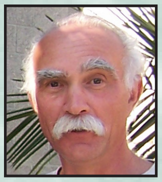
Motile: Reflecting an Architectural Style in a Mobile Code Language



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

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

June 2013
ISR Technical Report # UCI-ISR-13-1

Motile: Reflecting an Architectural Style in a Mobile Code Language

Michael M. Gorlick and Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697
mgorlick@acm.org, taylor@ics.uci.edu

Abstract

Decentralized services, that is, services distributed across multiple, distinct spheres of authority, offer substantial challenges; particularly when we demand that they be both adaptive and secure. We consider decentralized services in the context of COmputAtional State Transfer (COAST), an architectural style for which service adaptivity and security are principal concerns, and discuss how the style is reflected in our reference implementation: MOTILE, a mobile code language, and ISLAND, a complementary peering infrastructure.

We analyze both COAST and MOTILE/ISLAND from the perspectives of safety and security, detailing how these goals influenced the language, its supporting infrastructure, and the extent to which MOTILE/ISLAND conforms to the COAST style. We then evaluate a portion of COASTCAST, a decentralized service for the distribution, sharing, and manipulation of soft, real-time, high-definition video written in MOTILE/ISLAND, to illustrate the security and safety that the COAST style conveys.

The deep interplay between style and language is instructive. Our combined analyses and evaluation demonstrate that the three principal mechanisms of the style—mobile code, execution sites, and Capability URLs (CURLs)—act in concert to provide effective security and safety for decentralized services, with language-specific mechanisms playing a critical role.

Categories and Subject Descriptors D.2.11 [*Patterns*]: client/server

General Terms architectural style, mobile code, decentralized services, distributed services

Keywords COAST, Motile, Island

1. Introduction

We have formulated a set of design constraints (an architecture style) to guide the development of adaptable and secure decentralized services. But an architectural style alone, while helpful, is rarely sufficient. Often a style is deliberately under-specified to avoid limiting the range of implementations and in any case, a style addresses only critical constraints with little if any implementation guidance. The gap between a style and a conforming implementation may be significant, requiring sophisticated algorithms and infrastructure.

To simplify the development and deployment of decentralized services in our style we have implemented a style-specific language and a supporting infrastructure. Our case study, presented here, of the relationships between the style, the language, and the infrastructure reveals both the benefits and costs of a deep, style-specific language and infrastructure. In particular, considerable effort has been devoted to safety and security—we judge the exercise both worthwhile and a contribution to the security technology for decentralized services.

There are numerous examples of effective programming languages whose semantics and features are influenced by one or more architectural styles. We address the question of the intersection of architectural style and language in the context of COmputAtional State Transfer (COAST), a practical architectural style for secure and adaptive decentralized services. Our focus is MOTILE—a mobile code language designed and implemented expressly to meet the safety and security demands of COAST. Since the extent to which an architectural style can and should be reflected in a programming language is an open question, we present MOTILE and COAST as a detailed example of the marriage of language and architectural style.

We first expand on our view of decentralized services as motivation for COAST, describe the style, and offer design guidelines. We next introduce an example of MOTILE in action where a simple decentralized service is adapted by a service consumer to provide a new service not explicitly accommodated by the provider. Adaptations of this form are one of the primary motivations for COAST. From there we turn to a detailed review of the semantics of MOTILE, highlighting the intersection of architectural style, language semantics, service adaptivity, and service security.

The language run-time, embedded in a peering infrastructure called ISLAND, also plays a vital role in maintaining the safety and security constraints of the COAST style, so we review critical aspects of its structure and performance. Next we consider the language and its run-time in the context of a decentralized application for distributing and sharing high-definition, real-time video streams, examining the degree to which the language mechanisms provide tailored security policy in a manner consistent with the style.

2. Languages and Architectural Styles

Creating a style-specific language is, of course, not the only way that architectural styles have been supported in the past. Implementations of architectural styles often take the form of frameworks, for example, C2, an architectural style for GUI software, implemented as Java-based framework [55]; Rails, a RESTful web application framework written in Ruby [45]; and GStreamer, a data-flow style framework written in C for multimedia applications [1].

Frameworks have several distinguishing characteristics. First, they typically exhibit inversion of control where the flow of control is dictated by the framework and not by the application. In other words, the framework embodies a style-specific control flow to which the application must adapt. Second, a framework presents default behaviors that number among the distinguishing benefits of the style. This relieves the developer of the burden of implementing those critical behaviors. Third, each framework is usually extensible in some manner and those mechanisms are the means by which the style is adapted to the specific domain and use cases of the subject system. Finally, each framework also presents immutable behaviors that are not subject to change, often reflected in core portions of the framework implementation that can not be safely modified.

Frameworks may reduce implementation effort, speed deployment, and ease system evolution. Because a framework has many principal system design decisions already "baked in," it reduces the degrees of freedom in system design and consequently lessens development risk. However, a framework may interpret a style so narrowly that, in some cases, it becomes confining or obstructive. Finally, each framework is expressed in a particular programming language whose semantics, structure, performance, availability, or reliability may not be suitable for a specific system.

2.1 Domain-Specific Languages (DSLs)

While many DSLs are clearly not style-specific—such as SQL for relational database queries, `flex` for lexers, `bison` for parsers, or Mathematica for symbolic mathematics—others occupy a fuzzy middle ground between domain- and style-specific. These languages offer lessons for designing style-specific languages. For example, Orc [30] is a functional language used to express orchestrations and wide-area computations in a simple and structured manner reflecting a service architecture in which clients make requests to service providers, an abstraction of REST or WS-*. Orc deals with concurrency, ordering, and failure using a small set of powerful combinators—operators that abstract parallel and sequential execution, blocking, priority, and timeouts. Many common idioms (such as fork/join and priority poll) are implemented in Orc as higher-order functions. Orc illustrates that a rich and robust concurrency semantics requires only a few simple but general primitives combined with higher-order functions.

Adobe Postscript [2] is a functional, concatenative, interpreted language whose primary application is to describe the appearance of text, graphical shapes, and sampled images on printed or displayed pages. Postscript and its interactive derivative, Display Postscript [28], are complex languages containing about 400 commands of which approximately half are related to graphics (a reflection of the richness and complexity of two dimensional page description), the rest being general programming language constructions. Postscript relies heavily on global state to both minimize the size of programs (hence the amount of source code that must be shipped to a printer's Postscript interpreter) and improve interpreter efficiency (an essential consideration given the comparatively impoverished processors embedded in the laser printers of the late 1980s and early 1990s).

One can argue that PostScript is a mobile source-code language reflecting a specialized client/server style where the server is a printer or display device. In any case, it is clear that both the domain and the intended architectural style had a profound influence on the language. Scholz [48], however, contends that alternative formulations rooted in lazy functional programming and monad-based imperative streams are more transparent, efficient, and compact.

galsC is a C dialect targeted to event-driven, resource-constrained embedded systems such as networks of sensors and actuators [14, 15]. galsC is distinguished by its *Globally Asynchronous Locally Synchronous* (GALS) concurrency model in which actors communicate with one another at the application level asynchronously by message passing but within any single actor components communicate synchronously via method calls. Here we see an example of a computational model, actors, adapted to the resource and timing requirements of a specialized domain. Orc, Postscript, and galsC are domain-specific (orchestration, page description, and embedded systems) but each is also strongly influenced by an architectural style: REST, mobile code plus client/server, and event-driven, respectively.

2.2 Architectural Style-Specific Languages

Several style-specific languages focus on the REST architectural style. Links is a programming language for web applications that generates code for all three tiers of a web application from a single source [17]. Intended for rich "Ajax-like" applications, Links generates code for both client- and server-side. It also presents a unified programming model for dealing with the three-tier pattern of web business applications, a client-facing tier, a middle business logic tier, and a database back-end tier. Flapjax is a language designed for web mashups, browser-side applications that communicate with servers and have rich, interactive interfaces [33].

The dataflow architectural style has influenced many domain-specific programming languages, for example LabVIEW, a visual dataflow language for laboratory instrumentation, data acquisition, and industrial automation [57]. Distributed dataflow offers several examples: Swift is a scripting language for executing domain-specific applications repeatedly on large collections of file-based data [59]; however, Swift is deliberately sparse and omits many of the constructs commonly found in other scripting languages such as Python or the Bourne shell. Skywriting, on the other hand, is a complete pure functional scripting language for describing file-based distributed computations [39, 40].

Distributed systems, though too broad to be an architectural style, are addressed by several noteworthy languages. Emerald is a landmark object-based, mobile code language designed to simplify the construction of distributed systems [11, 29]. Erlang, first developed for telephone switches, is a functional language modeled after Prolog for highly reliable, soft real-time, distributed systems [5]. Clojure is a LISP-inspired functional language for distributed applications distinguished by its rich collection of persistent, functional data structures and its commitment to dynamic systems [23]. We comment further on other related languages in Section 8, after the relevant details of MOTILE and ISLAND have been presented.

3. COAST Background

COAST, an architectural style for decentralized services, defines the transfer of computations among peers, the naming of computations, and the interpretation and evaluation of computations [27]. The COAST style explicitly accommodates capability security [34] to confine and regulate visiting computations; its handmaiden, the Principle of Least Authority (POLA) [46], is the guide for allocating, modulating, and transferring capability among computations.

A capability is an unforgeable reference whose possession confers both authority and rights of access to a principal [50, 51]. Unless a computation holds a capability for a principal a it can not *directly* access or manipulate a in any way. In a world of mobile visiting computations capability security is the only reliable means by which a peer can protect itself against misuse, abuse, or attack.

The COAST style posits five rules:

1. *Computation exchange.* All services are computations whose sole means of interaction is the asynchronous messaging of closures, continuations, and binding environments
2. *Functional capability.* Each computation executes within the confines of some execution site $\langle E, B\rangle$ where E is an execution engine and B is a binding environment
3. *Computation names.* All computations are named by Capability URLs (CURLs), an unforgeable, tamper-proof cryptographic structure that conveys the authority to communicate
4. *Communication capability.* A computation may deliver a message to a computation x only if it holds a CURL u naming x
5. *Message interpretation.* The interpretation of a message delivered to computation x via a CURL u naming x is u-dependent

Rule 1 defines computations as closures, continuations, and binding environments and specifies exchange as the asynchronous messaging of same. Rule 1 also ensures isolation of computations since messaging is the only means by which computations may interact. Rule 2 allows COAST-based services to support multiple mobile-code programming languages, each with a distinct semantics and implementation—additional dimensions of service variation.

Rules 2, 3, and 4 are the basis for capability security under COAST. There are only two forms of capability, functional capability, which delimits what visiting mobile computations can *do*, and communication capability, which determines where, when, and how computations can *communicate* with another. Since the sole means of interaction among computations is asynchronous messaging (by Rule 1), Rules 3 and 4 guarantee that computation x, absent a CURL u naming computation y, can never directly communicate with y. By regulating the propagation of CURLs a COAST-based host can exert control over the local interactions of the computations that it executes.

Rule 5 ensures the freedom of computations to interpret messages in a computation-dependent manner. The same message m delivered to computations x and y via CURLs u_x and u_y (naming x and y respectively) may yield entirely different results—a dimension of service diversity.

3.1 Design Intuitions

First and foremost, COAST relies on mobile code [13, 24], a technology whose history stretches back to the mid-1960s. There is wide variation in mobility semantics and state management among mobile code languages, including differences in scoping and name resolution, dynamic linking, state distribution, migration strength, replication, and sharing [18, 19]. At a minimum, a mobile code language compatible with the COAST style must provide closures, continuations, and binding environments as fundamental language objects. However, COAST is silent on many other details, for example the implementation of concurrency and asynchronous messaging, the structure and formulation of execution sites, the transport representation for mobile code, and the exact contents of Capability URLs. These details are nontrivial and influence the efficacy of a COAST-compliant implementation in ways numerous and subtle.

Second, security and peer safety are overriding concerns. Decentralized services complicate security and mobile code exacerbates the risk. Among decentralized services the security environment is at best uneven and fluid. Individual services are governed by multiple, distinct spheres of authority; are implemented, deployed, and maintained by distinct, autonomous organizations; and evolve independently in response to the interests and circumstances (legal, regulatory, economic, social, competitive) of the responsible organization. By definition there is no one, single security perimeter; instead, there are multiple perimeters, and the failure of one perimeter must not lead to the breach of others.

Ajax-based mashups [32], itself a style reliant on mobile code (JavaScript source code embedded in web pages), is a popular form of decentralized services as well as a cautionary example. Historically, Ajax has introduced new vulnerabilities into web applications and increases the attack surface of applications by exposing additional targets for exploitation—both server- and browser side. Mashups are prey to a broad variety of attacks, such as XML external entity injection, request forgery, cross-site request forgery, covert manipulation of the browser DOM such as user interface redress ("clickjacking") or the capture and covert loss of sensitive information, and can be exploited as a launch point against other vulnerabilities. [53]. As we shall demonstrate, MOTILE/ISLAND capitalizes on the COAST rules to thwart attacks, reduce risk, and minimize damage.

Third, COAST-based services blur the distinction between service provider and service consumer; a service consumer can deploy an application to a provider just as easily as a service provider can deploy an application to a consumer. Decentralized services organized around mobile code present distinctive challenges to service consumers and providers alike. Variety and variation predominate. No single uniform code base is likely among large numbers of disparate services. While a small common core is feasible, beyond that, organizations will tend to specialize, driven by their own interests and needs. As we shall see, COAST-based mobile code helps providers and consumers compensate for the service omissions of others. Decentralized partners may come and go, alter security policy, or adjust trust levels as they please. The mechanisms of decentralized services must respect this reality. Fortunately, the security mechanisms of COAST are symmetric and dynamic, allowing providers and consumers alike to maintain a security posture appropriate to their needs and circumstances.

The influence of COAST on MOTILE/ISLAND, our reference implementation of COAST, is best motivated and demonstrated by a small MOTILE program. Section 4 presents a decentralized MOTILE implementation of a search for palindromes to sketch how a MOTILE programmer, building upon the assets presented by a service provider, creates a client-specific, custom service in a COAST application.

4. MOTILE/ISLAND in Action: Adaptive Services

An adaptive service offers service primitives that may be composed and rearranged on demand by clients to suit client needs. MOTILE, a single-assignment dialect of Scheme [21], is the language of service interaction and adaptation between provider and consumer. Each MOTILE computation is an independent thread of control (an actor [3]) executing within the context of an execution site $\langle E, B\rangle$ (COAST Rule 2). E is the execution engine for the actor: an interpreter, bytecode virtual machine, physical processor, JIT compiler, or the like. B is a binding environment, a map from symbols to values. At a minimum the values encompass immutable primitives such as numbers, characters, strings, and byte strings; persistent, functional, compound data structures such as lists, vectors, tuples, records and hash maps [41]; and higher-level constructs such as closures, continuations and binding environments.

When is the binding environment B of an execution site referenced? *Whenever the closure under execution references a free symbol not bound over lexical scope.* Consequently, when a closure is transmitted from one actor to another (via a message—the only means by which actors interact) it leaves all of the bindings of its free variables behind and obtains new bindings from the binding environment of the execution site in which it is called.

Why these particular binding semantics for MOTILE mobile code? These binding semantics:

- Are easy to explain and understand and are obvious when inspecting MOTILE source code. Other mobile code languages such as MAST [58], allow programmers to pick and choose which free variable bindings to drag along, as do some formal models [8]. These alternative formulations are COAST-consistent but are more difficult to explain and understand and their notation can complicate source code.
- Enforce functional capability. It is impossible for arriving mobile code to carry unwanted or dangerous functions through the security perimeter of a computation as a closure leaves all of its "global" bindings behind as it transits from one execution site to another. Since all executions are grounded in the contents of the binding environments

of their respective execution sites their "functional reach" is bounded. By restricting the contents of binding environments (an application of POLA), hosts can minimize the risk or at least confine the damage of an erroneous or malicious mobile closure.

- Improve security. Appropriate construction of the closures populating a binding environment (we omit the details here) prevents proprietary or sensitive implementations from ever leaving the confines of the execution site in which they appear. A COAST service provider can safely include specialized functions within its execution sites without fear that the implementations will be transmitted outside of its sphere of authority
- Offer opportunities for per-binding customization such as logging, monitoring, debugging, and restriction
- Relieve the COAST implementation of the burden of requiring a single, uniform execution engine throughout all decentralized services, thereby encouraging diversity of implementation and providing an important avenue for evolution and innovation

Finally, unlike other languages such as Lisp, Lua, PHP, Python, Ruby, or even classic Scheme, MOTILE does *not* contain an `eval`-like function. Consequently no MOTILE program (closure) can dynamically construct a program abstraction and then execute it. This restriction allows MOTILE/ISLAND to authoritatively determine the complete set of global functions that a closure *may* reference. A sphere of authority may simply decline to execute visiting code that references functions that may be easily abused (for example, functions for creating, writing, and reading scratch files) or may more closely monitor its execution than it might otherwise.

To illustrate MOTILE binding semantics consider the function `palindrome?` shown in Figure 1:

1. The symbols `s`, `left`, and `right` (lines 1, 3, and 4 respectively) are each bound within lexical scope
2. The expressions `(let ...)`, `(or ...)`, `(and ...)` are special forms recognized by the MOTILE compiler and in this context none of `let`, `or`, `and` are free symbols
3. The symbols `sub1` (lines 4 and 11), `string-length` (line 4), `>=` (line 6), `char=?` (line 8), `string-ref` (lines 8 and 9), and `add1` (line 10) are free but *not* bound within lexical scope

The latter set (item 3 above—`sub1` and its ilk) are bound (defined) by the binding environment B of the execution site. When a closure is transmitted to an execution site the bindings of all of its free variables not in lexical scope are left behind and rebound (redefined) by the binding environment of its destination execution site.

```
(define (palindrome? s)                      1
  (let loop                                  2
    ((left 0)                                3
     (right (sub1 (string-length s))))       4
    (or                                      5
     (>= left right)                         6
     (and                                    7
      (char=? (string-ref s left)            8
              (string-ref s right))          9
      (loop (add1 left)                     10
            (sub1 right))))))               11
```

Figure 1. A MOTILE predicate to test for palindromes. It returns true if its string argument s is a palindrome and false otherwise.

4.1 MOTILE Actors and Message Passing

Each MOTILE actor resides on an *island*, a single, network-accessible, uniform address space. An island is a locus of MOTILE computations operated and deployed by a single sphere of authority, the COAST analog of a web server. Each island is uniquely identified by an island-specific public key [7] that allows any two communicating islands to certify the identity of the other.

Actors are *spawned* by other actors. The initial trusted actors on an island are started ab initio when the island is created. The MOTILE primtive function (`spawn` f B) enables actor x residing on island I to create an additional island co-resident actor. It takes two arguments, a thunk (a zero-argument closure) f and a binding environment B. It spawns a new actor executing thunk f in the context of an execution site $\langle E, B\rangle$ where E is the execution engine of actor x and B, the binding environment argument of `spawn`. Remote (inter-island) spawning is implemented using `spawn` in combination with inter-actor message-passing.

Associated with each MOTILE actor is a *mailbox*, a queue for incoming messages. The MOTILE function `(receive)` returns the next queued message and removes it from the mailbox. If the mailbox is empty then `(receive)` blocks until a message arrives. MOTILE/ISLAND guarantees that messages sent from actor x to actor y arrive in transmission order; while not required by the actor model it simplifies the implementation of many distributed algorithms. Messages are the only means by which MOTILE actors may interact (COAST Rule 1).

Each Capability URL (CURL) names an actor on some island and conveys the capability to communicate with that actor (COAST Rule 3). The MOTILE function

(`curl/send` u v)

which takes two arguments, a CURL u and a MOTILE value v, transmits v asynchronously to the mailbox of the actor x named by u.

We now have enough to implement inter-island spawning

```
(let*                                          1
    ((reply (promise/new 60.0))                2
     (reply/settlement (car reply))            3
     (reply/curl       (cdr reply))            4
     (palindromes                              5
      (lambda ()                               6
        (curl/send reply/curl                  7
          (words/filter palindrome?)))))       8
  (curl/spawn J@ palindromes)                  9
  (promise/wait reply/settlement))             10
```

Figure 2. A MOTILE program to search for palindromes.

```
(define (curl/spawn u@ f)
  (curl/send u@ (list "SPAWN" f))
```

where argument `u@` is a CURL naming an actor to which the message `("SPAWN" f)` is directed. `curl/send` delivers a list $(u\ t\ v)$ where u is the CURL to which message v was directed and t is the identity of the transmitting island. This delivery structure is essentially dictated by COAST Rule 5 that grants each actor the right to interpret a message v in the context of the CURL u employed by the sender. At its discretion receiving actor x will (by convention) execute `(spawn f B)` to spawn an new actor executing closure `f` in the context of a binding environment `B` where the bindings of `B` may be u-, t-, and v-specific. However, x could interpret the value `("SPAWN" f)` quite differently, for example, logging the spawn request and forwarding it on to actor y for spawning only if the originating island t was known to be trustworthy (COAST Rule 5).

This simple example also illustrates two other consequences of the COAST rules. Actor x can spawn another co-resident actor only if:

- The `spawn` primitive function is defined in the binding environment of the execution site of x or
- Actor x communicates with actor y with the requisite functional capability. Either x either holds a CURL u for y or x holds a CURL u for actor z that forwards (directly or indirectly) the request of x on to y

This observation is the cornerstone of COAST capability security. In the former case x possesses the functional capability to spawn co-resident actors. In the latter case x possesses both the functional capability (`curl/send`) and the communication capability to enlist a proxy actor to act on x's behalf.

MOTILE implements "communication by introduction" [16] meaning that actor x can't directly communicate with actor y until it has been "introduced" to y. Actor x can acquire a CURL (an introduction) in only three ways:

- The closure with which it is spawned contains one or more CURLs among its lexical scope bindings
- Functions in the binding environment of the execution site of x may return CURLs as their values or the value of a binding (say a list) may contain one or more CURLs
- CURLs may be contained in the messages that x receives

Consequently, to the extent that an island's sphere of authority can interdict the immigration of CURLs embedded in closures, control the contents of the binding environments of execution sites, and embargo the transfer of CURLs in inter-actor message traffic it can regulate the transfer and propagation of communication capability.

4.2 Intra- versus Inter-island Message Passing

Actors residing on the same island, by definition of an island, share a single, homogeneous address space. MOTILE/ISLAND, for reasons of efficiency, implements intra-island message passing by reference. To eliminate the distinction between intra- and inter-island message passing MOTILE is a single-assignment language in which all primitive and compound values are immutable. Data structures such as extensible vectors and hash tables are functional and persistent [41], that is, immutable and version-preserving. All objects (including messages) shared among an island's actors are immutable; this improves actor isolation and eliminates the classic data races seen in threaded imperative languages.

4.3 Promises

To bridge the gap between functional programming and asynchronous messaging MOTILE implements *promises*, a proxy object for a result that, at the outset, is undefined because the computation of its value is incomplete. Figure 2 illustrates the use of promises. The client computation creates a new promise (named `reply`) with a lifespan of 60.0 seconds (line 2). There are two components to each promise, a *settlement* s and a CURL u (lines 3 and 4 respectively). The settlement s is effectively an opaque container (implemented as an actor) for the expected value while u is a single-use CURL that references the settlement. A single-use CURL u may be used as the target for a message transmission only once; thereafter any message transmitted via u will be immediately discarded by the receiving host.

Many computations within the network may hold a promise CURL u simultaneously, but only one message transmission via u will resolve the promise; any others will be silently rejected. Finally, `promise/wait` (line 10) blocks until either the promise is resolved (that is, some actor has transmitted a value to the settlement via the CURL `reply/curl`) or the promise expires (in this case 60 seconds after it was created). If the promise has been resolved then `promise/wait` returns the value of the resolution; otherwise it returns false, the default value if resolution fails (other options are available but omitted here for the sake of simplicity).

4.4 A Client Computation for Palindromes

Figure 2 outlines how a COAST client constructs an alternative service from the primitives defined by a service provider. For the sake of brevity, we assume that the `palindrome?`

predicate shown in Figure 1 is defined within the lexical scope of the code of Figure 2.

The variable `palindromes` (line 5) is bound to a thunk (a zero-argument closure) defined in lines 6–8. The function `words/filter` (line 8) is a domain-specific function defined by the binding environment of the destination execution site. It is a combinator that applies its argument, a predicate function (the function `palindrome?` defined in Figure1), to each word contained in the database of host J, the service provider, and returns a list containing only those words satisfying that predicate. The list of filtered words, each word a palindrome, is the value that resolves the promise settlement (`reply/settlement`) held by the client (lines 7–8). In line 9 the client actor spawns the execution of the `palindromes` computation on island J via CURL `J@` (we omit here, for the sake of clarity, the details of how the client actor came to possess this CURL). Finally (line 10), the client actor blocks, waiting for its list of palindromes.

4.5 Summary

The palindromes example of Figures 1 and 2 illustrates the critical role that binding environments (COAST Rule 2) play in COAST-based services. However, while the COAST rules call for closure transfer as a form of computation exchange (Rule 1), the style is silent on the the binding semantics for the variables of transported closures. MOTILE closure transfer always rebinds free variables not closed over lexical scope in the context of the binding environment of the destination execution site; in other words, the bindings for global free variables are always left behind. This choice is easily explained and understood, maximizes the opportunities for service variation and differentiation, and guarantees that the destination host has complete control of the functional capability that it allocates to visiting mobile code. Here a single element of an architectural style deeply influences the semantics, adaptivity, and security of a style-specific language.

5. Motile/Island Implementation

The COAST style is deliberately under-specified and many significant details relevant to security and safety are left to its implementation. For example, COAST Rule 2 requires that each COAST computation is confined to an execution site, but says nothing about the structure or semantics of execution engines or the particulars of binding environments, both of which are critical for managing functional capability. Nor does COAST include mention of serialization or the network transport representation for MOTILE mobile code and values. Here we sketch the details of the MOTILE compiler pertaining to execution engines, describe the interaction between compiler and serializer, and outline the implementation of binding environments—all elements of enforcing MOTILE binding semantics.

Both MOTILE and ISLAND are implemented in Racket[1], a well-known, high-performance Scheme implementation with a machine-independent bytecode virtual machine coupled to a native code JIT. MOTILE is implemented as a Scheme-to-Scheme closure compiler [22, 56] in which each MOTILE closure is compiled in lexical scope passing, binding environment passing, continuation passing style. The output of the MOTILE compiler is essentially a fixed execution engine (implemented as threaded code) for which the binding environment is an argument. In other words, our reference infrastructure holds the execution engine constant but allows the binding environment to vary independently among an island's execution sites.

From the perspective of the underlying Scheme host each compiled MOTILE closure f is a Scheme function $(f\ k\ s\ B)$ of three arguments: a continuation k (implemented as a single-argument function in the underlying Scheme host), a lexical-scope stack s (whose top frame contains f's arguments) and a binding environment B. This representation of f is hidden from users at MOTILE-level.

Each MOTILE closure f "knows" how to decompile itself into a MOTILE Assembly Graph (MAG), a machine-independent, engine-independent, directed graph representation of f. MAG nodes are instructions for an abstract, high-level lambda calculus engine whose annotations contain the lexical scope values captured by the closure. A MAG graph is human-readable and it is possible, though tedious, to directly program in the MAG representation. The call $(f\ 1\ \ldots)$ by trusted island-level code returns the MAG of MOTILE closure f. Other argument combinations (in each case the first argument is a distinct small positive integer) return additional f-specific meta-level information. A similar technique is used by Mobit [42], a portable Scheme-to-Scheme translator for Termite [25], an Erlang-influenced mobile code Scheme.

The MOTILE/ISLAND serializer is trusted code that reduces MOTILE values to a flat structure. As the serializer walks a value it decompiles any MOTILE closures that it finds into their MAG representations and flattens the graph. The serializer deliberately rejects some MOTILE types; for example, the serializer will not serialize a MOTILE actor since actors are pinned to the island on which they were born.

The deserializer is the inverse of the serializer. It accepts an instance of the flat structure generated by a serializer and reconstructs the expression; in particular, any MAGs are reconstructed into their graph form and then recompiled into closure form. For any value v for which $(\text{serialize } v)$ is defined the value returned by $(\text{deserialize } (\text{serialize } v))$ is structurally equivalent to v; in other words, structure sharing in v is preserved.

MOTILE binding environments are implemented as ideal hash trees [6], a functional data structure that is both immutable and version preserving [41]. As binding environ-

[1] `www.racket-lang.org`

ments and their contents are immutable they may be freely shared among multiple execution sites within a island. They are parsimonious as well, since a derived binding environment leaves the root binding environment unchanged (version preserving) and wherever possible an ideal hash tree shares unchanged subtrees with the tree from which it is derived. "Sculpting" a binding environment (that is, adding, removing, or modifying bindings) is inexpensive and there are no significant performance barriers to customizing a binding environment for each and every actor on an island. In practice many actors on an island will share a common binding environment.

The MOTILE compiler, in cooperation with the ISLAND serializer/deserializer, enforce the binding semantics for MOTILE mobile code. All free variables α not bound within lexical scope are compiled into lookups by name using the MAG instruction $(\mathrm{reference/global}\ \alpha)$ where α is the name of a binding within the binding environment of the mobile code's execution site.

The interactions between compiler and serializer/deserializer are sufficient for the binding semantics but will not pin sensitive functions in an island binding environment to the island. To illustrate, suppose that `zig` and `zag` are two closely held functions in a binding environment of an execution site on island I. But island I can't prevent MOTILE code from transmitting them off-island to island J—the expression `(curl/send J@ (list zig zag))` will (apparently) transmit both functions to island J from island I. Preventing this behavior requires additional cooperation between binding environments and serialization, as detailed in Section 6.

6. Serialize for Security

Three distinct but related mechanisms are the building blocks for important elements of intra- and inter-island security: binding environments, the binding semantics for MOTILE closures, and (de)serialization. Combined they implement fixed assets (pinning a computational, algorithmic, or information resource to an island) as well as the tracking and interdiction, both intra- and inter-island, of functional and communication capability. An island can monitor in real-time the transfer of functional capability on-island *and* the transfer, both incoming and outgoing, of communication capability. Knowledge of these transfers is essential for constructing higher-order security awareness [60] among decentralized services and the critical role of (de)serialization in this regard is an unexpected result.

In retrospect COAST facilitates the pivotal role of serialization. Its rules explicitly define the location of functional and communication capability and confine its transfer among computations to messaging. COAST is silent on the means of organizing and delineating spheres of authority but any implementation of the COAST style for decentralized services must choose a model. Once that choice is made the only points of interdiction, embargo, and control are binding environments and the transfer of CURLs and closures by messages.

Distributed messaging demands serialization, but the precise nature of that serialization is determined by choices well outside the scope of the COAST style. The abstract transfer representation of closures—a graph of instructions for a lambda calculus engine—works in our favor since it demands "deep" inspection for the sake of mobile code transmission and reception. A lower-level representation, such as byte codes for virtual machine instructions, requires less effort and inspection but is more difficult to analyze and less amenable to interdiction and embargo. In any case, both what the COAST style dictates and what it leaves unsaid, have strongly influenced MOTILE and its run-time support.

6.1 Inter-island MOTILE Binding Semantics

Circling the wagons to interdict the transmission of `zig` and `zag` (Section 5, last paragraph) requires one additional refinement. The binding environments of execution sites are largely populated with primitive operations, that is, functions implemented in the underlying Scheme host (for example `+`, `map`, or `list`) or are wrappers for domain-specific functions implemented in another language altogether such as C or Objective-C. In both cases mobility for such functions is impossible and likely not machine-independent; after all, what is the MOTILE/ISLAND serializer to do with a Racket Scheme or C function? Rationalizing the capture of global bindings is necessary as the sample code of Figure 3 illustrates: two global bindings, `zig` and `zag`, are captured in lexical scope (line 1) and then (apparently) transmitted off-island (line 2).

```
(let ((x zig) (y zag))                    1
  (curl/send J@ (list x y)))              2
```

Figure 3. Transmitting a fragment of a binding environment from island I to island J.

Confining functions to prevent their transmission off-island is quite simple. Each function f in the binding environment B of an island execution site is wrapped such that when decompiled by an island serializer it decompiles into a single MAG node $(\mathrm{reference/global}\ \alpha)$ where binding $\alpha : f \in B$. The wrapping mechanics are implemented as a small set of Racket Scheme macros and are applied to execution-site binding environments as islands are instantiated and deployed by a sphere of authority.

Enforcing MOTILE mobile code inter-island binding semantics, and in particular guaranteeing the integrity of island fixed code assets, requires the intimate cooperation of the MOTILE compiler, the detailed construction of execution site binding environments, and the serializer/deserializer of the underlying island infrastructure. The COAST style permits other interpretations of the mobile code binding semantics, but the demands of decentralized services, particularly the trust boundaries among multiple spheres of authority, dictate that an authority have the power to restrict asset mobility. It is an excellent example of the influence of safety and security on multiple, interacting subsystems.

6.2 Intra-island MOTILE Binding Semantics

Mobile code exchanged inter-island must be serialized for network transmission by the originating island and then deserialized by the receiving island, but intra-island code exchange is simply by reference. Consequently, actors x and y executing in the context of binding environments B_x and B_y can exchange functions found only in their respective binding environments, for example x sending function `zip` to y[2] via `(curl/send y@ zip)` and y sending function `zap` to x via `(curl/send x@ zap)`.

How can we prevent co-resident island actors from sharing functional capability that should remain sequestered? The answer combines serialization with binding environment sculpting. Consider the example above where actor y on island I transmits the closure `zap : ...` $\in B_y$ to x on island I in `(curl/send x@ zap)`. Serializing the value of variable `zap` means serializing the closure f to which `zap` is bound, that is, `zap` $: f \in B_y$. The construction of B_y guarantees that closure f will serialize to the MAG instruction $(\text{reference/global}$ `zap`$)$, thereby interdicting f. The second half of the answer requires that the primitive `curl/send` $\in B_y$ be specialized to serialize every message directed to a co-resident actor and then transmit the deserialization to its destination. Specializing `curl/send` and inserting it into binding environment B_y is straightforward and the cost of serialization/deserialization is low. However, this treatment increases the overhead of on-island actor communications over unmediated transmission by reference.

For many actors interdiction of this form is unnecessary: their binding environments may not contain closures that justify this treatment, their communication capability is sharply restricted (that is, they can't acquire the CURLs of actors that may present a threat to island integrity), or they may be executing trusted code whose behavior is well understood and consistent with island security policy. Sharing closures from a binding environment may be appropriate, for example, an actor can be a function factory, wrapping and specializing closures from its binding environment for distribution on demand to other actors, or it may distribute closures to actors for dynamic adaptation or upgrade.

6.3 Serialization/Deserialization is a Security Tool

By definition, serialization must recursively walk and dissect an entire compound or structured value (for example lists, hash maps, or closures). Serialization is the mobile code equivalent of deep packet inspection—everything within the object under serialization is exposed to view. The MOTILE/ISLAND serializer exposes the presence of functional and communication capability; detecting and noting each and every closure and CURL that it encounters as it serializes a value. Detecting closures originating from the binding environments of execution sites is easy, as they always decompile into the MAG instruction $(\text{reference/global}\ \alpha)$ for some symbol α and are the only MOTILE closures to do so. CURLs, a first-class data type in MOTILE, are just as easy to discover as serialization proceeds.

Serialization, the *outward* bound gate through which all inter-island communications must pass, exposes the outgoing transfer of communication capability (CURLs) among islands. The transmitting island always knows which CURLs are going where and can interdict (prevent the transmission of) any CURLs that violate island security policy. For example, islands contain actors responsible for critical island services such as communications and processor allocations whose CURLs should never be exposed off-island. Conversely, deserialization, the *inward* bound gate through which all inter-island communications must pass, exposes the incoming transfer of communication capability among islands. At this point an island I has an opportunity to prevent an untrusted, island-resident actor from acquiring a CURL it should not have, for example a CURL naming an island whose behavior is suspect or blacklisted. Nor can an island J hide a questionable CURL u buried inside a closure it spawns on I since I's deserializer will bring u to light as it recompiles the closure for execution on I.

Serialization and deserialization are required under any circumstances for inter-island communications and the additional overhead of collecting the CURLs embedded in inter-island messages is insignificant in comparison to the base costs of (de)serialization and network overhead. It is possible for an island I to track, for every actor x spawned on I by another island, all of the CURLs that x may hold either because they were encapsulated in the closure defining the spawning or they were delivered to x in messages (including closures) transmitted from other islands.[3] Island I can also assemble a partial view of the distribution of its CURLs among other islands by tracking I-CURLs in outbound messages and their subsequent use by remote islands to communicate with actors on I. More generally, aggregations of islands can pool extracts of their individual views of CURL distribution to obtain a broader perspective of CURL distribution as a whole that can be used for monitoring, debugging, performance, threat and traffic analysis, and early warning of attacks.

Where required, serialization and deserialization can be employed, as described in Section 6.2, in a like manner to track the transfer of functional capability (closures from binding environments) among actors intra-island. It is possible, if required by island security policy, for an island

[2] The shorthand `x@` and `y@` denotes CURLs for actors x and y respectively. How x came to hold `y@` and y hold `x@` is not discussed here.

[3] Tracking the CURLs delivered to x in messages from other, co-resident I-island actors is also easily done using a minor variation of the techniques described in Section 6.2.

to comprehensively track the transfer of critical closures island-wide, actor by actor, with an exhaustive accounting of the functional capability held by each actor. Alternatively, an island can track selectively, focusing its attention on high-value or suspicious actors.

7. CURLs: The Final (Security) Frontier

Capability URLs (CURLs) are the only form of communication capability in COAST; possession of a CURL u for actor y by actor x is necessary for x to deliver a message to y. The restrictions of functional capability and communication capability enforce "communication by introduction" and delineate the means by which any actor can acquire CURLs (Section 4.1). COAST is silent on the details of CURLs, leaving the choices of structure, mechanics, and cryptography open. MOTILE/ISLAND takes advantage of this lack of specificity to broaden and strengthen the security and safety of communication. Here the strength of the COAST style lies not in what it prescribes but what it excludes: the COAST style, by restricting all inter-computation communication to messaging via CURLs, offers conforming implementations an architectural "choke" point for implementation-specific interpretations of COAST messaging. Additional implementation details and examples of CURL use can be found in [26].

7.1 CURLs are Cryptographic Structures

An actor may generate CURLs for itself (assuming that the binding environment of its execution site contains the requisite primitive functions) but not for any other actor. CURLs are primitive MOTILE data types, are immutable, and each CURL is signed by the issuing island; consequently, CURLs are effectively impossible to counterfeit and are tamper-proof (COAST Rule 3).

CURLs are cryptographically unique—not only are the CURLs of one island distinguishable from the CURLs of another island but the CURLs of a single island are also distinguishable from one another. However, under MOTILE/ISLAND it is not possible for island I to reliably determine the actor to which a island J CURL refers. In other words, if I comes into possession of two J island CURLs, u and u', I can not reliably determine from the CURLs alone if u and u' both refer to the same actor x on J. This "cloaking" frustrates I's efforts to discover J's internal structure, thereby impeding attacks against J.[4]

7.2 CURLs Protect the Perimeter

CURLs are the "frontier" of an island. Guarding the frontier is fundamental to island security and safety. To this end each MOTILE/ISLAND CURL u is metered by a use count, limiting the total number of message transmissions allowed for u; a rate limit, a ceiling on the rate of message transmissions for u; and an expiration date, after which u is invalid. A use count of $n > 0$ for CURL u means that u may be used at most n times to transmit and deliver a message and is decremented when a message is delivered to the mailbox of the actor denoted by u. When an n-use CURL u is shared among multiple islands the (at most n) uses are first-come first-served.[5]

CURL rate limits are expressed in Hz, the maximum number of message deliveries per second, for example; a CURL for which the maximum rate is 1 message every 10 seconds has a rate limit of 0.1. ISLAND uses a leaky bucket algorithm to gauge the rate of message arrival; message deliveries exceeding the rate are silently rejected. When a CURL u is shared among multiple islands the message arrival rate is shared, first-come first-served, among those islands. In this case one island may enjoy a higher delivery rate relative to the others depending on the combined transmission rates, delivery timing and network conditions.[6]

Each CURL has an expiration date and once the CURL expires all messages sent via the CURL will be rejected by the destination island (since no island can safely rely on a sending island to respect the expiration date). CURL lifespans may be as short as a few milliseconds but common practice may see lifespans of a few hours to months. Use counts, rate limits, and expiration dates require that islands maintain island-side state for each CURL they issue. We omit details of the mechanisms but note that, at worst, the state memory amounts to a few tens of bytes per CURL and even a modest island can afford to have 10^5–10^6 CURLs in circulation at any one time.

Finally, a CURL u may be revoked by the issuing island at any time prior to its expiration. This allows the island, should it determine that u is being misused or has been acquired by an untrustworthy island, to immediately protect itself against further hostilities. CURL revocation may also be used to protect critical resources against the actions of erroneous or faulty islands or if the ill-behaving island is itself thought to be under attack. Preemptive isolation may slow or constrain the propagation of attacks and limit their damage.

7.3 CURL-Specific Constraints

Decentralized services must maintain a defensive posture since even a trusted partner, once hacked and penetrated, may be used as a platform for attacks on other portions of a decentralized infrastructure. Under COAST protection and adaptation are two sides of the same coin and to these ends a CURL may be further constrained in time, space, domain, or application by *guards* embedded within the CURL. Guards can implement: complex temporal restrictions (an island restricting service to a client to the even calendar days of the

[4] In general island I no more trusts an actor x on island J than it trusts island J itself.

[5] ISLAND also provides a mechanism by which multiple I island *distinct* CURLs $u_1, \ldots, u_m$, share a single use count.

[6] The same mechanism by which an island I shares use counts also allows multiple distinct CURLs of island I to share a single rate limit.

month); delegation (restricting use to a set of trusted islands thereby limiting the effective propagation of a CURL among islands); defensive consistency (ensuring that incoming messages conform to the expectations of the receiver); or preconditions based upon observables such as an island's workload or the spot price of gold on the London exchange.

CURLs, at the discretion of the issuing actor or island, may contain arbitrary metadata, including closures and binding environments. Among other roles CURL-borne closures implement guards: MOTILE predicates g of three arguments $(g\ u\ t\ v)$ where u is the CURL used by the sending island; t is the cryptographic identity of the sending island (including its network address); and v is the message payload, the value transmitted by the sending island. Its return value may depend upon the metadata contained in CURL u, the identity t of the sender, the value of the message v, or arbitrary conditions and events within the issuing island I or other islands known to I. The guards $g_1, \ldots, g_m$ of a CURL u issued by an island I are evaluated in sequence in the context of g_i-specific (hence I-specific) binding environments.

When a message m arrives at island I via CURL u then message m is deposited in the mailbox of the target actor only if u has not been revoked, the use count of u is positive (> 0), u is unexpired (the expiration date of u lies in the future), the arrival rate of messages via u (including the arrival of message m) does not exceed the rate limit of u, and the guards of u evaluate to true; otherwise the message is rejected by I.

7.4 Summary of CURLs

Managing communication capability among decentralized services is critical for limiting risk, containing damage, or minimizing loss of information from attacks. In this instance the influence of the COAST style is lightly prescriptive—it limits communication capability to CURLs but does not restrict the formulation of CURLs in any substantial way. MOTILE/ISLAND exploits this *freedom* in two distinctive ways: it enforces metered communication (per-CURL use count, expiration date, and rate limit) and employs COAST mobile code embedded in CURLs to enforce arbitrary restrictions on communications in space (delegation and non-delegation), time (temporal access), and domain (for example, limiting communications when workloads exceed an island-specific threshold). Since CURLs are tamper-proof[7] the issuing island of a CURL u can safely and securely embed island-, domain-, actor-, and CURL-specific functions within the CURL to regulate and modulate communication capability. This result is unexpected—when adequately constrained, mobile code can be used to implement rich and complex forms of higher-order security. Here the influence of the COAST style on the language and infrastructure is profound, and what the style omits is just as important as what it includes.

[7] A ny island J holding a CURL u purportedly issued by an island I can determine conclusively if u is counterfeit or corrupted in some manner.

8. Related Object-Capability Languages

COAST is an architectural style for which both functional and communication capability are explicit elements (Section 3, Rules 2, 3, and 4). Here, we examine related work through the lens of capability-based security; other important aspects of the related work are detailed in [26, 27].

First introduced in the context of operating systems such as KeyKOS [12], Amoeba [54], and W7 [44], capability-based security is embodied in several object-oriented programming languages such as E [34], AmbientTalk [20], and Caja [38]. All of these languages are examples of the *object-capability model* [36, 52], which restricts the classic object-oriented model in three ways: only connectivity begets connectivity (all access must derive from previous access), absolute encapsulation (no access to object internals absent an object's explicit consent), and all authority by references only[8] (no object can synthesize a reference to another object from bits alone).

Maffeis, Mitchell, and Taly [31] "formalize a form of [the] object-capability model, focusing on reachability properties, in the context of operational semantics of imperative languages and compare capabilities with authority principles" to prove that capability safety implies authority safety (no amplification of authority). They go on to prove that a subset of Caja [38], an object-capability JavaScript dialect, enforces capability safety. Caja is an example of a strong, dynamic typed object-capability language.

In Emerald, a mobile object language [29], object references are capabilities. Objects encapsulate the concepts of process, procedure, data, and location; objects are the units of programming and distribution, and the endpoints of communication within a distributed system. In addition, object location and mobility are explicit notions within the language. Everything within Emerald is an object, including primitive values such as booleans or integers. Emerald enforces abstract types (known informally as "duck types" today), objects are instantiated by nested lambda invocations, rather than classes or prototypes, and are fully encapsulated [43], a necessary restriction for the object-capability model. Emerald is an early example of a strong, static typed object-capability language.

Erlang [5] is a functional, single-assignment language with immutable data types. Orginally designed for implementing high-reliability, fault-tolerant, distributed systems, Erlang is based on the Actor model [3] of computation—isolated processes (actors) communicating solely via asynchronous messaging. Each actor is fully encapsulated, there is no global state, and actor references are capabilities (granting the right to communicate)—the rules by which actor references are acquired and transmitted conform to the

[8] For example, casting an integer as a pointer is a violation of this constraint.

object-capability model. Erlang is an example of a strong, dynamic typed object-capability language.

9. Evaluating COAST and MOTILE/ISLAND

Our analysis here emphasizes *capability confinement*, the degree to which a system can limit and regulate the propagation of capability—a critical element of safety and security for decentralized services [35, 37] and a principal concern of COAST.[9] We consider the efficacy of COAST-based capability security in the context of COASTCAST, a COAST-based application for distributing and manipulating high-definition video streams in soft real-time and in doing so evaluate MOTILE/ISLAND as an object-capability language. A functional and performance analysis of COASTCAST is given in [27].

For the sake of capability security COASTCAST assumes three different kinds of islands: camera islands, display islands, and relay islands; each distinguished by the services (execution sites) that it offers. Camera islands offer services for access to one or more live video cameras, display islands support display and control services for video streams, and relay islands provide publish/subscribe services to distribute compressed video streams to display islands.[10]

The allocation and propagation of capability is framed in terms of CURLs. First, access to camera services is restricted to only a few islands via non-delegable camera CURLs with limited use counts and lifespans. Each camera CURL contains camera-specific metadata including frame size, frame rate, and color model.

To obtain camera service on camera island I a client island J dispatches a thunk t via a camera CURL u for execution. In response (assuming that J is a legitimate delegate for u) I inspects CURL u and generates a custom binding environment B_u for t containing initialization, frame read, and encoder primitives tuned to the frame size, frame rate, and color model given in the CURL as well as transforms or other image operators offered as primitives. A camera island can issue a collection of CURLs to clients with each CURL specifying a different combination of frame size, frame rate, and color model.

In addition, camera island I imposes resource caps on the actor x it spawns to execute t to ensure that x does not waste processor cycles, hoard frames, or consume excess network bandwidth in transmitting the encoded video stream to the relay chosen by t (the CURL for the relay is included in the lexical scope bindings of closure t). It is straightforward (though not implemented) to also limit the lifespan of the client actor that is consuming video frames and, by using CURL rate limits, to bound how frequently any particular client may deploy its mobile code to the camera service. A camera island may provide camera service to several clients simultaneously. However, given the computational costs of video encoding and compression, our implementation reflects an island policy of one client per camera at any time.

The interaction between camera provider and camera service consumer in COASTCAST illustrates the propagation and confinement of capability. CURLs restrict access to cameras via non-delegation. Metadata within CURLs helps camera providers customize and restrict the functional capability granted to client code and confine a client's resource consumption. Time and rate limits on camera use help ensure that no one client unfairly monopolizes a camera. Higher-order temporal constraints can restrict use to a particular time of day or grant access conditional on events (for example, use for 30 minutes if an intrusion alarm is tripped). Client access to a camera can be withdrawn at any time by revoking the related CURL.

Camera access and use under COASTCAST illustrate why MOTILE/ISLAND is an object-capability language. A client must possess a CURL u to access a camera and only by prior access via some transitive chain of CURLs can a client acquire such a CURL (only connectivity begets connectivity). All objects are absolutely encapsulated (cameras, closures, actors, binding environments) as absent a specific access function in an execution site's binding environment no MOTILE actor can violate encapsulation. Finally, without direct access to the (de)serializer no untrusted code can synthesize a reference to an object such as a camera or an island (authority by references only). Under these conditions all MOTILE mobile code conforms to the object-capability model.

More work is needed to establish the safety and security of COAST and MOTILE/ISLAND. Continuation transfer (called out by COAST Rule 1) is unimplemented although there are hooks for it in the MOTILE compiler. In retrospect, continuation transfer is a security threat since it transmits a complete runtime stack whose lower frames may contain sensitive information that should remain on-island. Delimited continuations [49] can ameliorate this risk and are a topic for future research.

Threat analysis is a time-honored evaluation technique for security systems [10]. There can be numerous threats against a provider, including resource exhaustion, malicious exploitation of functional capability, efforts by mobile code to communicate with untrusted peers, the unsafe acquisition of additional functional or communication capability, and application-level denial of service attacks. Our analyses and experiments to date strongly suggest that the elements of execution site, CURLs, and mobile code can be manipulated to prevent, thwart, or minimize these threats. Additional

[9] Information confinement, for example decentralized information flow control [4, 9], is neither the intent nor the focus of COAST, and is not addressed here.

[10] While a single island may incorporate all three "kinds" of service it is convenient, for the purposes of analysis, to treat them as distinct. The isolation and encapsulation of COAST computations as actors bound to execution sites guarantees that this logical separation is an accurate representation of the actual behavior.

test applications (for example, access to electronic health records) are planned to confirm our hypothesis.

Finally, object-capability patterns [47] such as seal/unseal and proxy embody known security attributes and behaviors. We intend to explore their generalization to decentralized systems in the context of MOTILE/ISLAND.

10. Conclusion

Often an architectural style is based on prior patterns and practices. For COAST, however, there are few substantial prior examples to which we can refer. To gauge the efficacy of COAST we constructed both a reference implementation for the style, MOTILE/ISLAND, and a challenging application, COASTCAST, for the distribution and manipulation of soft real-time, high definition video. Maintaining the constraints of the style required the intimate cooperation of both the mobile code language MOTILE and its complementary peering infrastructure ISLAND. The consequences of the style are both subtle and pervasive and a comprehensive implementation is a significant undertaking—nonetheless our work to date is highly encouraging.

The dual goals of security and safety and the style, as an abstraction of principal security mechanisms, dominate almost all aspects of the language and infrastructure. In this case the COAST architectural style had a profound influence on MOTILE/ISLAND and accounts for all of its distinctive features: mobile code as the engine of both adaptivity and higher-order security, execution sites as a mechanism for the modulation and confinement of functional capability, and Capability URLs (CURLs) as the conveyance of communication capability in time, space, domain, and application.

Acknowledgments

This work was supported by National Science Foundation Grants CCF-0917129 and CCF-0820222.

References

[1] GStreamer: open source multimedia framework. URL `gstreamer.freedesktop.org`, February 2012.

[2] Adobe Systems Incorporated. *PostScript Language Reference Manual (Third Edition)*. Addison-Wesley, February 1999.

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

[4] O. Arden, M. D. George, et al. Sharing Mobile Code Securely with Information Flow Control. In *IEEE Symposium on Security and Privacy*, May 2012.

[5] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. ISBN 978-1-93435-600-5.

[6] P. Bagwell. *Ideal Hash Trees*. PhD thesis, Ecole Polytechnique Federale de Lausanne, Switzerland, 2001.

[7] D. J. Bernstein et al. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2:77—89, 2012.

[8] G. Bierman et al. Dynamic Rebinding for Marshalling and Update, with Destruct-time λ. In *International Conference on Functional Programming*, pages 99–110, August 2003.

[9] A. Birgisson, A. Russo, and A. Sabelfeld. Capabilities for Information Flow. In *Programming Languages and Security*, PLAS'11, pages 5:1–5:15. ACM, June 2011.

[10] M. Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, first edition, December 2002.

[11] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. The Development of the Emerald Programming Language. In *History of Programming Languages*, HOPL III, pages 11:1–11:51. ACM SIGPLAN, 2007.

[12] A. C. Bomberger et al. The KeyKOS Nanokernel Architecture. In *Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112. USENIX Association, April 1992.

[13] P. Braun and W. R. Rossak. *Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit*. Morgan Kaufmann, January 2005. ISBN 1558608176.

[14] E. Cheong and J. Liu. galsC: A Language for Event-Driven Embedded Systems. In *Proceedings of Design, Automation and Test in Europe*, DATE'05, pages 1050–1055. IEEE Computer Society, March 2005.

[15] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A Programming Model for Event-Driven Embedded Systems. In *Symposium on Applied Computing*, SAC'03, pages 698–704. ACM, 2003.

[16] T. Close. Decentralized Identification. `http://www.waterken.com/dev/YURL/`, 2001.

[17] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*, FMCO'06, pages 266–296. Springer-Verlag, 2007.

[18] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. A Characterization of Mobility and State Distribution in Mobile Code Languages. In *Proceedings of the First Workshop on Mobile Object Systems*, ECOOP '96, pages 309–318, July 1996.

[19] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In *Mobile Object Systems: Towards the Programmable Internet*, pages 91–109. Springer, 1997. ISBN 978-3-540-62852-1.

[20] T. V. Cutsem. *Ambient References: Object Designation in Mobile Ad hoc Networks*. PhD thesis, Vrije Universiteit Brussel, May 2008.

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

[22] M. Feeley and G. Lapalme. Using Closures for Code Generation. *Computer Languages*, 12(1):47–66, 1987.

[23] M. Fogus and C. Houser. *The Joy of Clojure: Thinking the Clojure Way*. Manning Publications, April 2011. ISBN 9781935182641.

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

[25] G. Germain, M. Feeley, and S. Monnier. Concurrency Oriented Programming in Termite Scheme. In *Scheme and Functional Programming Workshop*, pages 125–136, 2006.

[26] M. M. Gorlick and R. N. Taylor. Communication and Capability URLs in COAST-based Decentralized Services. In *REST: Advanced Research Topics and Practical Applications*. Springer, to appear Summer 2013.

[27] M. M. Gorlick, K. Strasser, and R. N. Taylor. COAST: An Architectural Style for Decentralized On-Demand Tailored Services. In *WICSA/ECSA'12*, pages 71–80, August 2012.

[28] D. A. Holzgang. *Display Postscript Programming*. Addison-Wesley, July 1990. ISBN 0201518147.

[29] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[30] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc Programming Language. In *Proceedings of FMOODS/FORTE 2009*, pages 1–25. Springer, June 2009.

[31] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[32] A. Mesbah and A. van Deursen. An Architectural Style for Ajax. In *WICSA'07*, pages 44–53. IEEE Computer Society, 2007.

[33] L. A. Meyerovich et al. Flapjax: A Programming Language for Ajax Applications. In *OOPSLA'09*, pages 1–20, October 2009.

[34] 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.

[35] M. S. Miller and J. S. Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. ASIAN'03, pages 224–242. Springer-Verlag, December 2003.

[36] M. S. Miller, C. Morningstar, and B. Frantz. Capability-Based Financial Instruments. In *International Conference on Financial Cryptography*, pages 349–378. Springer-Verlag, 2001.

[37] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Systems Research Laboratory, Johns Hopkins University, 2003.

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

[39] D. G. Murray and S. Hand. Scripting the Cloud with Skywriting. In *Hot Topics in Cloud Computing*, HotCloud'2010. USENIX Association, 2010.

[40] D. G. Murray et al. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Networked Systems Design and Implementation*, NSDI'11. USENIX Association, 2011.

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

[42] A. Piérard and M. Feeley. Towards a Portable and Mobile Scheme Interpreter. In *Scheme and Functional Programming Workshop*, pages 59–68, September 2007.

[43] R. K. Raj et al. Emerald: A General-Purpose Programming Language. *Software - Practice and Experience*, 21(1):91–118, 1991.

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

[45] S. Ruby, D. Thomas, and D. H. Hansson. *Agile Web Development with Rails (Fourth Edition)*. Pragmatic Bookshelf, March 2011. ISBN 1934356549.

[46] J. H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[47] M. Scheffler. Object-Capability Security in Virtual Environments. Master's thesis, Bauhaus-Universität Weimar, September 2007.

[48] E. Schulz. *A Framework for Programming Interactive Graphics in a Functional Programming Language*. PhD thesis, Freie Universit at Berlin, Berlin, Germany, 1998.

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

[50] J. S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, Philadelphia, Pennsylvania, 1999.

[51] J. S. Shapiro and N. Hardy. EROS: A Principle-Driven Operating System from the Ground Up. *IEEE Software*, 19(1): 26–33, January 2002.

[52] A. Spiessens. *Patterns of Safe Collaboration*. PhD thesis, Universite Catholique de Louvain, Louvain-la-Neuve, Belgium, February 2007.

[53] D. Stuttard and M. Pinto. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. John Wiley & Sons, 2nd edition, September 2011. ISBN 0470170778.

[54] A. S. Tanenbaum, R. van Renesse, H. van Staveren, et al. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[55] R. N. Taylor, N. Medvidovic, et al. A Component- and Message-Based Architectural Style for GUI Software. *Transactions on Software Engineering*, pages 390–406, June 1996.

[56] P. Thiemann. Higher Order Code Splicing. In *European Symposium on Programming Languages and Systems*, ESOP'99, pages 243–257. Springer-Verlag, March 1999.

[57] J. Travis and J. Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition)*. Prentice Hall, August 2006. ISBN 0131856723.

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

[59] M. Wilde et al. Swift: A Language for Distributed Parallel Scripting. *Parallel Computing*, 37(9):633–652, 2011.

[60] A. R. Yumerefendi and J. S. Chase. The Role of Accountability in Dependable Distributed Systems. In *Hot Topics in System Dependability*. USENIX Association, June 2005.