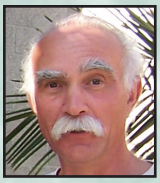
ISR
Institute for Software Research
University of California, Irvine

COAST: Architectures for Decentralized On-Demand Tailored Services



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

Kyle Strasser
University of California, Irvine
kstrasse@uci.edu

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

April 10, 2012
ISR Technical Report # UCI-ISR-12-2

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

COAST: Architectures for Decentralized On-Demand Tailored Services

Michael Gorlick Kyle Strasser Richard N. Taylor

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

Abstract

Decentralized systems are systems-of-systems whose services are governed by two or more separate organizations under distinct spheres of authority. Coordinated evolution among 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. To this end we contribute an architectural style, COmputAtional State Transfer (COAST), designed to provide extensive, but safe and secure, client-directed customization of decentralized services. COAST combines mechanisms from software architecture, cryptography, security, programming languages, and systems, granting application architects flexible provisioning of their core services and assets while, at the same time, protecting those services and assets from attack and misuse.

Keywords software architecture, decentralized systems, service-oriented architectures, client-driven adaptation.

1. Introduction

Architectures of future decentralized service ecosystems, systems-of-systems whose component systems are governed by multiple distinct organizations, pose significant challenges. Irrespective of architecture, all decentralized services must come to terms with: the unpredictable granularity of service interfaces and communication; recursively structured services that themselves rely upon multiple other services outside of their own organizational boundary; unpredictable semantic demands that belie assumptions of uniform interfaces; multiple implementation languages and environments; and a broad span of trust relationships that change over time. Decentralization exacts an ongoing toll as a service-consuming agency now exposes itself to the whims of another as services come and go, as the performance of decentralized resources fluctuates, or as application interfaces evolve. A service-providing agency must expose (to some degree) valuable assets to productively participate in decentralized ecosystems, yet access to those assets must be mediated and is rarely unconditional.

In this environment efforts on the part of service providers to vary, evolve and tailor content-based service interfaces and semantics in favor of particular service consumers simply will not scale. Previous work, from the point of view of mobile code [11, 26, 27], suggests an alternative: simply allow service consumers to program service providers. This approach introduces a division of labor between service providers and service consumers; providers create fine-grained and sufficiently general interfaces for their services and assets, and consumers orchestrate those fine-grained interfaces to produce exactly the services they require.

In this light, how should we design architectures for dynamic and tailored services in decentralized environments, specifically, *feasible* on-demand, customized, decentralized services? Our contributions are:

- A summary of *computation exchange*, a novel mode of interaction among Internet-scale service providers and consumers that enables tailored service construction and adaptivity (Section 2). Though we first sketched a view of computation exchange in prior work [6–8, 14] we describe here a more informed view of its architectural consequences, with particular emphasis on safety and security.
- *COmputAtional State Transfer* (COAST), an architectural style for computation exchange as the sole means for constructing, deploying and controlling services (Section 3). We devote special attention to the security mechanisms that we incorporated into the core elements of the style and the safety assurances that result.

- A mobile code language, MOTILE, and supporting infrastructure, ISLAND, that reflect the COAST style. When combined they induce security and flexibility for both service provider and service consumer (Section 4).
- A prototype application, COASTCAST, whose construction and performance suggests that MOTILE/ISLAND may be expressive (enough) and efficient (enough) for various decentralized service applications (Section 5).

2. Background: Computation Exchange

Computation exchange (the computational analogue of content exchange) is the bilateral exchange of computations among peers in a decentralized system containing multiple distinct spans of authority. In this regime content delivery is a byproduct of the evaluation of computations exchanged among peers. Our focus is *decentralized SOAs*, whose individual constituent services (themselves SOAs) operate under distinct spans of authority. Decentralized service providers must meet two conflicting goals: the protection of valuable fixed assets (e.g., servers, databases, sensors, data streams, and algorithms) and meeting the evolving service demands of a diverse client population. Computation exchange resolves that conflict, *but at a price*, imposing significant obligations on architectures that implement it as a core technology for tailoring and adapting decentralized services.

Computation exchange engenders all of the risks associated with mobile code [11] including waste or theft 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. In this environment perimeter protection, while desirable, is insufficient since even 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 service functions leading to inadvertent loss of service.

Notwithstanding, the advantages of computation exchange are too tempting to ignore [26, 27]: *customize services per-client* since computations dispatched by service consumers to service providers may compose service primitives to obtain tailored service, *transfer computation to data sources* to reduce system overhead, query latency and unwanted results, *increase agility for service providers* as they need not (nor can not) anticipate every possible service demand, *ease service upgrade* for providers who may offer multiple versions of a service simultaneously, *shorten development and deployment cycles* for both service providers and service consumers, *monetize a greater variety of service products* as both providers and consumers may interchange roles, *increase service innovation* as the barrier to constructing and deploying services is lowered, *increase service redundancy* as the multiplicity of service offerings increases thereby reducing the impact of service outages, *enlarge opportunities for collaboration* among organizations at all scales, *rapidly amortize fixed assets* through asset-based services to new markets, *increase access* to high-value services that otherwise may be beyond the reach of a small organization, *enrich service compositions* to better match evoling client requirements, *improve adaptation* to ever changing business needs, and *expand the ecology of service* as the risks of computation exchange are reduced.

In general, for decentralized SOAs, 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 [32]. 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. Experience teaches that retrofitting security to existing systems is problematic at best, hinting that security should be a core *architectural* element.

3. The COAST Architectural Style

An *architectural style* is

> a named collection of architectural design decisions that are applicable in a given context (*computation exchange*), constrain architectural design decisions specific to a specific system (*decentralized SOAs*) within that context, and elict beneficial qualities (*safety and security, among others*) in each resulting system [29].

Any architectural style addressing computational exchange must confront the security risks inherent in exchanging and executing mobile code across multiple, distinct, spheres of authority where malicious mobile code is a near certainty. The mechanisms of the style must be resilient to attack, to the extent that resilience is technically feasible. Where not, aggrieved parties must turn to other avenues (economic, social, legal, or cultural) for remediation.

The COAST architectural style relies on two principles: the *Principle of Least Authority* (POLA) [24] and *capability-based security* [3]. POLA dictates that security is a cross-product of the authority offered to a principal (that is, the base functional power available to a principal) and the rights of the principal (that is, the rights of use conferred upon that principal with respect to the 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 the respective rights to a principal. Our insight is that these principles apply equally well to both mobile code and the design of architectural styles.

For the safety and integrity of the platform, the authority and rights granted to visiting mobile code must be minimized. For example, the mobile code might be allowed to read exactly one specific file (say a machine-readable specification for an electronic component) and no other. Each distinct sphere of authority may confer separate sphere-specific authority and rights to the same mobile code, reflecting the fundamental freedoms of agency that any decentralized architectural style must accommodate. An architectural style can constrain both where and when the delegation of authority and rights occurs and the mechanisms of delegation. COAST is but one architectural style for computation exchange just as "pipes and filters" is but one of many architectural styles for data processing. The constraints COAST imposes, drawn from capability-based security, mandate where, when and how authority and rights are conveyed. Given a particular instance of the style (a specific architecture), those constraints assist providers and consumers alike in precisely identifying both the points in an architecture where capability is conveyed and the consequences of that conveyance.

Not all security properties of interest can be induced by architecture—some point solutions are required. However, architecture can identify *which* solutions to employ and how to integrate those solutions with the base elements of an architectural style. Architectures for decentralized mobile code exchange benefit from supporting services such as a Public-Key Infrastructure (PKI) and/or a Web of Trust (WoT) for predicting the rough outlines of mobile code behavior (for example, can we expect the code to be "well behaved"). Similarly, mechanisms for monitoring and accountability [32] are used to determine if those trust expectations hold. Finally, capability architectures can make good use of services that modulate and restrict capability on-demand in response to abuses and attacks.

The COAST style dictates:

- All resources are computations whose sole means of interaction is the asynchronous exchange of closures (functions plus their lexical-scope bindings [5]), delimited continuations[1] and binding environments [16]
- 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, untamperable cryptographic structure that conveys the authority to communicate
- Computation x may deliver a closure, delimited continuation or binding environment to computation y if and only if x holds a valid CURL u_y of y
- The interpretation of a closure, delimited continuation or binding environment delivered to computation y via CURL u_y is u_y-dependent

Over its lifespan each COAST computation is confined to some *execution site* $\langle E, B\rangle$. Execution engines E may vary from site to site: for example, a Scheme interpreter or a JavaScript just-in-time compiler. **The execution engine E defines the semantics of evalution of the computation. The binding environment B contains all of the functions and global variables offered to the computation at that execution site.** A computation is the execution of a closure λ by the execution engine E of its execution site in the context of B.[2] Names unresolved within the lexical scope of λ (the **free variables** of λ) are resolved, at time of reference, within the binding environment B of the execution site—if B fails to resolve the name the computation is terminated.

Each binding in B is either a primitive value (integer, string, boolean ...), a closure, or recursively a data structure (list, vector, CURL, binding environment ...) whose constituent values are themselves primitives, closures, or data structures. The set of all values reachable from B (either directly or transitively) sets the initial floor for the *functional capability* of a closure λ executing at site $\langle E, B\rangle$—all the procedures the mobile code may call. All additional functions either defined by λ or received inter-island as mobile code by λ can *not* increase the functional capability of λ since in each case their free variables are resolved by B the source of functional capability in execution site $\langle E, B\rangle$. However, the transfer via intra-island messaging of closures γ from the binding environments B_i of other actors executing in execution sites $\langle E, B_i\rangle$ to λ can increase the functional capability of λ since it is trivial for a closure γ to capture in its lexical scope a function f bound in B_i. Thus, over its lifespan, the functional capability of a computation λ can increase over and above the floor set by B_i.[3]

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 fungible resources such as processor cycles, memory, storage, or network bandwidth; rate-throttling of same; logging; or adaptations for debugging. The contents of B may reflect both domain-specific semantics (for example,

[1] Delimited continuations [10, 25] reduce the risk of needlessly and accidentally capturing state that may prove a security risk, and simplify the construction of many powerful control structures. For these reasons they are preferable here to the classic Scheme continuations captured by call/cc [5].

[2] Delimited continuations are reified as closures, so, for the sake of brevity and convenience, we will conflate the two.

[3] The techniques required to prevent λ from increasing its functional capability are well beyond the scope of this paper. Suffice to say that these techniques, many of them refined applications of the base mechanisms already available to Motile/Island, can be employed to guarantee that λ *never* acquires additional functional capability beyond that initially granted to it in binding environment B.

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). In keeping with POLA, each B of an execution site should contain *just* the functional capability that visiting mobile code requires to implement a confined segment of service. For example, we may have $\langle E, B_1 \rangle$ and $\langle E, B_2 \rangle$, $B_1 \neq B_2$, where B_1 and B_2 offer access to two disjoint subsets of the tables of a single relational data base. B_1 is for accounting while B_2 is for data-mining market trends.

The capability to communicate rests solely with CURLs. CURLs are unguessable, unforgeable and tamper-proof as they contain one or more large, cryptographic-grade random numbers and are signed by the issuer. Every message m (closure, delimited continuation or binding environment) a computation y receives is accompanied by the specific CURL u_y used by the sending computation. The interpretation of m is u_y-dependent because the CURL, as a capability, conveys both the authority to communicate and the specific rights bound with that authority.

CURLs may contain mobile code as metadata (in the form of closures, delimited continuations and binding environments) as well as any mobile Motile data type (for reasons of security not all Motile data types are mobile). Moreover, as u_y is tamper-proof y may safely rely on the state (and mobile code) u_y contains. When constructing CURL u_y as a target for message transmissions, y can ensure that u_y contains all of the static state y will need in the future (including arbitrary y-generated closures) to validate, restrict, and act upon the messages sent via u_y. In this manner a CURL conveys both the authority to communicate and the specific rights bound with that authority.

A CURL u_y denotes an execution site $\eta = \langle E, B \rangle$ offered by a computation y. When actor x sends a closure, delimited continuation, or binding environment for evaluation to y via u_y we say x is *dereferencing* u_y. There may be $j > 1$ such CURLs $u_{(y,\eta,j)}$ where each conveys a distinct capability with respect to execution site η of y—for example, y only accepts messages sent using $u_{(y,\eta,e)}$ that pattern-match a certain format. Furthermore, y may capture its base binding environment B, draw binding environments $B_1, \ldots, B_m$ from B (using environment sculpting) and "deploy" multiple distinct execution sites $\langle E, B_0 \rangle, \ldots, \langle E, B_m \rangle$, where each $\langle E, B_i \rangle$ is denoted by one or more distinct CURLs. In general, computation y has three degrees of freedom for each y-specific $\langle E, B \rangle$: the execution engine E, the binding environment B, and the CURLs denoting the E/B pair. These degrees of freedom give y broad latitude in defining and customizing the services it offers to other computations.

Just as a computation y can acquire additional functional capability over its lifespan it can also accumulate communication capability in the form of additional CURLs. The only sources of communication capability are CURLs:

- Contained in the closure λ defining the execution of y
- Returned as values by functions invoked by y
- Embedded as values in the messages y receives.

For both functional and communication capabilities, the only two sources of capability (beyond any CURLs included in the defining closure of the computation) are the functions in B available to y and the contents of the messages that y receives.

The sphere of authority of a COAST execution host unilaterally dictates the base semantics and side-effects of all possible closures by crafting any number of individual execution sites $\langle E, B \rangle$. No local computation beyond that permitted by B is possible since, for any mobile code λ, B is the only resolver for the free variables of λ.

No computation can escape the resource restrictions of $\langle E, B \rangle$ since resource consumption is either capped by E, by the behavior of specific functions in B, or is simply impossible because B doesn't contain the requisite functions (it is impossible for computation λ to consume file storage if all of the functions for creating files or writing to them are missing from the binding environment of λ's execution site). Computations confine all possible interactions with other computations by issuing CURLs that provide exactly the state necessary to interpret a message. No communication between computations is possible except through those CURLs. In this respect, the style defines all the architectural points at which capability may be restricted or modulated for the safety and security of service providers.

Given these security properties, we conjecture that COAST is a feasible architectural style for constructing client-tailored services. Computations are both suppliers and consumers of service. Computations, acting as suppliers, can offer multiple execution sites for service consumers, with each site specialized by execution engine, binding environment, and referencing CURLs. Computations, acting as consumers, can dispatch closures for evalution to any execution site for which they hold a valid referencing CURL. A consumer's closure composes just the execution site functions it requires to implement exactly the tailored service it requires.

Given the primacy of binding environments and CURLs as the arbiters of capability, it is clear how to introduce the point solutions that complement the essentials of a COAST implementation. Trust relationships, computed with data from a PKI/WoT infrastructure, are important when accepting closures as new computations and selecting the $\langle E, B \rangle$ offered to each computation—including the limits placed on the fungible resources made available to the computation and the policies bounding the computation's activity. Any security service need only monitor execution site activity and communications (via CURLs) to minimize misuse of capability, detect an attack (whose telltale signs are misuse of capability), mitigate the damage of an attack (by revok-

ing capability), and blame the responsible party (with the assistance of a PKI/WoT).[4]

4. Island/Motile: A COAST Reference Implementation

Our implementation of the COAST style contains two components: MOTILE, a mobile code language whose semantics and implementation enforce key constraints on the use and migration of capability, and ISLAND, a peering infrastructure for MOTILE computations. We focus on the key decisions made while implementing MOTILE/ISLAND and the architectural characteristics induced by those decisions, omitting many of the implementation details of MOTILE/ISLAND. We discuss four key constructs: the representation of COAST computations as actors, the role of MOTILE in that representation, the role of Island in computation exchange and its contributing security provisions, and a more detailed examination of the semantics of CURLs.

Each COAST computation is implemented as an actor [1]. Each actor is an independent thread of computation whose actions are limited to transmitting asynchronous messages to other actors, receiving asynchronous messages from other actors, conducting private computation, and spawning new actors. Each of these four actions are implemented (and perhaps selectively restricted) by functions in binding environments. Message transmission, in addition, requires that the sending actor possess a CURL naming the target actor.

MOTILE is a Scheme-like [5] single-assignment, functional language augmented with the actor model. All MOTILE actors are named by one or more CURLs. All MOTILE data structures are purely functional [21] (hence immutable). This choice reduces the semantic distinctions between intra-island messaging (where all collocated actors share an address space) and inter-island 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.

An ***island*** is a single, homogeneous address space occupied by one or more MOTILE actors. 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. No two islands share the same public key; no two islands share the same DNS name/port number pair. All islands are self-certifying [19, 31]—each authenticates its identity to every other island with which it communicates—and all communication between islands is encrypted. Each island is instantiated with an initial set of execution engines and binding environments and a single distinguished *root* actor. The root actor then bootstraps the island, spawning additional on-island actors (from a trusted MOTILE code base) for island governance.

A *CURL* denotes a fixed actor x residing on a specific island I and conveys the capability to communicate with x on I.[5] CURLs are cryptographically signed by the issuing island and any attempt at tampering or forgery can be detected by any actor. Given these guarantees no actor x may obtain the capability to communicate directly with actor y unless it comes into possession of a CURL for y. CURLs are a basic, serializable MOTILE data type and have precedent in the capability-augmented URLs of Waterken[6], the unique URLs of Second Life[7], and the time-limited, signed URLs of Amazon S3[8]. The capability to communicate is central to COAST systems and a primary means by which capability can propagate among actors. Each CURL supports, by construction, four base restrictions:

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

All are enforced by the issuing island of the CURL: no island I would reasonably trust any other island to enforce I-specific restrictions, and all four restrictions require I to maintain a small amount of state.

An actor a may issue multiple independent and unique CURLs for itself, each representing a distinct communication capability. Every message m transmitted to a residing on island I is accompanied by an I-signed CURL u_a. A trusted I-island actor inspects each CURL/message pair u_a/m on arrival, passing the pair onto a if and only if CURL u_a is valid and the pair satisfy all I-imposed restrictions. At CURL generation time actor a may insert arbitrary MOTILE expressions, including procedures generated by a, into CURL U_a in addition to customizing the base restrictions listed above. In this manner a enforces a-specific, u_a-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 the NYMEX must be $< \$1657$ per ounce").

Each CURL u_a carries with it both the mobile code and static state information a requires to enforce the u_a-specific restrictions. Therefore a, in many instances, need not retain any state at all to enforce those restrictions. Consequently,

[4] Details of a PKI/WoT are out of scope for our current efforts. Section 4 contains a few suggestions for how a PKI/WoT might be used.

[5] Actors themselves are not mobile and the "identity" of each actor is distinct. Actor z is *spawned* when a closure is transmitted by an actor x to an execution site maintained by an actor y. z has an identity distinct from that of both x and y. Actors are *not* agents [4], neither computation exchange nor COAST are models for agents, and MOTILE/ISLAND is *not* an agent infrastructure.

[6] http://waterken.sourceforge.net

[7] http://wiki.secondlife.com/wiki/Protocol#Capabilities

[8] http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAuthentication.html

inter-actor communication can be *stateless* since the CURL can be used as the carrier of both the context and functions a requires to interpret and enact the message. Even in the worst case, a might retain only a modest amount of state.

MOTILE is implemented in Racket, a high-performance Scheme dialect, as a single-pass, MOTILE-to-Racket, closure compiler [9] whose "object code" is Racket closures. Closure transmission is the means by which MOTILE actors exchange computations; hence, the semantics imposed by the MOTILE compiler play a key role in the mechanics of closure exchange. The closures output by the MOTILE backend are in continuation-passing (Section 3.4 of [5]), lexical-stack-passing, binding-environment-passing style.[9] When a closure is sent from one actor to another (whether intra- or inter-island) the closure abandons its old binding environment B and is invoked with another binding environment C by the receiving actor. In other words, whenever a closure λ transfers from one actor to another it leaves all of the bindings of its free variables behind.

To transmit a MOTILE closure inter-island (and, recursively, any MOTILE data structure containing such, including binding environments) the sending island "decompiles" the closure into a directed (possibly cyclic) graph whose nodes are akin to the instructions of a high-level lambda calculus abstract machine, serializes the graph, and transmits the entire serialization as a byte string. The receiving island converts the byte string back into graph form and recompiles the abstract code into the closures of the host language (in our case, Racket closures).[10]

Only core MOTILE datatypes are serializable, though other host-specific data types may be introduced and accessed through binding environments and manipulated by MOTILE actors (for example, neither file handles nor database connections can be serialized). In the taxonomy of [11] MOTILE offers *weak* mobility. This choice is deliberate, as *strong* mobility [11] threatens the security of decentralized collaborations. Weak mobility prevents visiting mobile code from automatically transferring access to island fixed assets (for example, a database or sensor) off-island, where auditing and detecting misuse of the asset would be more difficult. Weak mobility implies that direct control of a high-value asset will never transfer to another sphere of authority, even inadvertently through error.

Serialization is necessary for inter-island computation exchange. However, for the sake of performance, intra-island exchange is simply by reference, since an island is a single, homogeneous address space and all MOTILE structures are immutable. However this efficiency has consequences: I-island actor x can share closures in its binding environment B_x with another I-resident actor y (in fact x can, under certain conditions, share the complete binding environment B_x of its execution site with y). Sharing functional capability in this fashion can be both useful—x may be a factory dispensing safely confined versions of powerful functions to other on-island actors—and dangerous if x transfer functions to y that y should never possess. While a full discussion of this issue and the preventative mechanisms required to constrain or forestall the intra-island "leakage" of functional capability is outside the scope of this paper the first line of defense is to prevent x from ever communicating with y in the first place, a task for which CURLs are designed.

Actors on a single island are arranged in a tree structure, with capability afforded narrowing from the root to the leaves of the tree. The root actor delegates capability and resources to trusted actors called *chieftains*. Each chieftain is responsible for the behavior of the actors who are members of its *clan* who themselves may be chieftains. The resources (processor cycles, memory, network bandwidth, file space and so on) granted to a chieftain are divided, according to clan policy, among the membership of the clan. This hierarchy of chieftains and actors is illustrated in Figure 1. Figure 2 illustrates a chieftain accepting a closure for a new actor in the context of a site $\langle E, B\rangle$ the chieftain selects.

The root actor is the chieftain of the distinguished clan, the *root* clan. The chieftain of each clan is responsible for creating all of the other actors within the clan and for monitoring their behavior. Actors created by a chieftain are *members* of the clan represented by the chieftain. In addition a chieftain $\mathcal{C}$ can create chieftains (and hence clans) $\mathcal{C}_1, \ldots, \mathcal{C}_m$ to each of which chieftain $\mathcal{C}$ allocates a portion of its resources. $\mathcal{C}$ is the *parent* of chieftains $\mathcal{C}_j$. In Figure 1 the root chieftain is the parent of chieftains $\mathcal{A}$ and $\mathcal{B}$ while chieftain $\mathcal{A}$ is the parent of chieftain $\mathcal{C}$. The root chieftain, and chieftains $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ are members of the root, $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ clans respectively. Every actor on an island (including chieftains) is the member of exactly one clan.

The capability granted to a chieftain is always equal to or strictly less than the capability of its parent. All actors, including chieftains, execute within the confines of a sandbox that bounds the resources they may consume. Any actor (chieftain) whose resource consumption exceeds its budget is suspended and its clan chieftain (parent chieftain) notified. The responsible chieftain terminates the errant actor and releases the resources it holds. If a chieftain is guilty of excessive consumption then the entire clan (including all derived clans) is terminated and then restarted. This policy staves off denial of service attacks (via resource exhaustion) from untrusted actors executing closures from malicious islands, prevents catastrophic island failure due to innocent errors in visiting mobile code, and constrains runaway resource con-

[9] Every closure generated by the backend accepts three extra arguments: a continuation, a lexical scope stack, and a binding environment. All references to free variables are resolved, at point of reference, relative to the current binding environment argument of the closure—usually the binding environment B of the execution site of the actor. These details are managed by the MOTILE runtime and are invisible to MOTILE programmers.

[10] As of April 2012 closure (and recursively binding environment) transmission is fully implemented; however, delimited continuation transmission is not.

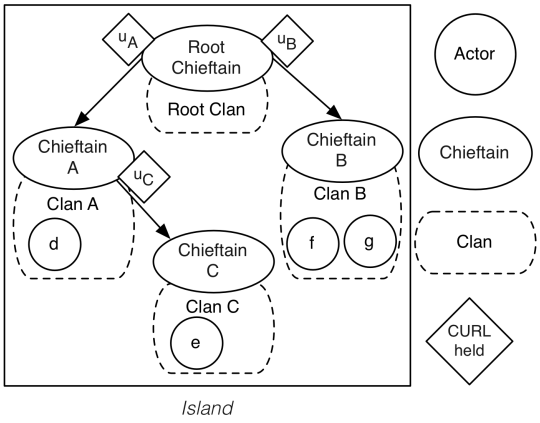


Figure 1. An arrangement of chieftains, clans and actors on an island. In this instance all chieftains hold CURLs naming their child chieftains.

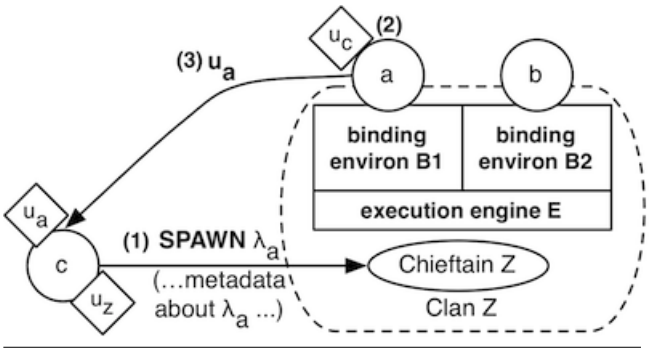


Figure 2. An simplified actor startup sequence for an actor a where a grants communication capability to its spawner c. (1) Some actor c submits a closure λ_a as a new spawn in clan Z. λ_a has a CURL u_c naming c embedded in it. (2) Chieftain Z allocates $\langle E, B_1 \rangle$ to the new actor a. (3) a sends a new CURL u_a back to c and c now has the capability to send to a. c is then free to pass u_a on to any other actor for which it holds a CURL.

sumption by an element of the trusted code base of the island. In all cases a log entry is generated for post analysis and debugging.

The propagation of capability across a single island is the starting point for analyzing the security and safety of COAST across multiple spheres of authority. One island can support hundreds of individual actors simultaneously and a single GB of main memory will accommodate 50 or more islands whose actor populations are on the order of 10^2–10^3. Given the modest memory footprint per island, architects may feasibly reduce the complexity of individual islands to improve island security and safety at the cost of increasing the total number of islands required for service provisioning.

Narrowing the binding environments of execution sites is a powerful tool for elimiminating needless capability. As a matter of design and best practice, chieftains practice *environment sculpting* to generate tailored, domain- and service-specific binding environments for their execution sites. MOTILE binding environments are persistent, functional structures [21], hence immutable and safely shared among any number of actors simultaneously, and the cost of deriving one binding environment from another (environment sculpting) is negligible. Per-binding customization supports tracing, security logging, error checking, code instrumentation, metering, and resource monitoring.

These implementation details induce system properties complementary to those induced by the COAST style itself. Here we summarize the most important of them.

The semantics of the medium of exchange (expressions in the MOTILE language) enforce a particular view of data and code mobility that is island-centric; that is, each island trusts only its own implementation. In this context, weak mobility serves an additional purpose, by preventing access to fixed assets (such as sensors or databases) beyond the immediate sphere of authority of the island. The island need only employ local actions (such as killing an actor) to halt any attack against local assets.

The serialization semantics of Motile separates the capabilities required for the successful execution of mobile code (the free identifiers that must be present in the binding environment of the destination's execution site) from the capabilities carried with it (any procedures or data in the lexical scope bindings of the closure). This separation allows islands to apply nonstandard interpretions to visiting mobile code for many purposes including: debugging, tracing, logging, dynamic analysis, profiling, trending, and policy enforcement.

COAST computations can customize the authority granted to anyone who communicates with them via restrictions on CURLs. Each actor specifies exactly the parameters of the communication it desires to respond to (to the extent that its requirements can be adequately expressed using the current restriction mechanisms – mechanisms are feasible). Since all CURLs are also revocable any attack conducted using an issued CURL may be halted once the attack is detected.

Chieftains and clans provide a framework for reasoning about security at each point in the architecture of an island as capability narrows as one descends the tree of clans. The mechanisms outlined here are flexible enough to accommodate wrappers, monitors, tracing and traffic analysis—all useful tools for evaluating behavior [32] and managing trust relationships with the help of a PKI/WoT.

5. Evaluation

We consider COAST performance and expressivity in the context of COASTCAST, a system for sharing and manipulating real-time High Definition (HD) decentralized video streams as *flows* [13]. A flow is a stream for which the

source, middle, and sink may each be independently manipulated as if the flow were a pliable "garden hose."

COASTCAST implements HD video flows, from camera to display, as collections of actors exchanging computations. A single flow may transit multiple spans of authority: the flow source (real-time HD video cameras), flow midpoints, and flow sink (displays of various sizes and resolutions) may reside on many distinct hosts each answering to a distinct and separate sphere of authority. Streaming and manipulating real-time video is a demanding application domain and in COASTCAST it serves merely as a convenient substitute for any high-bandwidth, soft real-time, data stream.

A single video flow contains, at a minimum, from source to sink ($\rightarrow$ indicates both the presence and direction of computation exchange):

$$\text{video encoder} \rightarrow \text{publish/subscribe relay} \rightarrow \text{video decoder} \rightarrow \text{display driver}$$

Video encoders and decoders are deployed (as actors spawned from MOTILE closures) to islands containing cameras and displays respectively. A video encoder, calling functions available in the binding environment of its execution site, is granted access to a camera on the island (an example of a fixed island asset). A video decoder, again through functions in the binding environment of the decoder's execution site, obtains a CURL granting communication capability to the island display driver (itself an actor executing a MOTILE closure). Publish/subscribe relays are deployed (as actors spawned from MOTILE closures) as transit points within the flow to simplify dynamic flow modification following flow establishment. The closures implementing all four roles are generated at runtime, each parameterized by the CURLs required to construct a complete flow. To replicate portions of a flow we merely dispatch those closures to other execution sites as needed.

Four distinct binding environments are used by islands implementing COASTCAST: camera access and video encoding, video decoding, display driver, and a standard binding environment allocated to actors that do not require domain-specific functions or special privileges that approximates the Scheme R5RS standard [18] in scope and size.

We explore the architectural expressivity of COAST in this domain by sketching the features possible with closure spawning, exchange and replication. COASTCAST users share individual flows by exploiting the mobility of MOTILE closures. To share a video flow with Alice, Bob instructs the video decoder actor d_1 responsible for the flow to spawn a copy of d_1 on Alice's island using the closure λ_d, forming the new and distinct video decoder actor d_2.[11] Re-establishment of the flow between publish/subscribe relay r, decoder d_2 and display driver happens automatically once λ_d is accepted as a new spawned actor d_2. Alice's island allows

[11] To do this Bob must have a CURL for a chieftain on Alice's island. How Bob came to possess this CURL is out of scope for this discussion.

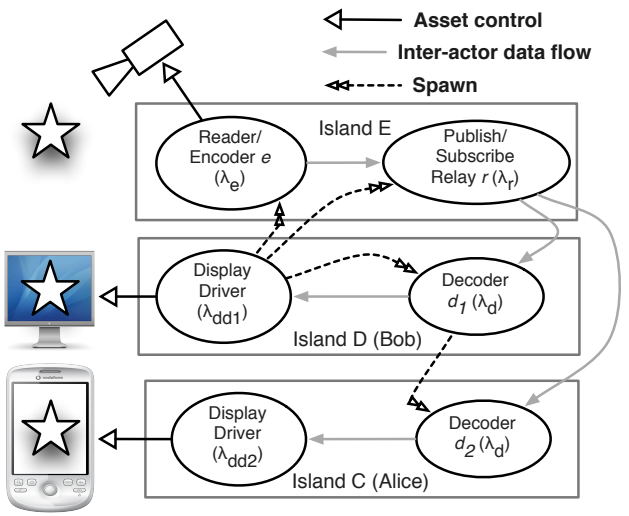


Figure 3. A COAST user constructing a complete flow by spawning a reader/encoder e, publish/subscribe relay r and decoder d (on islands E and D), and that user sharing the flow with another user by copying closure λ_d (to island C). Each island is a distinct sphere of authority. Clans and chieftains are ommitted for clarity.

d_2 to discover the location of her own display driver actor through a binding environment function. Since the CURL u_r (naming r) is embedded in the closure λ_d, d_2 can subscribe to r's flow using u_r immediately upon spawning. Multiple flows are shared by repeating the process for each decoder d_n. Figure 3 illustrates these features.

To explore flow composition as a consequence of closure generation and spawning, we implemented picture-in-picture (PIP). Two video flows, designated major flow f_M and minor flow f_m, are composited so that f_m's output is scaled and displayed in a subregion of the display of f_M. We implement PIP as a specialized form of video decoder, pip, that holds CURLs to the publish/subscribe relays for both f_M and f_m. λ_{pip}, the closure implementing pip, is generated after acquiring, at runtime, two CURLs identifying the relays r_1 and r_2 responsible for publishing f_M and f_m. Like d_1, pip can spawn λ_{pip} on other islands to replicate (share) its services. It is also capable of decomposing itself into two separate decoders d_3 and d_4, recovering the standalone views of the flows f_M and f_m. pip creates d_3 and d_4 by generating two fresh closures using the CURLs naming r_1 and r_2 and spawning those closures on its own island.

Moving video decoders from one island to another is akin to moving the sink end of the "garden hose" from one location to another. Moving video encoders is the equivalent of shifting the source end to another location. To shift the source of the flow a video encoder (actor) e transmits a closure λ_e of itself to an island I whose fixed assets include a camera and supporting execution site. On island I a new actor is spawned executing λ_e and the original e terminates.

Table 1. Summary of Performance Measurements (* signifies < 0.1%)

	CPU Time (msecs)			
Activity	E	%	D	%
1 All actor activity	1,180,271	86.3	197,583	32.0
1.1 Video processing	1,164,856	85.2	26,628	4.3
1.2 Binding resolution	18	*	50	*
1.3 On-island messaging	14	*	84	*
2 Inter-Island messaging	137,404	10.0	227,018	37.0
2.1 Serialization	198	*	146	*
2.2 Deserialization	8	*	4,480	0.7
3 On-screen rendering	N/A	N/A	120,213	19.6
4 Miscellaneous	50,272	3.4	69,862	11.1
Total CPU msecs	1,367,947	100	614,676	100

We measured the performance of the flow from E to D illustrated in Figure 3 where island E hosts a video encoder and a publish/subscribe relay, and a second island D hosts a video decoder and a display driver. To eliminate measurement error due to network latency we hosted the two islands D and E as separate processes on a single Ubuntu Linux 10.04 host equipped with a 4-core Intel i7 2600K processor and 8 GB of RAM. Our video camera, a Logitech C910, captures 720p resolution video with a maximum framerate of 10 FPS. Our actual framerate in these experiments averaged 9.6 FPS. After flow establishment we streamed data for approximately 2.5 minutes per trial. Using the standard Racket profiler we collected performance data every 0.1 milliseconds and ran a total of ten trials. Our results are reported in Table 1, a sum of the profiles from all ten runs. For any trial where an activity was not observed we assigned 0 to that activity count in the trial (activities 1.2, 1.3, and 2.2 were absent from some profiles). "Miscellaneous" refers to background activities at both actor and island level, such as intermediate message routing, user interface initialization, infrastructure idle time, one-time bootstrapping and setup, and the overhead of profiling.

Like many other video applications all of the low-level video processing of COASTCAST (encoding, decoding, and rendering) is conducted with library routines implemented in the C language and made available in two of the binding environments of islands C, D and E of Figure 3. With this in mind the distribution of CPU activity within these two islands is unlikely wholly representative of COAST-based service architectures at large. However, many service applications will have this general form where MOTILE/ISLAND serves as a coordination and orchestration layer over a variety of backend, computationally-intensive services.

In this environment the overhead of resolving free variables (18 and 50 msecs respectively on islands E and D) is swamped by the costs of video encoding on island E (85% of E's CPU cycles) and video decoding and video rendering on island D (approximately 24% of D's CPU cycles). Much of the actor activity not accounted for by video processing or binding environment lookup consists of actors blocking on their message queues waiting for the next message or closure to arrive. As COASTCAST transmits individual video frames inter-island as closures it is worth noting, in this example, that the cost of serializing closures for transmission from encoding island E to decoding island D is so low as to be insignificant (< 0.1%). Likewise, the cost of deserializing and recompiling those closures on decoding island E is also quite small (< 1%). While this example might be atypical, other measurements (not reported here) for inter-island serialization and deserialization indicate that the overhead is quite modest.

6. Related Work

COAST and MOTILE/ISLAND have been influenced by prior work on mobile code including remote evaluation [26, 27], Scheme-based mobile code languages [12, 15, 23, 30], the actor-like language Erlang [2], the object-capability language E [20], and capability-based architectures [3]. Island self-certification is drawn from self-certifying file systems [19] and self-certifying URLs [17, 31]. Our previous work on CREST [6–8, 14] inspired computation exchange and led us to consider the problem of *scalable decentralized services* that COAST addresses.

7. Conclusion

Starting with an architectural paradigm (computation exchange) we derive an architectural style (COAST) consistent with that paradigm, construct a reference implementation (MOTILE/ISLAND) consistent with the style, and evaluate an application (COASTCAST) that exploits both the style and the reference implementation to deliver novel features arising out of computation exchange. At each step in the progression, from paradigm to style to reference implementation, we sketch the derivation of constraints and mechanisms that ensured each successive step was consistent with its predecessor and, in doing so, draw out the security principles and properties embodied in each step. In this sense the security of the architectural style, COAST, and the reference implementation, MOTILE/ISLAND, is necessary and explicit though not always sufficient. A full analysis of the architecture-induced security properties for the last step of the progression, from reference implementation to application, is not reported here and is the subject of ongoing work. Our approach to the derivation and design of architectural styles and their instantiations as infrastructure, frameworks, and applications hints at a form of disciplined analysis for the directed invention of novel architectural styles.

Up to this point we have focused on basic safety properties for mobile code. However COAST may also contribute toward higher-level security properties, themselves implemented as COAST-based services. In particular, COAST is an attractive medium for attack detection, isolation, mitigation, and non-repudiable blame [32].

Our prior work in self-adaptive architectures [22, 28] suggests that COAST is suitable for architecture-based adaptation and runtime evolution in decentralized systems. We conjecture that variants of mobile code, binding environments and CURLs can induce *unilateral adaptivity*—customized semantics and behavior for selected collaborators in a decentralized ecosystem.

Acknowledgments

This work supported by the National Science Foundation under Grant Nos. CCF-0917129 and CCF-0820222. Any opinions, findings, and conclusions or recommendations expressed here are our own and do not necessarily reflect the views of NSF.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, December 1986.

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

[3] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112. USENIX Association, 1992.

[4] P. Braun and W. R. Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann, 2004.

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

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

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

[8] J. R. Erenkrantz, M. M. Gorlick, and R. N. Taylor. 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.

[9] M. Feeley and G. Lapalme. Using closures for code generation. *Computer Languages*, 12(1):47–66, 1987. ISSN 0096-0551. doi: http://dx.doi.org/10.1016/0096-0551(87)90012-9.

[10] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 180–190, New York, New York, USA, January 1988. ACM.

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

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

[13] M. M. Gorlick. Streaming state kinematics and flow engineering. Technical Report UCI-ISR-06-3, Institute for Software Research, University of California, Irvine, March 2006.

[14] M. M. Gorlick, K. Strasser, A. Baquero, and R. N. Taylor. CREST: principled foundations for decentralized systems. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH'11, pages 193–194, October, 2011. ACM.

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

[16] S. Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

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

[18] R. Kelsey, W. Clinger, and J. R. (Editors). *Revised^5 Report on the Algorithmic Language Scheme*, February 20, 1998.

[19] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, South Carolina, USA, 1999. ACM Press.

[20] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[21] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[22] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, and D. Rosenblum. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May-June 1999.

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

[24] J. H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

[25] C. Shan. Shift to control. In *Proceedings of the Fifth Workshop on Scheme and Functional Programming*, September 2004.

[26] J. W. Stamos and D. K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4): 537–564, 1990. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/88616.88631.

[27] J. W. Stamos and D. K. Gifford. Implementing remove evaluation. *IEEE Transactions on Software Engineering*, 16(7): 710–722, July 1990.

[28] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *Proceedings of the Eighth Joint Working IEEE/IFIP Conference on Software Ar-*

chitecture and Third European Conference on Software Architecture, pages 171–180. IEEE Computer Society, 2009.

[29] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, January 2010. ISBN 0470167742.

[30] D. Vyzovitis and A. Lippman. MAST: A dynamic language for programmable networks. Technical report, MIT Media Laboratory, May 2002.

[31] M. Wolfe. SCURL authentication: A decentralized approach to entity authentication. Master's thesis, University of California Irvine, October 2011.

[32] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First conference on Hot topics in system dependability*, HotDep'05, Berkeley, CA, USA, June 2005. USENIX Association.