

Chapter 2

Communication and Capability URLs in COAST-based Decentralized Services

Michael M. Gorlick and Richard N. Taylor

2.1 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 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 such varied domains as 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, but joining or leaving a supply-chain system should not be undertaken lightly.

All decentralized systems are intrinsically dynamic: members join and leave, service relationships change, system implementations and deployments vary (as do 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 is immune to malicious behavior.

*The bibliography that accompanies this chapter appears at the end of this volume and is also available as a free download as Back Matter on SpringerLink, with online reference linking.

M. M. Gorlick (✉) · R. N. Taylor
University of California, Irvine, CA 92697-3455, USA
e-mail: mgorlick@acm.org

R. N. Taylor
e-mail: taylor@ics.uci.edu

Computational State Transfer (COAST), an architectural style based on the idiom of *computation exchange*, targets decentralized systems and their associated 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 [81] 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 goals 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 containing keys to session state held server-side, in violation of REST precepts, demonstrated. In contrast to cookies, web continuations [132] preserve stateless interactions. The emergence of Ajax (mashups) and the exploitation of computation in the browser suggested a more prominent role for code mobility [89], namely constructing and deploying customizations and application services [65]. 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 [65–67], 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 [66, 69] and Firewatch [95], 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 [97]. Here our focus is communication security, whereby COAST hosts manage communications among computations and modulate access to critical services. However, before introducing these 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.

2.2 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 [32, 164], remote evaluation [219–221], REST [81], and service-oriented architectures [70]. 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 [89] 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 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 defensible security perimeter when function is integrated across the multiple, separate trust domains [257]. 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.

2.3 The COAST Architectural Style

COMputAtional State Transfer (COAST) is an architectural style for decentralized and adaptive systems [97]. 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 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) [202] and *capability-based security* [34]. 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 [74]), and binding environments (maps of name/value pairs [113])
- 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. In general, Carol can monitor Dave’s computations executing on her infrastructure or her own computations executing elsewhere using automated “report-back” CURLs, selectively wrapped closures, monitoring functions provided at the service execution site, publish/subscribe events originated by a service provider, or third-party monitoring.

COAST offers two distinct forms of capability, (1) functional capability—what a computation may do, and (2) 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 execution 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 computation 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 . The CURL u specifies which execution site $\langle E, B \rangle$ of x is to be used for the interpretation of message m where binding environment B strictly confines the functional capability granted to mobile code contained in the incoming message.

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: (1) contained in the closure defining the computation; (2) returned as values by functions invoked; or (3) embedded as values in the messages received.

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. Fig. 2.1 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 [173] (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

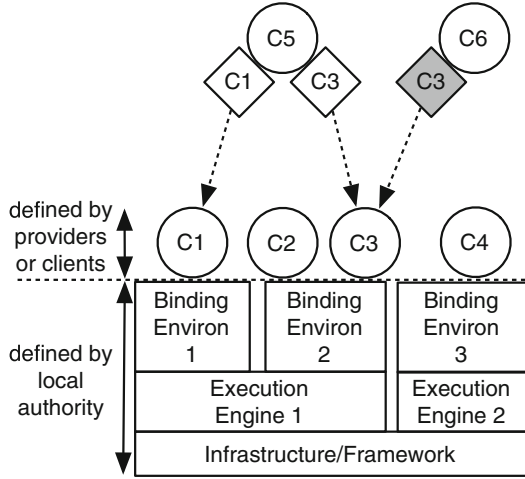


Fig. 2.1 The notional structure of a COAST execution host where a trusted code base allocates execution engines and binding environments to computations, whose implementations are sourced from a variety of other organizations. Distinct functional capability is held by each of the three individual binding environ(ment)s shown here. Computations (shown as *circles*) C1–C4 have been deployed by service providers and clients. Computations C5 and C6 each hold a distinct CURL (shown 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

computations with which 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 directly with computation y unless it already holds or obtains (via function call or messaging) a CURL issued by y .

2.4 Capability URLs in Detail

Each CURL u denotes a specific computation x and contains a self-certifying network address [115], 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

```

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

```

Fig. 2.2 A simple MOTILE program. Island *I* sends a closure to island *Guide* for execution that does nothing but transmit the number 42 back to island *I*

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.

A Simple Motile Program Fig. 2.2 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 5 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 5) 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 (lines 2–4).

The function *curl/send* given in lines 6–8 transmits a spawn command (the second argument, line 8) to the computation denoted by the first argument, a CURL for island *Guide* bound to the variable *Guide@* (line 7; the details of how computation x acquired CURL *Guide@* are omitted for the sake of brevity and clarity). The closure, $(\text{lambda } \dots)$, evaluated by an execution site of island *Guide*, immediately transmits


```

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

```

Fig. 2.3 A computation on island *I* dispatches a closure to island *J* to obtain all of the palindromes in a database of words

the message 42 back to computation *x* on island *I* via `CURL Guide@`. The `MOTILE` function `receive`, called on line 9 of computation *x*, blocks until a message *m* for computation *x* arrives and returns that message *m* as its value.

A Client-Defined Service Fig. 2.3 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 s. 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 14–16 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`. `words/filter`, a domain-specific function, is resolved in the binding environment of the closure’s execution site on island *J*. It takes a predicate *f* as its argument, traverses the word database applying *f* to each word *w*, and returns a list.

Line 17 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 18, the computation on island *I* waits (a maximum of 60 s, the lifespan of the promise) 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 18.

```

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

```

Fig. 2.4 Generating a time-limited, day-specific sales coupon as a CURL

The program of Fig. 2.3 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 database 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 Fig. 2.4. 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 lines 1–4 *sale* is granted a balance of three total uses, a rate limit of once every 17 s, 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). CURLs are intended for use by computations, not people; however, they can be serialized as text for storage in files or out of band transmissions such as email.

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.

REST: Advanced Research Topics and Practical Applications

Pautasso, C.; Wilde, E.; Alarcon, R. (Eds.)

2014, IX, 222 p. 58 illus., 25 illus. in color., Hardcover

ISBN: 978-1-4614-9298-6