is encrypted. Each island is instantiated with an initial set of execution engines, binding environments, and a set of trusted computations that allocate execution sites to visiting computations. Those trusted actors have access to implementation-level MOTILE primitives that other computations are not permitted to call; for example, creating an actor, instantiating island-wide user interfaces, and staging fixed island assets. These trusted actors also issue CURLs naming themselves, with the distinction that their CURLs are *durable*—valid even after an island is restarted. Computations holding a CURL for a trusted actor t are permitted to send a closure to t to spawn a new COAST computation. The specific execution engine and binding environment allocated to that new computation conform to the security and usage policy enforced by t.

Capability URL Implementation: Every CURL u denotes a specific computation x and contains:

- An *address*, the public key, DNS name, and IP port number of island I
- A *path* (a list of MOTILE values, possibly empty), defining for x the domain of interpretation of a message sent via u

- The *resource key*, a globally unique cryptographic identifier [33], used by island I as an index to CURL-specific, island-side state (including CURL timestamps, use count, and rate limit)
- The *creation* and *expiration* timestamps of u. After the deadline (the expiration timestamp) any message sent via this CURL will be rejected
- A *use count*, a positive integer, giving the nominal maximum number of messages that may be delivered to x via u
- A *rate limit*, a positive number, giving the nominal maximum rate (in Hz) at which messages transmitted via u to x will be delivered to x
- Arbitrary metadata that may include primitive values, standard structures such as lists or vectors, other CURLS, closures, continuations, and binding environments
- A cryptographic signature (over the contents of u) generated by the island I on which computation x resides. The signing, based on the private key of island I, allows any computation holding CURL u to verify that u is a valid CURL for x on I

A CURL supports, by construction, four base restrictions:

- Use count (total number of messages per CURL)
- Expiration date (after which the CURL is invalid)
- Rate limits (rate of message transmissions per CURL)
- Revocation (permanently withdraw, per CURL, the capability to communicate).

All are enforced by the issuing island I, since no island would reasonably trust another to enforce its own restrictions, and all four restrictions require the issuing island to maintain a small amount of state. Let u be a CURL for actor x. A trusted actor of I inspects each CURL/message pair u/m on arrival, passing the pair onto actor x if and only if CURL u is valid and the pair satisfy all I-imposed restrictions. At CURL generation time, x and I may both insert arbitrary MOTILE expressions into u in addition to customizing the base restrictions listed above. In this manner x enforces x-specific, u-specific restrictions on communication including complex temporal constraints ("only on alternate Thursdays before noon"), use scenarios ("only legal expressions in a domain-specific language"), limits on delegation (only messages from island J) and conditionals based on observables ("the price of gold on NYMEX must be $<$ \$1657 per ounce"), and I in turn enforces restrictions it places on x and its collaborators. Each CURL contains the mobile code and static state that x will require to enforce those additional observable restrictions.

6 Related Work

COAST and MOTILE/ISLAND have been influenced by prior work on mobile code including remote evaluation [48, 49], Scheme-based mobile code languages [27, 42, 51], the actor-like language Erlang [5], the object-capability language E [36], and capability-based operating systems [44, 46]. Island self-certification is drawn from self-certifying file systems [35] and URLs [31]. Our previous work on CREST

[14, 17, 18, 25] inspired computation exchange and led us to consider the problem of secure decentralized services that COAST addresses.

The idiom of computation exchange is partially reflected in Emerald [30], a system devoted to high-performance object mobility. Like computation exchange, Emerald emphasizes fine-grain state and code transfers among hosts, but assumes a single sphere of authority, identical host processors, and extends no further than a local area network.

Kali Scheme [10] implemented distinct address spaces containing multiple threads (the equivalent of islands) as a language construction and introduced closure and continuation exchange in messages as a mechanism for spawning threads in remote address spaces.

Self-protective behavior for the sake of ensuring progress (liveness) and system integrity is a vital interest of local security. Resource sandboxing is a common defense mechanism to forestall denial of service attacks via resource exhaustion and is available in several languages including Java [34] and Racket [53]. Execution sandboxing denies executing programs unsafe access to critical resources. The Google Native Client [55] employs software fault isolation [52] to confine the execution of untrusted native Intel x86 code. Extensions to Native Client [4] adapt these techniques to the complex run-times of high-performance, dynamic, JIT-enhanced languages such as JavaScript.

Several mechanisms were employed by Telescript [23], an object-oriented, mobile agent system, for which security was a concern [50]. Mobile Telescript agents were executed by a host-independent virtual machine within *places*, virtual locations devoted to a particular service: for example, a ticket purchase, or catalog search. Mobile agents and places were tagged with a designation of authority (the originating organization). Agents were granted *permits* by the managing authority of the place, which confined the capabilities granted to an agent and set resource caps. Telescript can be regarded as a mobile-code-based decentralized SOA.

Agent technology draws from both distributed systems and programming languages, notably for strong mobility. For example, Agent Tcl [32] (now DAgents) had four principal goals: ease of agent migration, transparent communication among agents, support for multiple agent languages, and effective security. Agent Tcl implements "whole" agent mobility where the only unit of code mobility is the entire binary image of the agent and relies on Safe Tcl to confine the executing Tcl agents where a set of trusted scripts provide limited access (based on access control lists) to unsafe functionality.

Object capability security is a pivotal influence on COAST. A capability [13], fuses access to, and designation of, a protected resource into a single, unforgeable reference. The object capability security model [36] implements confinement [46], revocation, and multilevel security [37]; offers patterns for non-delegation [39]; resolves the problem of the Confused Deputy [12, 28]; and is a base mechanism for information flow control [6, 38]. The Emerald language [43] is an early example of an object-capability language.

CURLs have precedent in the self-certifying URLs (YURLS) of Waterken [11], the unique URLs of Second Life [2], and the time-limited, signed URLs of Ama-

zon S3 [1]. YURLs embrace "communication by introduction" in which a client, interacting with a trusted partner, is granted the capability to communicate with a specialized service acting on behalf of (or equivalently, as a proxy for) the trusted partner. Both YURLs and CURLs contain one or more large, cryptographic numbers, in the former the SHA hash of the public key of a web site and in the latter, the public key of an island and the resource key of the resources account affiliated with the CURL. Consequently, both YURLs and CURLs are impossible to guess but YURLs can be forged as they are not signed. In contrast, since CURLs are signed with the private key of the issuer they cannot be forged, are tamper-proof, and non-repudiable.

7 Conclusion and Future Work

Since decentralized services, by definition, have no so single defensible perimeter, all of the constituent services must be self-defensive. Capability security is the principal defensive mechanism for COAST-based systems and takes two forms: functional capability, circumscribed by the execution engine and binding environment of the individual execution sites of computations, and communications capability, where communication by introduction and Capability URLs limit and shape the ability of computations to inter-communicate. By design CURLs prevent arbitrary communication among service components and, by constraining communication, reduce the risks and consequences of both accidental errors and malicious attacks. Communication between computations x and y is possible only if at least one of the two holds a CURL for the other. However, that is the minimum necessary condition since any messaging between the two must also satisfy a CURL-specific use cap (the total number of messages that may be sent), rate limit (the frequency in Hertz at which messages may be sent) and an expiration deadline (the "end of life" for the CURL). With these constraints a computation can regulate the total number of messages that it receives from another, the arrival rate of those messages, and the span of time over which it can expect to hear from another computation—all of which can thwart or reduce abuse of service and ensure fair service for others.

These basic constraints are useful but insufficient for enforcing service agreements based on *observables*, real-world phenomena (weather, processor load, stock prices, ...), the states of the communicating computations, or the states of computations elsewhere. CURLs, when combined with embedded MOTILE mobile code, facilitate:

- Preconditions and use restrictions incorporating observables
- Explicit state transfer in the spirit of REST
- Service customization
- Service transfer for which both the state of the service and its implementation are completely explicit
- Non-delegation that incorporates arbitrary temporal and use constraints

The combination of communication by introduction and mobile code is a significant contribution to the safety and security of decentralized services.

Mobile code embedded in CURLs can serve other functions as well including logging, message tracing, debugging, exception handling, event distribution, traffic analysis, checkpointing and service restart. Many interesting research questions remain for example, domain-specific security languages or service-level contracts as embedded mobile code in CURLs, language constructions for incorporating, and responding to, resource restrictions in CURLs, hierarchical constraints in CURLs that reflect layered, system-level concerns, the roles of CURLs with embedded mobile code in dynamic software update, and COAST-like communication by introduction for embedded and soft real-time systems.

Acknowledgements We are indebted to Kyle Strasser whose adroit implementation of COASTCAST broadened our understanding of communication capability in MOTILE/ISLAND and the means by which functional capability could be manipulated to support security.

This work supported by the United States National Science Foundation under Grant Nos. CCF-0917129 and CCF-0820222.

References

1. http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAuthentication.html, March 2006.
2. http://wiki.secondlife.com/wiki/Protocol#Capabilities, February 2011.
3. AGHA, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, December 1986.
4. ANSEL, J., MARCHENKO, P., ET AL. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 2011 Conference on Programming Language Design and Implementation* (New York, New York, USA, June 2011), PLDI'11, ACM.
5. ARMSTRONG, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
6. BIRGISSON, A., RUSSO, A., AND SABELFELD, A. Capabilities for information flow. In *Proceedings of the Conference on Programming Languages and Security* (New York, New York, USA, June 2011), PLAS'11, ACM.
7. BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39–59.
8. BOMBERGER, A. C., FRANTZ, W. S., HARDY, A. C., HARDY, N., LANDAU, C. R., AND SHAPIRO, J. S. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures* (1992), USENIX Association, pp. 95–112.
9. BRAUN, P., AND ROSSAK, W. R. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann, 2004.
10. CEJTIN, H., JAGANNATHAN, S., AND KELSEY, R. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems 17*, 5 (1995), 704–739.
11. CLOSE, T. Decentralized identification. http://www.waterken.com/dev/YURL/, 2001.
12. CLOSE, T. ACL's don't. Tech. Rep. HPL-2009-20, HP Laboratories, February 2009.
13. DENNIS, J. B., AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Communications of the ACM 9*, 3 (March 1966), 143–155.
14. ERENKRANTZ, J. R. *Computational REST: A New Model for Decentralized, Internet-Scale Applications*. PhD thesis, University of California, Irvine, September 2009.

15. ERENKRANTZ, J. R., GORLICK, M., SURYANARAYANA, G., AND TAYLOR, R. N. Harmonizing architectural dissonance in REST-based architectures. Tech. Rep. UCI-ISR-06-18, Institute for Software Research, University of California, Irvine, December 2006.
16. ERENKRANTZ, J. R., GORLICK, M., AND TAYLOR, R. N. CREST: A new model for decentralized, internet-scale applications. Tech. Rep. UCI-ISR-09-4, UCI Insitute for Software Research, September 2009.
17. ERENKRANTZ, J. R., GORLICK, M. M., SURYANARAYANA, G., AND TAYLOR, R. N. From representations to computations: The evolution of web architectures. In *Symposium on the Foundations of Software Engineering* (September 2007), pp. 255–264.
18. ERENKRANTZ, J. R., GORLICK, M. M., AND TAYLOR, R. N. Rethinking web services from first principles. In *Proceedings of the 2nd International Conference on Design Science Research in Information Systems and Technology* (Pasadena, California, May 2007).
19. ERL, T. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice-Hall, 2005.
20. FELLEISEN, M. The theory and practice of first-class prompts. In *Proceedings of the Symposium on Principles of Programming Languages* (New York, New York, USA, January 1988), ACM, pp. 180–190.
21. FIELDING, R. T., AND TAYLOR, R. N. Principled design of the modern web architecture. *ACM Transactions on Internet Technology 2*, 2 (May 2002), 115–150.
22. FUGGETTA, A., PICCO, G. P., AND VIGNA, G. Understanding Code Mobility. *IEEE Transactions on Software Engineering 24*, 5 (1998), 342–361.
23. GENERAL MAGIC INC. *Telescript Language Reference*. Sunnyvale, California, USA, October 1995.
24. GORLICK, M. M., GASSTER, S. D., PENG, G. S., AND MCATEE, M. Flow webs: Architecture and mechanism for sensor webs. In *Proceedings of the Ground Systems Architecture Workshop* (Manhattan Beach, California, USA, March 26–29 2007).
25. GORLICK, M. M., STRASSER, K., BAQUERO, A., AND TAYLOR, R. N. CREST: principled foundations for decentralized systems. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion* (October, 2011), SPLASH'11, ACM, pp. 193–194.
26. GORLICK, M. M., STRASSER, K., AND TAYLOR, R. N. COAST: An architectural style for decentralized on-demand tailored services. In *Proceedings of 2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture* (August 2012), WICSA/ECSA'12, pp. 71–80.
27. HALLS, D. A. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, June 1997.
28. HARDY, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review 22*, 4 (1988), 36–38.
29. JAGANNATHAN, S. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994), 456–492.
30. JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.
31. KAMINSKY, M., AND BANKS, E. SFS-HTTP: Securing the web with self-certifying URLs. Tech. rep., MIT Laboratory for Computer Science, 1999.
32. KOTZ, D., GRAY, R., NOG, S., RUS, D., CHAWLA, S., AND CYBENKO, G. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing 1*, 4 (July 1997), 58–67.
33. LEACH, P., MEALLING, M., AND SALZ, R. A universally unique identifier (UUID) URN namespace. RFC 4122, July 2005.
34. LINDHOLM, T., AND YELLIN, F. *Java Virtual Machine Specification*, 2nd ed. Prentice-Hall, April 1999.
35. MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, South Carolina, USA, 1999), ACM Press, pp. 124–139.

36. MILLER, M. S. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.
37. MILLER, M. S., AND SHAPIRO, J. S. Paradigm regained: Abstraction mechanisms for access control. In *Eighth Asian Computing Science Conference* (December 2003), ASIAN'03, Springer-Verlag, pp. 224–242. Available as http://www.hpl.hp.com/techreports/2003/HPL-2003-222.pdf.
38. MURRAY, T. *Analysing the Security Properties of Object-Capability Patterns*. PhD thesis, Hertford College, University of Oxford, Oxford, UK, 2010.
39. MURRAY, T., AND GROVE, D. Non-delegatable authorities in capability systems. *Journal of Computer Security 16*, 6 (December 2008), 743–759.
40. NELSON, B. J. *Remote procedure call*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1981.
41. OKASAKI, C. *Purely Functional Data Structures*. Cambridge University Press, 1998.
42. PIÉRARD, A., AND FEELEY, M. Towards a portable and mobile Scheme interpreter. In *Proceedings of the Scheme and Functional Programming Workshop* (September 2007), pp. 59–68.
43. RAJ, R. K., TEMPERO, E. D., LEVY, H. M., BLACK, A. P., HUTCHINSON, N. C., AND JUL, E. Emerald: A general-purpose programming language. *Software - Practice and Experience 21*, 1 (1991), 91–118.
44. REES, J. A. *A Security Kernel Based on the Lambda Calculus*. PhD thesis, Massachusetts Institute of Technology, 1996.
45. SALTZER, J. H. Protection and the control of information sharing in Multics. *Communications of the ACM 17*, 7 (1974), 388–402.
46. SHAPIRO, J. S. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Philadelphia, Pennsylvania, 1999.
47. STAMOS, J. W. *Remote evaluation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1986. (Also available as MIT Technical Report MIT/LCS/TR-354, MIT, Cambridge, Massachusetts, 1986).
48. STAMOS, J. W., AND GIFFORD, D. K. Implementing remote evaluation. *IEEE Transactions on Software Engineering 16*, 7 (July 1990), 710–722.
49. STAMOS, J. W., AND GIFFORD, D. K. Remote evaluation. *ACM Transactions on Programming Languages and Systems 12*, 4 (1990), 537–564.
50. TARDO, J., AND VALENTE, L. Mobile Agent Security and Telescript. In *Proceedings of the 41st IEEE International Computer Conference* (Washington, DC, USA, 1996), COMPCON'96, IEEE Computer Society, pp. 58–63.
51. VYZOVITIS, D., AND LIPPMAN, A. MAST: A dynamic language for programmable networks. Tech. rep., MIT Media Laboratory, May 2002.
52. WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, New York, USA, December 1993), ACM, pp. 203–216.
53. WICK, A., AND FLATT, M. Memory accounting without partitions. In *Proceedings of the 4th international symposium on Memory management* (New York, New York, USA, 2004), ACM, pp. 120–130.
54. WOLFE, M. SCURL authentication: A decentralized approach to entity authentication. Master's thesis, University of California Irvine, October 2011. Available as http://gradworks.umi.com/15/02/1502427.html.
55. YEE, B., SEHR, D., DARDYK, G., CHEN, B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy* (2009).
56. YUMEREFENDI, A. R., AND CHASE, J. S. The role of accountability in dependable distributed systems. In *Proceedings of the First conference on Hot topics in system dependability* (Berkeley, CA, USA, June 2005), HotDep'05, USENIX Association.